Making H5I Multi-Thread Safe:
A Sketch Design

John Mainzer
Lifeboat, LLC

9/15/22

Introduction

Unlike H5E, it was obvious from the start that H5I would have at least one dependency – specifically H5E as H5I can flag errors. While there is no fundamental reason why H5I should have any further dependencies on any other packages in the HDF5 library, in actuality there are quite a few.

As this is a preliminary sketch design for retrofitting multi-thread safety on H5I, the initial objective is to outline the issues to be addressed, and to outline possible solutions. It will be necessary to study additional packages before this sketch design can be brought to something approaching final form.

A Quick Overview of H5I

H5I exists to provide indexing services both to the HDF5 library proper, and to application programs. The basic services may be summarized as follows:

- Create and delete types of indexes. Here the type of an index indicates the type of entries the index supports. In the HDF5 library proper, types include error messages, files, data sets, etc.

- Insert, lookup, and delete entries in individual indexes. To insert an entry the user provides the type of the target index, and a void pointer to whatever data is to be associated with the entry, and receives an ID in return. This ID is used for subsequent lookups and deletions. Note that IDs have reference counts, and under normal circumstances are not deleted until their reference counts drop to 0.

  The private API also provides calls to:

  - lookup the ID associated with a given void pointer,
  - modify the void pointer associated with an existing index entry

    - insert an entry into an index with a specified ID

- Iteration and searching through indexes. Here the user provides a function and an index type. The function is executed on every ID in the target index (in the case of iteration) or until it reports success (in the case of searches).

- Miscellaneous services including incrementing and decrementing reference counts on entries and types of indexes, tests for validity, fetching the number of index entries of a give type, deletion of types of indexes with all their entries, etc.

Unfortunately, there are also miscellaneous calls to look up the file or name associated with entries in certain indexes – which result in dependencies on packages in addition to H5E. The hope is that these dependencies can be resolved either through re-architecting or through multi-thread safety requirements on the external calls – but we will not know until the target packages have been examined in greater detail.

The iteration calls also raise issues.

In iteration calls that pass the void pointer associated with the current id to the iteration call back function, H5I calls H5I_unwrap on this pointer before passing it to the callback. While this is a no-op in the public API, for some internal types of indexes (H5I_FILE, H5I_GROUP, H5I_DATASET, H5I_ATTR, and H5I_DATATYPE), the call to H5I_unwrap() results in a H5VL call, and a subsequent call into the appropriate VOL connector. While I have largely bypassed this issue for now, it is an unknown that will have to be resolved before we proceed with actual development.

The external iteration calls (H5Isearch() and H5Iiterate()) also present issues, as the HDF5 library has no control over these user supplied call backs – in principle they can make arbitrary calls back into the HDF5 library.

Multi-Thread Issues in H5I

As with H5E, leaving aside the issue of the unexpected dependencies, there appear to be no fundamental reasons why H5I can't be made multi-thread safe. That said, there are a number of issues to be dealt with. Before discussing how these challenges might be addressed, it will be useful to discuss each of these issues in greater detail.

Use of other HDF5 packages by H5I

As should be obvious from the above outline of the H5I package, there is no functional reason why H5I has to make calls to packages in the HDF5 library other than H5E for error reporting. That said, in its current implementation it does – specifically it has calls to:

- H5MM
- H5FL,
- H5E
- H5VL
- H5F

Note that while H5P is included in H5I source code files, it appears to be used only for access to a single constant used in a H5VL call – and thus it isn't listed above.

As in H5E, H5MM and H5FL are easily avoided by using the C dynamic memory allocation functions directly, and by either not maintaining free lists, or maintaining them internally.

The dependency on H5E is a larger issue, as it presents the possibility of lock ordering issues since we may need to call H5E from H5I, which in turn may call H5I. In the long term, the safest way to resolve this would be to remove H5E's dependency on H5I. However, there are several other ways which would work – albeit with greater danger of inadvertent insertion of deadlocks. Perhaps the easiest would be to make any locks used by H5I recursive.

The dependencies on H5VL and H5F spring from the above mentioned public API calls to determine the file or name associated with certain types of IDs, and the calls to H5I__unwrap() in some of the iteration calls. Indeed, the problem is potentially greater than this, as initial call trees for these public API calls indicate that H5CX and H5T are also involved, along with calls into VOL connectors. That said, the current focus is on H5I proper – thus these issues are left as known unknowns for now.

The callbacks used in both the public and private iteration calls (H5Isearch, H5Iiterate(), H5I_iterate(), etc.) are of similar concern as they have the potential to introduce dependencies, and thus potential lock ordering issues.

For the immediate objective (retrofitting multi-thread safety on H5E, H5I, H5P, H5CX, and H5VL), this is not too troubling, as we need only consider H5I iteration calls from these five packages – and an initial scan indicates that only H5VL is involved. The remainder of the HDF5 library is not an immediate concern, as any H5I iteration calls from these packages will be protected by the global lock for now.

Finally, the callbacks used in the public iteration calls (H5Isearch() and H5Iiterate()) have the potential to introduce arbitrary indirect dependencies. In principle, the free_func provided to H5Iregister_type, and the realize and discard functions provided to H5Iregister_future() present similar issues.

Multi-thread thread issues in H5I proper

A review of the H5I public, private, and developer APIs reveals the following multi-thread safety issues in the current H5I implementation.

Use of uthash to implement indexes

H5I uses uthash – a collection of macros – to implement the hash tables used to implement indexes. According to uthash documentation, uthash can be made thread safe by wrapping all uthash macros in a read / write lock. Write locks are required for operations modifying the target hash table, with read locks being sufficient for all other operations.

At present, uthash is not integrated into the HDF5 error reporting system. This is convenient, as it removes H5E from the problem, and thus avoids any lock ordering / lock recursion issues from this quarter at least. However, if HDF5 is going to stay with uthash indefinitely, this oversight should be corrected.

Use of global variables

The list of types of indexes and the next available index type are kept in global variables – with the obvious potential for race conditions and resulting data corruption.

Locks around the appropriate critical regions are an obvious solution, but they present lock ordering issues in the event of failure due to the resulting H5E calls. Locks could be dropped prior to error calls, but a solution based on atomic operations would be preferable as it would likely be faster and less error prone.

Potential race conditions in indexes and index entries

A multi-thread version of H5I must allow simultaneous operations by multiple threads. Thus it is possible that there will be simultaneous operations on a given index, or even on a given ID in that index.

The structures currently used to implement both types of indexes (H5I_type_info_t) and index entries proper (H5I_id_info_t) contain a number of fields where race conditions are an obvious issue – most notably reference counts, and (in the case of types of indexes) counts of existing IDs.

As with the global variables, locks around critical regions are the obvious solution, but solutions based on atomic operations would be preferable.

Mark and Sweep Operations

H5I allows scans and searches of types of indexes. My understanding is that to avoid breaking existing tests, H5I does not immediately delete entries deleted by the user during scans or searches. Instead, entries are marked for deletion, and then deleted in a subsequent sweep.

The same approach is used in the clear and destroy operations – which compound the problem by leaving index entries in a partially deleted state pending the subsequent sweep to complete the deletion.

This has the effect of making iteration operations of whatever type large critical regions – which is obviously undesirable in a multi-thread implementation. At a minimum, repairing this will require algorithmic changes, and may require subtle modifications to the H5I API as well.

Support for Future IDs

Addition of support for the asynchronous VOL required the addition of support for future IDs. In particular, there is code in the ID lookup function that attempts to convert a future ID into an actual ID. This operation is somewhat involved, and presents a variety of race conditions should the same future ID be looked up simultaneously by multiple threads.

The critical region here is sufficiently large, that locking may be the only practical solution.

Public API Race Conditions

The nature of an index service in a multi-thread environment makes it possible for the client application to create race conditions – for example, it will always be possible for one thread to delete an ID (or a whole index) out from under another thread.

This is an unsolvable problem from the perspective of H5I – thus H5I's responsibility will be to at least appear to execute operations in some order, and to keep indexes 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.

Design Considerations

The unexpected dependencies of H5I on other packages are worrying. Even if my initial impressions are correct, and we can both avoid all potential deadlocks from all calls from H5I to other packages, and tame the public iteration call back problem through strict constraints on the functions, the resulting code will be error prone. Since the HDF5 library is under constant development, re-insertion of deadlocks is inevitable unless we have simple, easily followed and

verified rules for avoiding them. Failure to do this will result in maintenance nightmare, as prompt detection of newly inserted deadlocks in regression testing is problematic at best.

With this point in mind, I can't but conclude that in the long term, a multi-thread safe implementation of H5I will have to hold no locks whenever it makes a call to another package – thus removing all H5I related lock ordering concerns[1]. Ideally, this would be accomplished by making the multi-thread safe implementation of H5I lock free, but small critical regions containing no function calls and protected by locks should be workable as well.

However, in the short term, time and resource constraints give us a strong incentive to minimize the changes required for the initial implementation. This, however, must be balanced with efficiency and integration concerns, as not only must the packages required to support multi-thread VOL connectors be multi-thread safe, they must also work together without deadlocks, and must facilitate sufficient performance gains as to justify their development.

While it would be best to avoid locks completely to sidestep the lock ordering question, there are several impediments:

- For thread safety, uthash requires a R/W lock around all of its macros. Thus, avoiding locks here would require replacing uthash with some other lock free data structure. Note that this is not as large a problem as it may seem, as until we integrate uthash into HDF5's error reporting system, these locks present no potential for deadlocks. Further, if we do integrate uthash properly, we should be able to drop locks before calling H5E to report errors.

- Many of the calls that modify indexes modify the relevant data structures in a variety of places – both in the data structures and the code. Making all these changes atomically while maintaining ordering constraints would likely require significant redesign of the code, and may be impossible without public API changes.

- The mark and sweep algorithm used in H5Iclear_type() and H5Idestroy_type() leave the target index in an invalid state between the mark and sweep passes. Absent algorithm changes, it will be difficult to prevent access to the target index during this period without locking.

For these reasons, the approach to making H5E multi-thread safe offered below is largely lock based, and presumes that the external dependencies can be managed well enough to make this design approach viable at least in the short term.

At best, I regard this approach to be an initial solution aimed at facilitating relatively rapid implementation of a prototype. At worst, it will prove impractical in the light of further investigation, and will have to be re-worked in part or in whole. Its primary advantage is

1 With the possible exception of H5E – if we remove that package's dependency on H5I.

simplicity. It is offered only in rough sketch form, as there is no point in putting further effort into it until the full requirements are well understood.

Making H5I Multi-Thread Safe

Given that resource considerations dictate at least a partially lock based solution for the first cut, the least expensive approach would be to put a recursive lock on H5I, allowing only one thread into H5I at a time. The recursive lock would be needed to allow for calls from H5E back into H5I.

However, this allows for no concurrency – and thus brings into question the performance requirement. Fortunately, with minimal effort, we should be able to relax this one thread in H5I at a time constraint as follows:

Global Variables

H5I uses two global variables to maintain a list of indexes (or types as they are called in the code). These are H5I_type_info_array_g, which contains pointers to the instances of H5I_type_info_t that are the root structures for each index or type, and H5I_next_type_g, which is an integer containing the index of the next free slot in H5I_type_info_t.

The operations on these global variables are both simple and tightly coupled. Thus, with minimal code changes, we should be able to maintain the consistency of these variables with atomic operations and not have to resort to locks.

The code that searches for unused entries in H5I_type_info_array_g does present issues. The obvious solution is to simply delete it, but this functionality can probably be retained via the definition of a special null pointer indicating that a cell is allocated but not valid.

Indexes (AKA Types)

The base structure for each index (or type) is an instance of H5I_type_info_t.[2] After initialization is complete, the instance of H5I_type_info_t associated with each index (or type) is pointed to by H5I_type_info_array_t[type] – here type is the integer ID of the index. NULL entries in this array indicate that the associated index has not yet been created or has been discarded.

Individual entries in each index are represented by instances of H5I_id_info_t. Each index entry is assigned a unique ID. This ID has the ID of the host index (or type) encoded in it – allowing an ID to be looked up even if the ID of the index it resides in is not available.

2 See Appendix 1 for the definition of this and all other structures mentioned in this section.

When a new ID is registered, an instance of H5I_id_info_t is allocated, initialized, and loaded with the supplied void *. An ID is allocated in the target index (or type), and the new instance of H5I_id_info_t is inserted into the target indexes hash table, with the new ID as the key.

Thus each index (or type) is represented with one instance of H5I_type_info_t, with zero or more instances of H5I_id_info_t stored in the hash table that is rooted in the instance of H5I_type_info_t.

To sketch out a locking protocol, we need to catalog the fields that are modified during the various operation on indexes – this is provided in Appendix 3.

Review of this table shows that API calls that don't modify the hash table modify few fields in the relevant instances of H5I_type_info_t and H5I_id_info_t – usually just one but occasionally two.

This suggests the following approach to ensuring multi-thread safety in the indexes (or types) proper:

- Associated a recursive read/write lock with each index (or type).
- Require all H5I API calls that modify the hash table to acquire a write lock on the target index (or type) before proceeding.
- Require all other H5I API calls to obtain a read lock on the target index (or type). Further, identify all fields of H5I_type_info_t and H5I_id_info_t modified by these calls and require them to be accessed via atomics.

This approach is clearly simple and relatively easy to implement, and provides the locking required to use uthash in a multi-thread environment. Further, if future IDs are excluded[3], it offers a significant degree of concurrency. However, there are a few points that should be discussed.

The first is efficacy – will this approach maintain the indexes in an internally consistent state, and make it appear to the clients that the H5I API calls have been executed in some sequential order? The only issue I see here is the modification of fields in H5I_type_info_t and H5I_id_info_t while only a read lock is held.

If only one field is modified, and that modification is done atomically, that should preserve the appearance of sequential execution of API calls. With two or more fields, there is the possibility

3 If an index contains a future id, a simple lookup of that id will trigger an attempt to convert it into an actual id. Along with other things, this operation requires a deletion from the hash table – and thus a write lock. Fortunately, future objects are rare, and we could modify indexes to track the number of future objects they contain – permitting use of a read lock in most cases.

of interleaving – however the only externally visible example of this is the regular and application reference counts in H5I_id_info_t. Since reference count decrements present the possibility of deletion, those operations will require a write lock. Thus only increments will be done concurrently – and it is hard to see how allowing the regular and application reference counts to be temporarily out of sync will be an issue as long as both remain positive. That said, this is a point to keep in mind as I continue to review the target packages.

A second point is the observation that by creating a separate read / write lock for each index, I have introduced a potential deadlock. This potential could be realized if any of the user callbacks supplied to the iteration API calls, or any of the external calls that I have left un-investigated attempts an H5I API call on another index. This seems improbable but not impossible – I must keep an eye out for it.

Third, note that all the iteration API calls will require a write lock – severely restricting concurrency. While this constraint could be relaxed selectively, the proposed solution is at best a temporary measure – and thus I don't see the point.

Finally, the use of recursive read/write locks should allow H5I to keep its error calls unchanged. While I haven't investigated in detail, I expect that the recursive locks will also be necessary for some iterative APIs.

Appendix 1 – H5I public API calls

After some type and macro definitions, this appendix contains a list of all the public H5I API calls, along with call trees, relevant structure definitions, and descriptions of their processing with particular emphasis on multi-thread safety issues. 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 H5Ipublic.h and H5Idev.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 not investigated H5Iget_file_id() and H5Iget_name() beyond construction of an initial call tree, as these functions contain direct calls to H5VL, and H5F, and to other packages farther down the call tree. As the focus of the current effort is H5I, I am bypassing these calls for now.

```
/**
 * Library type values.
 * \internal Library type values.  Start with `1' instead of `0' because it
 *           makes the tracing output look better when hid_t values are large
 *           numbers. Change the TYPE_BITS in H5I.c if the MAXID gets larger
 *           than 32 (an assertion will fail otherwise).
 *
 *           When adding types here, add a section to the 'misc19' test in
 *           test/tmisc.c to verify that the H5I{inc|dec|get}_ref() routines
 *           work correctly with it. \endinternal
 */
//! <!-- [H5I_type_t_snip] -->
typedef enum H5I_type_t {
    H5I_UNINIT = (-2),  /**< uninitialized type                         */
    H5I_BADID  = (-1),  /**< invalid Type                               */
    H5I_FILE   = 1,     /**< type ID for File objects                   */
    H5I_GROUP,          /**< type ID for Group objects                  */
    H5I_DATATYPE,       /**< type ID for Datatype objects               */
    H5I_DATASPACE,      /**< type ID for Dataspace objects              */
    H5I_DATASET,        /**< type ID for Dataset objects                */
    H5I_MAP,            /**< type ID for Map objects                    */
    H5I_ATTR,           /**< type ID for Attribute objects              */
    H5I_VFL,            /**< type ID for virtual file layer             */
    H5I_VOL,            /**< type ID for virtual object layer           */
    H5I_GENPROP_CLS,    /**< type ID for generic property list classes */
    H5I_GENPROP_LST,    /**< type ID for generic property lists         */
    H5I_ERROR_CLASS,    /**< type ID for error classes                  */
    H5I_ERROR_MSG,      /**< type ID for error messages                 */
    H5I_ERROR_STACK,    /**< type ID for error stacks                   */
    H5I_SPACE_SEL_ITER, /**< type ID for dataspace selection iterator   */
    H5I_EVENTSET,       /**< type ID for event sets                     */
    H5I_NTYPES          /**< number of library types, MUST BE LAST!     */
} H5I_type_t;

H5Iprivate.h:#define H5I_IS_LIB_TYPE(type) (type > 0 && type < H5I_NTYPES)

/****************************/
/* Package Private Typedefs */
/****************************/
```

```
/* ID information structure used */
typedef struct H5I_id_info_t {
   hid_t           id;                /* ID for this info */
   unsigned    count;          /* Ref. count for this ID */
   unsigned    app_count;  /* Ref. count of application visible IDs */
   const void *object;        /* Pointer associated with the ID */

   /* Future ID info */
   hbool_t                             is_future;      /* Whether this ID represents a
future object */
   H5I_future_realize_func_t realize_cb;    /* 'realize' callback for future object */
   H5I_future_discard_func_t discard_cb;  /* 'discard' callback for future object */

   /* Hash table ID fields */
   hbool_t               marked; /* Marked for deletion */
   UT_hash_handle hh;         /* Hash table handle (must be LAST) */
} H5I_id_info_t;

/* Type information structure used */
typedef struct H5I_type_info_t {
   const H5I_class_t *cls;               /* Pointer to ID class */
   unsigned                init_count;   /* # of times this type has been initialized
*/
   uint64_t                  id_count;     /* Current number of IDs held */
   uint64_t                  nextid;         /* ID to use for the next object */
   H5I_id_info_t          *last_id_info; /* Info for most recent ID looked up */
   H5I_id_info_t          *hash_table;  /* Hash table pointer for this ID type */
} H5I_type_info_t;

/****************************/
/* Library Private Typedefs */
/****************************/

typedef struct H5I_class_t {
   H5I_type_t type;       /* Class "value" for the type */
   unsigned   flags;       /* Class behavior flags */
   unsigned   reserved; /* Number of reserved IDs for this type */
                                    /* [A specific number of type entries may be
                                     * reserved to enable "constant" values to be
                                     * handed out which are valid IDs in the type,
                                     * but which do not map to any data structures
                                     * and are not allocated dynamically later.]
                                     */
   H5I_free_t free_func; /* Free function for object's of this type */
} H5I_class_t;

H5I Public API calls:

/**
 * \ingroup H5IUD
 *
 * \brief Registers an object under a type and returns an ID for it
 *
 * \param[in] type The identifier of the type of the new ID
 * \param[in] object Pointer to object for which a new ID is created
 *
 * \return \hid_t{object}
 *
 * \details H5Iregister() creates and returns a new ID for an object.
 *
 * \details The \p type parameter is the identifier for the ID type to which
 *          this new ID will belong. This identifier must have been created by
```

```
 *          a call to H5Iregister_type().
 *
 * \details The \p object parameter is a pointer to the memory which the new ID
 *          will be a reference to. This pointer will be stored by the library
 *          and returned via a call to H5Iobject_verify().
 *
 */
H5_DLL hid_t H5Iregister(H5I_type_t type, const void *object);

H5Iregister()
 +-H5I__register()
```

In a nutshell:

H5Iregister() inserts the supplied void pointer in the index of the indicated type, and returns an ID that can be used to access this void pointer at later date.

In more detail:

The function tests to see if the supplied type is a library type (i.e. one used internally). It fails if it is. Otherwise it calls H5I__register() with the app_ref. realize_cb, and discard_cb parameters set to TRUE, NULL, and NULL respectively.

H5I__register() performs some sanity checks and flags errors if they fail (interaction with H5E). Assuming success, it allocates a new instance of H5I_id_info_t via a call to H5FL_CALLOC(), constructs a new ID via a call to the H5I_MAKE macro, loads the new instance of H5I_id_info_t, and inserts it into the hash table associated with the type via a call to the HASH_ADD() macro.

Note that the object provided for insertion in the index is simply stored by reference (i.e. only a pointer is saved). Since the caller may retain a pointer to this object, the index has no control over access to it. Thus maintaining mutual exclusion on this object to avoid corruption must be the caller's responsibility.

Finally, before returning, H5I__register() makes note of the most recent ID referenced of this type.

H5I__register() returns the new ID, which is returned to the caller by H5Iregister().

Multi-Thread safety concerns:

Read access to the H5I_next_type_g and H5I_type_info_array_g global variables to validate the supplied type, and to look up a pointer (type_info) to the instance of H5I_type_info_t associated with the target index.

Read / write access to *type_info for purposes of:

- Allocating the next ID (type_info->nextid)
- Incrementing the number of IDs in the index (type_info->id_count)
- Setting the last id touched (type_info->last_id_info)
- Inserting the instance of H5I_id_info_t associated with the new id into the hash table associated with the type (type_info->hash_table)

Use of H5FL_CALLOC() to allocate the instance of H5I_id_info_t used to store the new ID and its void *.

```
/**
 * \ingroup H5IUD
 *
 * \brief Returns the object referenced by an ID
 *
 * \param[in] id ID to be dereferenced
 * \param[in] type The identifier type
 *
 *
 * \return Pointer to the object referenced by \p id on success, NULL on failure.
 *
 * \details H5Iobject_verify() returns a pointer to the memory referenced by id
 *          after verifying that \p id is of type \p type. This function is
 *          analogous to dereferencing a pointer in C with type checking.
 *
 * \note H5Iobject_verify() does not change the ID it is called on in any way
 *       (as opposed to H5Iremove_verify(), which removes the ID from its
 *       type's hash table).
 *
 * \see H5Iregister()
 *
 */
H5_DLL void *H5Iobject_verify(hid_t id, H5I_type_t type);

H5Iobject_verify()
 +-H5I_object_verify()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nutshell:

H5Iobject_verify() looks up the supplied ID & type pair, and, absent errors, returns the void pointer that was associated with this ID and type in a previous H5Iregister() or H5Iregister_future() call.

In more detail:

H5Iobject_verify() first tests to see if the supplied type either out of range, or is a library type (i.e. one used internally by the HDF5 library). It fails if either of these conditions are true.

Otherwise, it calls H5I_object_verify() with the supplied parameters, and returns that function's return value.

H5I_object_verify() verifies that the type is in range -- that is that it is greater than zero and less than the H5I_next_type_g global, and that the supplied type matches the supplied ID via H5I_TYPE() macro.

If all sanity checks pass, H5I_object_verify() calls H5I__find_id(). Absent errors, H5I__find_id() returns a pointer to the instance of H5I_id_info_t associated with the ID in the target index -- call this pointer info. H5I_object_verify() returns info->object, which is the void pointer associated with the ID in a previous register call.

H5I__find_id() obtains the type associated with the supplied ID via the H5I_type() macro, validates that it is in range, and looks up the pointer to the associated instance of H5I_type_info_t in the H5I_type_info_array_g global array, and stores this pointer in type_info.

If type_info is not NULL, and type_info->init_count is positive, it looks up the target id, checking type_info->last_id_info first, and using the HASH_FIND() macro if that fails. The result of this search is stored in the local variable id_info, and in type_info->last_id_info – both of which will be NULL if the target id is not found.

If the search is successful (i.e., id_info is not NULL), H5I__find_id() tests to see if the index entry is a "future" entry (i.e., if id_info->is_future). This is an uncommon case, used only (to my knowledge) by the asynchronous VOL.

If it is, H5I__find_id() calls the user provided realize callback (id_info->realize_cb)((void *)id_info->object, &actual_id) which was provided in H5Iregister_future() (see below).

While I have not located any documentation specifying the behavior of the user supplied realize_cb function, from context it seems that it is supposed to find the actual ID associated with the future_id if it exists, and return it in *actual_id. If (id_info->realize_cb)() fails in this for whatever reason, H5I__find_id() returns NULL.

Assuming the actual ID is found, H5I__find_id() makes note of the future object, and calls H5I__remove_common() both to delete the actual ID from the index, and to return the object associated with the actual ID. The function then sets the object associated with the future ID equal to the object associated with the actual id (i.e. sets id_info->object = actual_object;), and calls the discard_cb to discard the object previously associated with the future ID. Finally, it converts the "future" index entry to an actual entry by setting:

```
id_info->is_future  = FALSE;
id_info->realize_cb = NULL;
id_info->discard_cb = NULL;
```

Finally, id_info is returned to the caller

Multi-Thread safety concerns:

Read access to the H5I_next_type_g and H5I_type_info_array_g global variables to validate the supplied type, and to look up a pointer (type_info) to the instance of H5I_type_info_t associated with the target index.

Read / write access to *type_info for purposes of:

- Reading type_info→init_count.
- Obtain a pointer (id_info) to the instance of H5I_id_info_t associated with the target ID.
- Setting the last ID touched (type_info→last_id_info).

Read access to *id_info for purposes of reading:

- id_info→is_future and
- id_info→object.

If id_info→is_future is TRUE, matters become much more involved from a thread safety perspective as H5I__find_id() attempts to convert the future id into the actual id. Assuming that it is successful, this involves the following additional accesses to data that is accessible to other threads:

Read of the H5I_marking_g global.

R/W access to *type_info to lookup of the instance of H5I_id_info_t associated with the actual id, and (if H5I_marking_g is FALSE) removal of the associated instance of H5I_id_info_t from the hash table.

R/W access to the instance of H5I_id_info_t associated with the actual id. In particular read of the object field, and (if H5I_marking_g is TRUE), set of the marked field. Finally, (if H5I_marking_g is FALSE), free of the instance via H5FL_FREE().

R/W access to the instance of H5I_id_info_t associated with the supplied ID (*id_info). In particular:

- execution of the realize_cb() to obtain the actual id,
- execution of the discard_cp to discard id_info→object,
- set id_info→object equal to the object field of the actual id
- set id_info→realize_cb = NULL
- set id_info→discard_cb = NULL

- set id_info→is_future = FALSE

Note that these accesses are spread across a number of functions – which complicates matters further.

```
/**
 * \ingroup H5IUD
 *
 * \brief Removes an ID from its type
 *
 * \param[in] id The ID to be removed from its type
 * \param[in] type The identifier type
 *
 *
 * \return Returns a pointer to the memory referred to by \p id on success,
 *         NULL on failure.
 *
 * \details H5Iremove_verify() first ensures that \p id belongs to \p type.
 *          If so, it removes \p id from its type and returns the pointer
 *          to the memory it referred to. This pointer is the same pointer that
 *          was placed in storage by H5Iregister(). If id does not belong to
 *          \p type, then NULL is returned.
 *
 *          The \p id parameter is the ID which is to be removed from its type.
 *
 *          The \p type parameter is the identifier for the ID type which \p id
 *          is supposed to belong to. This identifier must have been created by
 *          a call to H5Iregister_type().
 *
 * \note This function does NOT deallocate the memory that \p id refers to.
 *       The pointer returned by H5Iregister() must be deallocated by the user
 *       to avoid memory leaks.
 *
 */
H5_DLL void *H5Iremove_verify(hid_t id, H5I_type_t type);

H5Iremove_verify()
 +-H5I__remove_verify()
    +-H5I_remove()
       +-H5I__remove_common()
```

In a nutshell:

Delete the index entry associated with the supplied id, and return the void pointer that was supplied at registration. If the H5I_marking_g global is TRUE, just mark the entry for deletion without actually deleting it at this time.

In more detail:

H5Iremove_verify() verifies that the supplied type is not a HDF5 library internal type. If it is, the function flags an error and returns NULL.

Assuming that this test passes, the function calls H5I__remove_verify().
and returns whatever value that function returns.

H5I__remove_verify() verifies that the supplied type and id match (via H5I_TYPE() -- returning NULL without flagging an error if they do not. If they do, it calls H5I_remove() and returns whatever value that function returns.

H5I_remove() looks up the type embedded in the supplied ID, verifies that it is valid, and looks up the associated type info. It then calls H5I__remove_common() passing a pointer to this type info and the supplied ID as parameters. The function saves the value returned by H5I__remove_common() and returns this value

Using the supplied type info, H5I__remove_common() looks up the supplied ID in the hash table associated with the supplied type_info using HASH_FIND(). If the associated instance of H5I_id_info_t is not found, the function flags an error and returns NULL.

If the associated instance of H5I_id_info_t is found, H5I__remove_common() tests the H5I_marking_g global.

If H5I_marking_g is FALSE, the instance of H5I_id_info_t is removed
from the type specific hash table via HASH_DELETE(). If H5I_marking_g is TRUE, the marked field of the instance of H5I_id_info_t is set to TRUE.

In either case, if target id was the last id of this type accessed,
type_info->last_id_info is set to NULL (thread safety), and the
return value of the function is set equal to the void pointer that
was provided when the id was registered.

If H5I_marking_g is FALSE, the target instance of H5I_id_info_t is freed via H5FL_FREE().

The number of IDs of the target type is decremented, and the function returns.

Multi-Thread safety concerns:

Read access to the H5I_next_type_g and H5I_type_info_array_g global variables to validate and look up a pointer (type_info) to the instance of H5I_type_info_t associated with the target index. Also read access to the H5I_marking_g global to determine how to implement the removal.

Read / write access to *type_info for purposes of:

- Reading type_info→init_count.
- Obtain a pointer (id_info) to the instance of H5I_id_info_t associated with the target ID.
- Reading the last id touched (type_info→last_id_info) and setting it to NULL if it equals id_info.
- Decrementing type_info->id_count

Read / write access to *id_info for purposes of reading:

- setting id_info→marked = TRUE if H5I_marking_g is TRUE
- reading id_info→object.

Freeing *id_info via H5FL_FREE().

```
/**
 * \ingroup H5I
 *
 * \brief Retrieves the type of an object
 *
 * \obj_id{id}
 *
 * \return Returns the object type if successful; otherwise #H5I_BADID.
 *
 * \details H5Iget_type() retrieves the type of the object identified by
 *          \p id. If no valid type can be determined or the identifier submitted is
 *          invalid, the function returns #H5I_BADID.
 *
 *          This function is of particular use in determining the type of
 *          object closing function (H5Dclose(), H5Gclose(), etc.) to call
 *          after a call to H5Rdereference().
 *
 * \note Note that this function returns only the type of object that \p id
 *       would identify if it were valid; it does not determine whether \p id
 *       is valid identifier. Validity can be determined with a call to
 *       H5Iis_valid().
 *
 */
H5_DLL H5I_type_t H5Iget_type(hid_t id);

H5Iget_type()
 +-H5I_get_type()
 +-H5I_object()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nutshell:

Return the type to which the supplied id belongs.

In more detail:

The type of an ID is encoded in the ID.

Thus H5Iget_type() calls H5I_get_type() which invokes the H5I_TYPE() macro to extract the type from the supplied ID, and returns this value to H5Iget_type().

H5Iget_type() then verifies that the type is valid, and that the supplied id is valid. If both tests pass, the type is returned to the caller. If not, an error is flagged and H5I_BADID is returned.

The bad type test accesses the H5I_next_type_g global.

The test for the validity of the supplied ID calls H5I_object(), which attempts to look up the ID, and return the void pointer that was supplied in the register call that created the ID. It does this via a call to H5I__find_id() which returns a pointer (info) to the instance of H5I_id_info_t associated with the id, or NULL if the search fails. If the search succeeds, the function returns info→object.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

Multi-Thread safety concerns:

Leaving aside H5I__find_id(), the only multi-thread safety concern in H5Iget_type() is read access to the H5I_next_type_g global.

In contrast, H5I__find_id() has significant multi-thread safety issues – particularly if the target id is a future id. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
**
 * \ingroup H5I
 *
 * \brief Retrieves an identifier for the file containing the specified object
 *
 * \obj_id{id}
 *
 * \return \hid_t{file}
 *
 * \details H5Iget_file_id() returns the identifier of the file associated with
 *          the object referenced by \p id.
 *
 * \note Note that the HDF5 library permits an application to close a file
 *       while objects within the file remain open. If the file containing the
 *       object \p id is still open, H5Iget_file_id() will retrieve the
 *       existing file identifier. If there is no existing file identifier for
 *       the file, i.e., the file has been closed, H5Iget_file_id() will reopen
 *       the file and return a new file identifier. In either case, the file
 *       identifier must eventually be released using H5Fclose().
 *
 * \since 1.6.3
 *
 */
H5_DLL hid_t H5Iget_file_id(hid_t id);
```

```
H5Iget_file_id()
 +-H5VL_vol_object()
 |  +-H5I_get_type()
 |  +-H5I_object()
 |  |  +-H5I__find_id()
 |  |     +-(id_info->realize_cb)()
 |  |     +-H5I__remove_common()
 |  |     +-(id_info->discard_db)()
 |  +-H5T_get_named_type()
 +-H5F_get_file_id()
    +-H5VL_object_get()
    +-H5I__find_id()
```

```
    |  +-(id_info->realize_cb)()
    |  +-H5I__remove_common()
    |  +-(id_info->discard_db)()
    +-H5VL_set_vol_wrapper()
    |  +-H5CX_get_vol_wrap_ctx()
    |  |  +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |  |     ||
    |  |    H5CX__get_context()
    |  |     +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |  |     +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    |  +-((vol_obj->connector->cls->wrap_cls.get_wrap_ctx)(vol_obj->data,
&obj_wrap_ctx)
    |  +-H5VL_conn_inc_rc()
    |  +-H5CX_set_vol_wrap_ctx()
    |     +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |        ||
    |       H5CX__get_context()
    |        +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |        +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    +-H5VL_wrap_register()
    |  +-H5CX_get_vol_wrap_ctx()
    |  |  +- ... see above
    |  +-H5T_already_vol_managed()
    |  +-H5VL__wrap_obj()
    |  |  +-H5CX_get_vol_wrap_ctx()
    |  |  |  +- ... see above
    |  |  +-H5VL_wrap_object()
    |  |     +-(connector->wrap_cls.wrap_object)(obj, obj_type, wrap_ctx)
    |  +-H5VL_register_using_vol_id()
    |     +-H5VL_new_connector()
    |     |  +-H5I_object_verify()
    |     |  |  +-H5I__find_id()
    |     |  |     +-(id_info->realize_cb)()
    |     |  |     +-H5I__remove_common()
    |     |  |     +-(id_info->discard_db)()
    |     |  +-H5I_inc_ref()
    |     |  |  +-H5I__find_id()2Y
    |     |  |     +- ... see above
    |     |  +-H5I_dec_ref()
    |     |     +-H5I__dec_ref()
    |     |        +-H5I__find_id()2Y
    |     |        |  +- ... see above
    |     |        +-(type_info->cls->free_func)((void *)info->object, request)
    |     |        +-H5I__remove_common()
    |     +-H5VL_register()
    |     |  +-H5VL__new_vol_obj()
    |     |  |  +-H5VL__wrap_obj()
    |     |  |     +- ... see above
    |     |  +-H5I_register()
    |     |     +-H5I__register()
    |     +-H5VL_conn_dec_rc()
    |        +-H5I_dec_ref()
    |           +- ... see above
    +-H5I_inc_ref()
    |  +-H5I_dec_ref()
    |     +- ... see above
    +-H5VL_reset_vol_wrapper()
       +-H5CX_get_vol_wrap_ctx()
       |  +- ... see above
       +-H5VL__free_vol_wrapper()
       |  +-(*vol_wrap_ctx->connector->cls->wrap_cls.free_wrap_ctx)(vol_wrap_ctx-
>obj_wrap_ctx)
       |  +-H5VL_conn_dec_rc()
```

```
|     +- ... see above
+-H5CX_set_vol_wrap_ctx()
   +- ... see above
```

Skipped for now due to calls to H5VL, H5CX, H5F, and H5T.

Return to this call after reviewing H5VL and H5CX.

```
/**
 * \ingroup H5I
 *
 * \brief Retrieves a name of an object based on the object identifier
 *
 * \obj_id{id}
 * \param[out] name A buffer for thename associated with the identifier
 * \param[in] size The size of the \p name buffer; usually the size of
 *                 the name in bytes plus 1 for a NULL terminator
 *
 * \return ssize_t
 *
 * \details H5Iget_name() retrieves a name for the object identified by \p id.
 *
 * \details Up to size characters of the name are returned in \p name;
 *          additional characters, if any, are not returned to the user
 *          application.
 *
 *          If the length of the name, which determines the required value of
 *          \p size, is unknown, a preliminary H5Iget_name() call can be made.
 *          The return value of this call will be the size in bytes of the
 *          object name. That value, plus 1 for a NULL terminator, is then
 *          assigned to size for a second H5Iget_name() call, which will
 *          retrieve the actual name.
 *
 *          If the object identified by \p id is an attribute, as determined
 *          via H5Iget_type(), H5Iget_name() retrieves the name of the object
 *          to which that attribute is attached. To retrieve the name of the
 *          attribute itself, use H5Aget_name().
 *
 *          If there is no name associated with the object identifier or if the
 *          name is NULL, H5Iget_name() returns 0 (zero).
 *
 * \note Note that an object in an HDF5 file may have multiple paths if there
 *       are multiple links pointing to it. This function may return any one of
 *       these paths. When possible, H5Iget_name() returns the path with which
 *       the object was opened.
 *
 * \since 1.6.0
 *
 */
H5_DLL ssize_t H5Iget_name(hid_t id, char *name /*out*/, size_t size);

H5Iget_name()
 +-H5VL_vol_object()
 |  +-H5I_get_type()
 |  +-H5I_object()
 |  |  +-H5I__find_id()
 |  |     +-(id_info->realize_cb)()
 |  |     +-H5I__remove_common()
 |  |     +-(id_info->discard_db)()
 |  +-H5T_get_named_type()
```

```
 +-H5I_get_type()
 +-H5VL_object_get()
    +-H5VL_set_vol_wrapper()
    |  +-H5CX_get_vol_wrap_ctx()
    |  |  +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |  |     ||
    |  |    H5CX__get_context()
    |  |     +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |  |     +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    |  +-((vol_obj->connector->cls->wrap_cls.get_wrap_ctx)(vol_obj->data,
&obj_wrap_ctx)
    |  +-H5VL_conn_inc_rc()
    |  +-H5CX_set_vol_wrap_ctx()
    |     +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |        ||
    |       H5CX__get_context()
    |        +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |        +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    +-H5VL__object_get()
    |  +-(cls->object_cls.get)(obj, loc_params, args, dxpl_id, req)
    +-H5VL_reset_vol_wrapper()
       +-H5CX_get_vol_wrap_ctx()
       |  +- ... see above
       +-H5VL__free_vol_wrapper()
       |  +-(*vol_wrap_ctx->connector->cls->wrap_cls.free_wrap_ctx)(vol_wrap_ctx-
>obj_wrap_ctx)
       |  +-H5VL_conn_dec_rc()
       |     +- ... see above
       +-H5CX_set_vol_wrap_ctx()
          +- ... see above
```

Skipped for now due to calls to H5VL, H5CX, and H5T.

Return to this call after reviewing H5VL and H5CX.

```
/**
 * \ingroup H5I
 *
 * \brief Increments the reference count for an object
 *
 * \obj_id{id}
 *
 * \return Returns a non-negative reference count of the object ID after
 *         incrementing it if successful; otherwise a negative value is
 *         returned.
 *
 * \details H5Iinc_ref() increments the reference count of the object
 *          identified by \p id.
 *
 *          The reference count for an object ID is attached to the information
 *          about an object in memory and has no relation to the number of
 *          links to an object on disk.
 *
 *          The reference count for a newly created object will be 1. Reference
 *          counts for objects may be explicitly modified with this function or
 *          with H5Idec_ref(). When an object ID's reference count reaches
 *          zero, the object will be closed. Calling an object ID's \c close
 *          function decrements the reference count for the ID which normally
 *          closes the object, but if the reference count for the ID has been
 *          incremented with this function, the object will only be closed when
```

```
 *          the reference count reaches zero with further calls to H5Idec_ref()
 *          or the object ID's \c close function.
 *
 *          If the object ID was created by a collective parallel call (such as
 *          H5Dcreate(), H5Gopen(), etc.), the reference count should be
 *          modified by all the processes which have copies of the ID.
 *          Generally this means that group, dataset, attribute, file and named
 *          datatype IDs should be modified by all the processes and that all
 *          other types of IDs are safe to modify by individual processes.
 *
 *          This function is of particular value when an application is
 *          maintaining multiple copies of an object ID. The object ID can be
 *          incremented when a copy is made. Each copy of the ID can then be
 *          safely closed or decremented and the HDF5 object will be closed
 *          when the reference count for that that object drops to zero.
 *
 * \since 1.6.2
 *
 */
H5_DLL int H5Iinc_ref(hid_t id);
```

```
H5Iinc_ref()
 +-H5I_inc_ref()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nut shell:

Find the target id, and increment both its regular and applications reference counts. Return the new value of the application reference count.

If the target ID is a future ID, in passing, attempt to convert it to a real ID. Note that this attempt may cause the function to fail.

In greater detail:

H5Iinc_ref() calls H5I_inc_ref() with the supplied ID, and the app_ref parameter set to TRUE.

H5I_inc_ref() calls H5I__find_id() to obtain a pointer to the instance of H5I_id_info_t associated with the ID. Assuming that this is successful, the function increments both the regular an application reference counts, and returns the new value of the application reference count.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

Multi-Thread safety concerns:

Leaving aside H5I__find_id(), the only multi-thread safety concern in H5Iget_type() is read / write access to the count and app_count fields of the target instance of H5I_id_info_t .

In contrast, H5I__find_id() has significant multi-thread safety issues – particularly if the target ID is a future ID. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
/**
 * \ingroup H5I
 *
 * \brief Decrements the reference count for an object
 *
 * \obj_id{id}
 *
 * \return Returns a non-negative reference count of the object ID after
 *         decrementing it, if successful; otherwise a negative value is
 *         returned.
 *
 * \details H5Idec_ref() decrements the reference count of the object
 *          identified by \p id.
 *
 *          The reference count for an object ID is attached to the information
 *          about an object in memory and has no relation to the number of
 *          links to an object on disk.
 *
 *          The reference count for a newly created object will be 1. Reference
 *          counts for objects may be explicitly modified with this function or
 *          with H5Iinc_ref(). When an object identifier's reference count
 *          reaches zero, the object will be closed. Calling an object
 *          identifier's \c close function decrements the reference count for
 *          the identifier which normally closes the object, but if the
 *          reference count for the identifier has been incremented with
 *          H5Iinc_ref(), the object will only be closed when the reference
 *          count reaches zero with further calls to this function or the
 *          object identifier's \c close function.
 *
 *          If the object ID was created by a collective parallel call (such as
 *          H5Dcreate(), H5Gopen(), etc.), the reference count should be
 *          modified by all the processes which have copies of the ID.
 *          Generally this means that group, dataset, attribute, file and named
 *          datatype IDs should be modified by all the processes and that all
 *          other types of IDs are safe to modify by individual processes.
 *
 *          This function is of particular value when an application is
 *          maintaining multiple copies of an object ID. The object ID can be
 *          incremented when a copy is made. Each copy of the ID can then be
 *          safely closed or decremented and the HDF5 object will be closed
 *          when the reference count for that that object drops to zero.
 *
 * \since 1.6.2
 *
 */
H5_DLL int H5Idec_ref(hid_t id);

H5Idec_ref()
 +-H5I_dec_app_ref()
    +-H5I__dec_app_ref()
       +-H5I__dec_ref()
       |  +-H5I__find_id()
       |  |  +-(id_info->realize_cb)()
       |  |  +-H5I__remove_common()
       |  |  +-(id_info->discard_db)()
```

```
|  +-(type_info->cls->free_func)((void *)info->object, request)
|  +-H5I__remove_common()
+-H5I__find_id()
   +-(id_info->realize_cb)()
   +-H5I__remove_common()
   +-(id_info->discard_db)()
```

In a nutshell:

Decrement both the regular and application reference counts on the target id. If the regular reference count drops to zero, delete the target instance from the index.

If the target ID is a future ID, in passing, attempt to convert it to a real ID. Note that this attempt may cause the function to fail.

In more detail:

After some sanity checks, H5Idec_ref() calls H5I_dec_app_ref(), and returns its return value.

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() (if it exists) to free info→object, calls H5I__remove_common() to remove *info from the index, and returns 0.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

H5I__remove_common() looks up the target ID in the index to obtain a pointer (info) to the associated instance of H5I_id_info_t.

If the H5I_marking_g global is FALSE, it removes *info from the index, and frees it via a call to H5FL_FREE().

If the H5I_marking_g global is TRUE, it sets info→marked = TRUE.

In either case, it decrements type_info→id_count, and returns info→object.

Multi-Thread safety concerns:

Leaving aside H5I__find_id() and H5I_remove_common() (which is called by H5I__find_id(), and thus included in its discussion), the multi-thread safety concerns in H5Idec_ref() are:

- Read access to H5I_type_info_array_t to obtain a pointer (type_info) to the instance of H5I_type_info_t associated with the index containing the target id.
- Execution of type_info→free_func()
- Decrement of info→count and info→app_count.

H5I__find_id() has significant multi-thread safety issues – particularly if the target ID is a future ID. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
/**
 * \ingroup H5I
 *
 * \brief Retrieves the reference count for an object
 *
 * \obj_id{id}
 *
 * \return Returns a non-negative current reference count of the object
 *         identifier if successful; otherwise a negative value is returned.
 *
 * \details H5Iget_ref() retrieves the reference count of the object identified
 *          by \p id.
 *
 *          The reference count for an object identifier is attached to the
 *          information about an object in memory and has no relation to the
 *          number of links to an object on disk.
 *
 *          The function H5Iis_valid() is used to determine whether a specific
 *          object identifier is valid.
 *
 * \since 1.6.2
 *
 */
H5_DLL int H5Iget_ref(hid_t id);

H5Iget_ref()
 +-H5I_get_ref()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nut shell:

Lookup the supplied id. If it exists, return its application reference count. If the call fails for any reason, return -1.

In more detail:

After some sanity checks, H5Iget_ref() calls H5I_get_ref() with the app_ref parameter equal to TRUE. It returns whatever value H5I_get_ref() returns.

H5I_get_ref() calls H5I__find_id() to look up the target index entry and return a pointer (info) to the instance of H5I_id_info_t associated with the id. If it is successful, H5I_get_ref() returns the current value of either the regular or the application reference count as directed by the app_ref parameter.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

Multi-Thread safety concerns:

Leaving aside H5I__find_id() , the multi-thread safety concerns in H5Iget_ref() are read access to either info→count or info→app_count, depending on the value of the app_ref parameter passed to H5I_get_ref().

H5I__find_id() has significant multi-thread safety issues – particularly if the target ID is a future ID. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
/**
 * \ingroup H5IUD
 *
 * \brief Creates and returns a new ID type
 *
 * \param[in] hash_size Minimum hash table size (in entries) used to store IDs
 *                      for the new type
 * \param[in] reserved Number of reserved IDs for the new type
 * \param[in] free_func Function used to deallocate space for a single ID
 *
 * \return Returns the type identifier on success, negative on failure.
 *
 * \details H5Iregister_type() allocates space for a new ID type and returns an
 *          identifier for it.
 *
 *          The \p hash_size parameter indicates the minimum size of the hash
 *          table used to store IDs in the new type.
 *
 *          The \p reserved parameter indicates the number of IDs in this new
 *          type to be reserved. Reserved IDs are valid IDs which are not
 *          associated with any storage within the library.
 *
```

```
 *          The \p free_func parameter is a function pointer to a function
 *          which returns an herr_t and accepts a \c void*. The purpose of this
 *          function is to deallocate memory for a single ID. It will be called
 *          by H5Iclear_type() and H5Idestroy_type() on each ID. This function
 *          is NOT called by H5Iremove_verify(). The \c void* will be the same
 *          pointer which was passed in to the H5Iregister() function. The \p
 *          free_func function should return 0 on success and -1 on failure.
 *
 */
H5_DLL H5I_type_t H5Iregister_type(size_t hash_size, unsigned reserved, H5I_free_t
free_func);

H5Iregister_type()
 +-H5I_register_type()
```

In a nutshell:

Create a new type of index as specified, and return its ID. On failure, return a negative value.

In more detail:

H5Iregister_type() first attempts to allocate an ID for the new type.

If H5I_next_type_g is less than H5I_MAX_NUM_TYPES, it sets new_type = H5I_next_type_g and then increments H5I_next_type_g.

If this approach fails, it scans the global H5I_type_info_array_g array skipping library defined types looking for a NULL entry. If it finds one, it sets new_type equal to the index of the NULL entry.

If this second approach fails, the function fails.

Assuming an ID can be allocated for the new type, H5Iregister_type() allocates an instance of H5I_class_t via a call to H5MM_calloc(), initializes it with the data provided and the new ID, and then calls H5I_register_type() to perform the actual registration.

H5I_register_type() allows multiple registrations of a given type – of which more in the discussion of multi-thread safety concerns.

After some initial sanity checks, H5I_register_type() reads the new id from the supplied instance of H5I_class_t (cls->type), and then examines the global H5I_type_info_array_g array at that index (H5I_type_info_array_g[cls->type]). If that index contains NULL, it allocates a new instance H5I_type_info_t (via H5MM_calloc), sets type_info to point to it, and sets H5I_type_info_array_g[cls->type] = type_info.

If H5I_type_info_array_g[cls->type] is not NULL, the function sets type_info = H5I_type_info_array_g[cls->type].

The function tests type_info->init_count. If it is zero it initializes *type_info.

Finally, it increments type_info->init_count and returns.

Multi-Thread safety concerns:

In the absence of any access control on the H5I_next_type_g and H5I_type_info_array_g global variables, the current algorithm for allocating type IDs has a number of race conditions which appear to make it possible for a given ID to be allocated more than once – not to mention the possibility that other threads that require only read access to these variables will see them in an inconsistent state.

In addition, H5I_register_type() acceptance of multiple registrations of a given type present potential race conditions potentially resulting in data corruption unless calls for a given type are somehow serialized.

```
/**
 * \ingroup H5IUD
 *
 * \brief Deletes all identifiers of the given type
 *
 * \param[in] type Identifier of identifier type which is to be cleared of identifiers
 * \param[in] force Whether or not to force deletion of all identifiers
 *
 * \return \herr_t
 *
 * \details H5Iclear_type() deletes all identifiers of the type identified by
 *          the argument \p type.
 *
 *          The identifier type's free function is first called on all of these
 *          identifiers to free their memory, then they are removed from the
 *          type.
 *
 *          If the \p force flag is set to false, only those identifiers whose
 *          reference counts are equal to 1 will be deleted, and all other
 *          identifiers will be entirely unchanged. If the force flag is true,
 *          all identifiers of this type will be deleted.
 *
 */
H5_DLL herr_t H5Iclear_type(H5I_type_t type, hbool_t force);

/* User data for H5I__clear_type_cb */
typedef struct {
    H5I_type_info_t *type_info; /* Pointer to the type's info to be cleared */
    hbool_t          force;     /* Whether to always remove the ID */
    hbool_t          app_ref;   /* Whether this is an appl. ref. call */
} H5I_clear_type_ud_t;

H5Iclear_type()
 +-H5I_clear_type()
    +-H5I__mark_node()
       +-H5I__mark_node()
```

In a nutshell:

Delete all IDs of the target type with ref count 1 (i.e. not in current use) from the index. If the force flag is set, delete all IDs of the target type regardless of ref count.

In more detail:

H5Iclear_type() verifies that the supplied type is not a library type, and then calls H5I_clear_type() with the supplied type and force parameters, and with the app_ref parameter set to TRUE. It returns whatever value H5I_clear_type() returns.

Using the H5I_next_type_g and H5I_type_info_array_g global variables, H5I_clear_type() validates the supplied type, and looks up a pointer (udata.type_info) to the instance of H5I_type_info_t associated with the target type, and loads it into its user data (instance of H5I_clear_type_ud_t – see above) along with the force and app_ref parameters.

It then sets the H5I_marking_g global to TRUE, and uses the uthash HASH_ITER macro to set up a for loop to scan through all the entries in the hash table associated with the target id type -- calling H5I__mark_node() on each such entry.

H5I__mark_node() examines the supplied instance of H5I_id_info_t. If the force flag is set or if its ref count (info->count) is no greater than 1, it marks it for deletion (NOTE: it the app_ref flag is set, the ref count condition is changed to ref count - application ref count (info->app_count) <= 1).

If either of the above conditions are met, H5I__mark_node() discards the target of the void * that was provided on registration if a free function is provided for the type (The info->discard_cb is used for future objects if provided). The marked flag (info->marked) is then set to TRUE, and the id count (type_info→id_count) is decremented before H5I__mark_node() returns.

After the initial marking scan through the hash table associated with the target type, H5I_clear_type() sets the H5I_marking_g global to FALSE, and then uses HASH_ITER to setup a second scan, running the HASH_DELETE macro on every id whose marked flag was set in the prior scan. After removal from the hash table, each instance of H5I_id_info_t is freed via H5FL_FREE().

Assuming no errors have been detected, H5I_clear_type() then returns.

Multi-Thread safety concerns:

H5Iclear_type() has the multi-thread safety issues of accessing data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read/write access to the target instance of H5I_type_info_t, and
- Read/write/delete access to instance of H5I_id_info_t in the target index.

- Use of H5FL_FREE()

In addition, H5Iclear_type() displays a fundamental issue not seen so far in this pass through the public H5I API – specifically, H5I__mark_node() leaves index entries in a half deleted state pending their eventual full deletion in the second pass through the hash table. Absent changes in the algorithm, the only solution that comes to mind is to treat the entire H5Iclear_type() call as critical region.

From discussion with Dana, I gather that this mark and sweep approach was adopted to avoid issues with the regression test for H5Iitterate() (discussed below). Must investigate this to see if changes to H5I public API semantics would be required to avoid this issue.

```
/**
 * \ingroup H5IUD
 *
 * \brief Removes an identifier type and all identifiers within that type
 *
 * \param[in] type Identifier of identifier type which is to be destroyed
 *
 * \return \herr_t
 *
 * \details H5Idestroy_type deletes an entire identifier type \p type. All
 *          identifiers of this type are destroyed and no new identifiers of
 *          this type can be registered.
 *
 *          The type's free function is called on all of the identifiers which
 *          are deleted by this function, freeing their memory. In addition,
 *          all memory used by this type's hash table is freed.
 *
 *          Since the H5I_type_t values of destroyed identifier types are
 *          reused when new types are registered, it is a good idea to set the
 *          variable holding the value of the destroyed type to #H5I_UNINIT.
 *
 */
H5_DLL herr_t H5Idestroy_type(H5I_type_t type);
```

```
H5Idestroy_type()
 +-H5I__destroy_type()
    +-H5I_clear_type()
     | +-H5I__mark_node()
     |    +-H5I__mark_node()
     +-H5MM_xfree_const()
     |    +-H5MM_xfree()
     +-H5MM_xfree()
```

In a nutshell:

Discard the target index type, along with all IDs that may reside in the target index.

In more detail:

H5Idestroy_type() verifies that the target type is not an internal HDF5 library type, and then calls H5I__destroy_type(), returning whatever value that function returns.

H5I__destroy_type() validates the type id reading the H5I_next_type_g global in the process, and then gets a pointer to the target type from the indicated entry in the H5I_type_info_array_g global array of pointer to H5I_type_info_t. It verifies that this pointer is not NULL, and that type_info->init_count is positive.

If these tests pass, H5I__destroy_type() then calls H5I_clear_type() with the force parameter set to TRUE, and the app_ref parameter set to FALSE. Any error return from this call is ignored.

H5I_clear_type() is discussed in detail in H5Iclear_type() above. For purposes of this discussion, it should be sufficient to note that with the above parameters, it will discard all IDs of the target type.

On H5I_clear_type()'s return, H5I__destroy_type() tests to see if the H5I_CLASS_IS_APPLICATION flag is set in type_info->cls->flags. If so, it frees type_info→cls via a call to H5MM_xfree_const().

The hash table is then freed via the HASH_CLEAR() macro, followed by the *type_info itself (via H5MM_xfree()).

Finally, the target entry in the H5I_type_info_array_g global array is set to NULL just before the function returns.

Multi-Thread safety concerns:

Structurally, H5Idestroy_type() is very similar to H5Iclear_type(), and thus has all the multi-thread concerns surrounding that call with the addition of write access to the H5I_type_info_array_g global array.

```
/**
 * \ingroup H5IUD
 *
 * \brief Increments the reference count on an ID type
 *
 * \param[in] type The identifier of the type whose reference count is to be
incremented
 *
 * \return Returns the current reference count on success, negative on failure.
 *
 * \details H5Iinc_type_ref() increments the reference count on an ID type. The
 *          reference count is used by the library to indicate when an ID type
 *          can be destroyed.
 *
 *          The type parameter is the identifier for the ID type whose
 *          reference count is to be incremented. This identifier must have
 *          been created by a call to H5Iregister_type().
 *
 */
H5_DLL int H5Iinc_type_ref(H5I_type_t type);

H5Iinc_type_ref()
 +-H5I__inc_type_ref()
```

In a nutshell:

Increment the reference count on the indicated index.

In more detail:

After accessing the H5I_next_type_g global variable to validate the supplied type and verify that it is not a library type, H5Iinc_type_ref() calls H5I__inc_type_ref(), and returns whatever that function returns.

H5I__inc_type_ref() does sanity checks on the supplied type. If they pass, it gets a pointer (type_info) to the target type from the indicated entry in the H5I_type_info_array_g global array. It verifies that this pointer is not NULL, and if so, increments type_info->init_count, and returns the new value of that field.

Multi-Thread safety concerns:

H5Iinc_type_ref() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read/write access to the target instance of H5I_type_info_t, specifically the init_count field.

```
/**
 * \ingroup H5IUD
 *
 * \brief Decrements the reference count on an identifier type
 *
 * \param[in] type The identifier of the type whose reference count is to be
decremented
 *
 * \return Returns the current reference count on success, negative on failure.
 *
 * \details H5Idec_type_ref() decrements the reference count on an identifier
 *          type. The reference count is used by the library to indicate when
 *          an identifier type can be destroyed. If the reference count reaches
 *          zero, this function will destroy it.
 *
 *          The type parameter is the identifier for the identifier type whose
 *          reference count is to be decremented. This identifier must have
 *          been created by a call to H5Iregister_type().
 *
 */
H5_DLL int H5Idec_type_ref(H5I_type_t type);

H5Idec_type_ref()
 +-H5I_dec_type_ref()
      +-H5I_clear_type()
       | +-H5I__mark_node()
       |    +-H5I__mark_node()
```

```
        +-H5MM_xfree_const()
        |    +-H5MM_xfree()
        +-H5MM_xfree()
```

In a nutshell:

Decrement the reference count of the target type. If the index count drops to zero, discard all IDs in the target index, and then discard the index type as well.

In greater detail:

After verifying that the supplied type is not a library type, H5Idec_type_ref() calls H5I_dec_type_ref(), and returns whatever value that function returns.

Using the H5I_next_type_g global, H5I_dec_type_ref() does sanity checks on the supplied type. If they pass, it gets a pointer (type_info) to the target type from the indicated entry in the H5I_type_info_array_g global array. It verifies that this pointer is not NULL, and if so, it tests to see if type_info→init_count is 1.

If it is not, it decrements type_info->init_count and returns that value.

If it is, it calls H5I__destroy_type() and returns zero.

H5I__destroy_type() is discussed in detail in H5Idestroy_type() above, and thus need not be discussed here.

Multi-Thread safety concerns:

As per H5Idestroy_type().

```
**
 * \ingroup H5IUD
 *
 * \brief Retrieves the reference count on an ID type
 *
 * \param[in] type The identifier of the type whose reference count is to be retrieved
 *
 * \return Returns the current reference count on success, negative on failure.
 *
 * \details H5Iget_type_ref() retrieves the reference count on an ID type. The
 *          reference count is used by the library to indicate when an ID type
 *          can be destroyed.
 *
 *          The type parameter is the identifier for the ID type whose
 *          reference count is to be retrieved. This identifier must have been
 *          created by a call to H5Iregister_type().
 *
 */
H5_DLL int H5Iget_type_ref(H5I_type_t type);
```

```
H5Iget_type_ref()
 +-H5I__get_type_ref()
```

In a nutshell:

Return the reference count on the target type.

In greater detail:

After accessing the H5I_next_type_g global variable to validate the supplied type and verify that it is not a library type, H5Iget_type_ref() calls H5I__get_type_ref(), and returns whatever value that function returns.

H5I__get_type_ref() does sanity checks on the supplied type. If they pass, it gets a pointer (type_info) to the target type from the indicated entry in the H5I_type_info_array_g global array. It verifies that this pointer is not NULL, and if so, it returns the current value of type_info->init_count.

Multi-Thread safety concerns:

H5Iget_type_ref() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read access to the target instance of H5I_type_info_t, specifically the init_count field.

```
/**
 * \ingroup H5IUD
 *
 * \brief Finds the memory referred to by an ID within the given ID type such
 *        that some criterion is satisfied
 *
 * \param[in] type The identifier of the type to be searched
 * \param[in] func The function defining the search criteria
 * \param[in] key A key for the search function
 *
 * \return Returns a pointer to the object which satisfies the search function
 *         on success, NULL on failure.
 *
 * \details H5Isearch() searches through a given ID type to find an object that
 *          satisfies the criteria defined by \p func. If such an object is
 *          found, the pointer to the memory containing this object is
 *          returned. Otherwise, NULL is returned. To do this, \p func is
 *          called on every member of type \p type. The first member to satisfy
 *          \p func is returned.
 *
 *          The \p type parameter is the identifier for the ID type which is to
 *          be searched. This identifier must have been created by a call to
 *          H5Iregister_type().
 *
```

```
 *          The parameter \p func is a function pointer to a function which
 *          takes three parameters. The first parameter is a \c void* and will
 *          be a pointer to the object to be tested. This is the same object
 *          that was placed in storage using H5Iregister(). The second
 *          parameter is a hid_t and is the ID of the object to be tested. The
 *          last parameter is a \c void*. This is the \p key parameter and can
 *          be used however the user finds helpful, or it can be ignored if it
 *          is not needed. \p func returns 0 if the object it is testing does
 *          not pass its criteria. A non-zero value should be returned if the
 *          object does pass its criteria. H5I_search_func_t is defined in
 *          H5Ipublic.h and is shown below.
 *          \snippet this H5I_search_func_t_snip
 *          The \p key parameter will be passed to the search function as a
 *          parameter. It can be used to further define the search at run-time.
 *
 */
H5_DLL void *H5Isearch(H5I_type_t type, H5I_search_func_t func, void *key);


typedef struct {
    H5I_search_func_t app_cb;  /* Application's callback routine */
    void *            app_key; /* Application's "key" (user data) */
    void *            ret_obj; /* Object to return */
} H5I_search_ud_t;


/* User data for iterator callback for ID iteration */
typedef struct {
    H5I_search_func_t user_func;  /* 'User' function to invoke */
    void *            user_udata; /* User data to pass to 'user' function */
    hbool_t           app_ref;    /* Whether this is an appl. ref. call */
    H5I_type_t        obj_type;   /* Type of object we are iterating over */
} H5I_iterate_ud_t;


H5Isearch()
 +-H5I_iterate()
    +-H5I__iterate_cb()
       +-H5I__unwrap()
       |  +-H5VL_object_data()
       |  |  +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |  |     +- ??? -- must investigate
       |  +-H5T_get_actual_type()
       |     +-H5VL_object_data()
       |        +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |           +- ??? -- must investigate
       +-func() -- user function provided in call to H5Iserach()
```

In a nutshell:

Scan through the IDs of the target type, running the supplied search function on each. Return when the search function returns either success or error.

In greater detail:

On entry, H5Isearch() verifies that the supplied type is not a library type, and then initializes an instance of H5I_search_ud_t as follows:

```
udata.app_cb  = func;
udata.app_key = key;
udata.ret_obj = NULL;
```

It then calls H5I_iterate() with the supplied type, H5I__search_cb() as the func parameter, a pointer to the instance of H5I_search_ud_t as the udata parameter, and with the app_ref parameter set to TRUE. The return value of H5I_iterate() is ignored, and the function returns udata.ret_obj to the caller.

On entry, H5I_iterate() does some sanity checks, looks up the target type's instance of H5I_type_info_t in the global H5I_type_info_array_g, and stores that pointer in type_info.

If type_info is not NULL, type_info->init_count > 0, and type_info->id_count > 0, H5I_iterate() proceeds as follows:

First, it sets its user data in an instance H5I_iterate_ud_t and initializes it as follows:

```
iter_udata.user_func  = func;    // H5I__search_cb in this case
iter_udata.user_udata = udata;   // udata from H5Isearch()
iter_udata.app_ref    = app_ref; // TRUE in this case
iter_udata.obj_type   = type;    // the target index type
```

The the function uses the HASH_ITER macro to set up a for loop to iterate through all IDs in the target type. For each such ID that is not marked for deletion (i.e. the marked field in the associated instance of H5I_id_info_t is not set), H5I_iterate() calls H5I__iterate_cb() with the item parameter pointing to the instance of H5I_id_info_t associated with the current ID, NULL for the key parameter, and the udata parameter pointing to the instance of H5I_iterate_ud_t just initialized. If H5I_iterate() returns either H5_ITER_STOP or H5_ITER_ERROR, H5I_iterate() breaks out of the for loop and returns – flagging an error in the latter case.

H5I__iterate_cb() checks to see if the app_ref field of the user data provided by H5I_iterate() is TRUE, and if application reference count on the target instance of H5I_id_info_t (*_item) is positive.

If these tests pass, H5I__iterate_cb() calls H5I__unwrap() on the void pointer that was passed to H5Iregister(), and passes the result to the search function (instance of H5I_search_func_t) that was passed into H5Isearch(). If the search function returns a positive value, the return value of H5I__iterate_cb() is set to H5_ITER_STOP, if a negative value, H5_ITER_ERROR. Otherwise, H5I__iterate_cb() returns H5_ITER_CONT.

In the context of external API's, H5I__unwrap() is a no-op. It simply returns the void pointer that was passed to it in the object parameter.

However, for library IDs of H5I_FILE, H5I_GROUP, H5I_DATASET, or H5I_ATTR type, the void pointer is cast tor a pointer to H5VL_object_t, and passed to H5VL_object_data(). The return value of H5VL_object_data() is returned to the caller. Similarly, if the ID is of type

H5I_DATATYPE, the void pointer is cast to a pointer to H5T_t and passed to H5VL_object_data() -- where
if may be passed to H5VL_object_data(). Again, the value from H5T_get_actual_type() is returned to the caller.

Tracing through H5VL_object_data() and its subsequent calls to see what is going on here is beyond the scope of the current investigation. However, this will have to be addressed when H5VL is investigated, if not before.

Multi-Thread safety concerns:

H5Isearch() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read access to the target instance of H5I_type_info_t.
- Read access to every instance of H5I_id_info_t In the target index

In addition, there are the unknown issues raised by H5I__unwrap() as discussed above.

Leaving aside the issues raised by H5I__unwrap(), H5I has little control over the behavior of the user provided search function.

Finally, there is the fact that H5Isearch() iterates through the IDs in the target index. While H5Isearch() doesn't have the obvious data consistency issues of H5Iclear() and H5Idestroy(), it will probably be convenient to apply the same solutions to it as well.

```
/**
 * \ingroup H5IUD
 *
 * \brief Calls a callback for each member of the identifier type specified
 *
 * \param[in] type The identifier type
 * \param[in] op The callback function
 * \param[in,out] op_data The data for the callback function
 *
 * \return The last value returned by \p op
 *
 * \details H5Iiterate() calls the callback function \p op for each member of
 *          the identifier type \p type. The callback function type for \p op,
 *          H5I_iterate_func_t, is defined in H5Ipublic.h as:
 *          \snippet this H5I_iterate_func_t_snip
 *          \p op takes as parameters the identifier and a pass through of
 *          \p op_data, and returns an herr_t.
 *
 *          A positive return from op will cause the iteration to stop and
 *          H5Iiterate() will return the value returned by \p op. A negative
 *          return from \p op will cause the iteration to stop and H5Iiterate()
 *          will return failure. A zero return from \p op will allow iteration
```

```
 *          to continue, as long as there are other identifiers remaining in
 *          type.
 *
 * \since 1.12.0
 *
 */
H5_DLL herr_t H5Iiterate(H5I_type_t type, H5I_iterate_func_t op, void *op_data);

typedef herr_t (*H5I_iterate_func_t)(hid_t id, void *udata);

typedef struct {
    H5I_iterate_func_t op;      /* Application's callback routine */
    void *             op_data; /* Application's user data */
} H5I_iterate_pub_ud_t;


H5Iiterate()
 +-H5I_iterate()
    +-H5I__iterate_cb()
       +-H5I__unwrap()
       |  +-H5VL_object_data()
       |  |  +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |  |     +- ??? -- must investigate
       |  +-H5T_get_actual_type()
       |     +-H5VL_object_data()
       |        +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |           +- ??? -- must investigate
       +-op() -- user function provided in call to H5Iiterate()
```

In a nutshell:

Scan through the IDs of the target type, running the supplied function on each. Return early if the supplied function returns either success or error.

In greater detail:

H5Iiterate() initializes an instance of H5I_iterate_pub_ud_t as follows:

```
int_udata.op           = op;
int_udata.op_data = op_data;
```

and then calls H5I_iterate() with the supplied type as the type parameter, H5I__iterate_pub_cb as the func parameter, the int_udata as the udata, and TRUE as the app_ref parameter. H5Iiterate() returns the value returned by H5I_iterate().

From this point, H5Iiterate is very similar to H5Isearch()

H5I_iterate() is the same as in H5Isearch() with the exception of the initialization of its instance of H5I_iterate_ud_t:

```
        iter_udata.user_func  = func;        //
H5I__iterate_pub_cb in this case
        iter_udata.user_udata = udata;   // udata from
H5Iiterate() -- this is
```

```
                                                                    //
the delta
        iter_udata.app_ref    = app_ref;   // TRUE in this case
        iter_udata.obj_type   = type;       // the target index
type
```

Similarly, H5I__iterate_cb() functions much as it does in H5Isearch(), the difference being in the user function called (H5I__iterate_pub_cb() vs H5I__search_cp()) and the user data.

H5I__iterate_pub_cb() is simpler than H5I__search_cb(). It just calls the function supplied to H5Isearch() with the current ID and udata supplied to H5Isearch() as parameters. It translates the return value to either H5_ITER_STOP, H5_ITER_ERROR, or H5I_ITER_CONT as appropriate, and returns.

Multi-Thread safety concerns:

Much the same as H5Isearch(), with additional concerns about the function supplied to H5Iiterate(). Since this function is only supplied with the ID of the index entry under examination, it will probably have to make a H5I call to obtain the associated data – complicating the multi-thread safety problem.

```
/**
 * \ingroup H5IUD
 *
 * \brief Returns the number of identifiers in a given identifier type
 *
 * \param[in] type The identifier type
 * \param[out] num_members Number of identifiers of the specified identifier type
 *
 * \return \herr_t
 *
 * \details H5Inmembers() returns the number of identifiers of the identifier
 *          type specified in \p type.
 *
 *          The number of identifiers is returned in \p num_members. If no
 *          identifiers of this type have been registered, the type does not
 *          exist, or it has been destroyed, \p num_members is returned with
 *          the value 0.
 *
 */
H5_DLL herr_t H5Inmembers(H5I_type_t type, hsize_t *num_members);

H5Inmembers()
 +-H5I_nmembers()
```

In a nutshell:

Return the number of IDs in the target type.

In greater detail:

After accessing the H5I_next_type_g and the H5I_type_info_array_g global variables to validate the supplied type and verify that it is not a library type, H5Inmembers() calls H5I_nmembers() to obtain the current number of entries in the target type, and return that value in Inum_members.

H5I_nmembers() validates the type again, and then obtains a pointer (type_info) to the target instance of H5I_type_info_t. If this pointer is NULL, or if type_info→init_count is non-positive, H5I_nmembers() returns zero. Otherwise, it returns type_info->id_count.

Multi-Thread safety concerns:

H5Iget_type_ref() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read access to the target instance of H5I_type_info_t, specifically the init_count and id_count fields.

```
/**
 * \ingroup H5IUD
 *
 * \brief Determines whether an identifier type is registered
 *
 * \param[in] type Identifier type
 *
 * \return \htri_t
 *
 * \details H5Itype_exists() determines whether the given identifier type,
 *          \p type, is registered with the library.
 *
 * \since 1.8.0
 *
 */
H5_DLL htri_t H5Itype_exists(H5I_type_t type);
```

```
H5Itype_exists()
```

In a nutshell:

Return TRUE if the specified type exists, and FALSE otherwise.

In greater detail:

Look up the entry in the global H5I_type_info_array_g array indicated by the supplied type. Return FALSE if this entry is NULL, and TRUE otherwise.

Multi-Thread safety concerns:

H5Itype_exists() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,

```
/**
 * \ingroup H5I
 *
 * \brief Determines whether an identifier is valid
 *
 * \obj_id{id}
 *
 * \return \htri_t
 *
 * \details H5Iis_valid() determines whether the identifier \p id is valid.
 *
 * \details Valid identifiers are those that have been obtained by an
 *          application and can still be used to access the original target.
 *          Examples of invalid identifiers include:
 *          \li Out of range values: negative, for example
 *          \li Previously-valid identifiers that have been released:
 *              for example, a dataset identifier for which the dataset has
 *              been closed
 *
 *          H5Iis_valid() can be used with any type of identifier: object
 *          identifier, property list identifier, attribute identifier, error
 *          message identifier, etc. When necessary, a call to H5Iget_type()
 *          can determine the type of the object that \p id identifies.
 *
 * \since 1.8.3
 *
 */
H5_DLL htri_t H5Iis_valid(hid_t id);

H5Iis_valid()
 +-H5I__find_id()
    +-(id_info->realize_cb)()
    +-H5I__remove_common()
    +-(id_info->discard_db)()
```

In a nutshell:

Look up the supplied ID. If it doesn't exist, of if it has a zero application ref count (i.e. it is HDF5 library internal), return FALSE. Otherwise return TRUE.

In greater detail:

H5Iis_valid() calls H5I__find_id() to look up a pointer to the instance of H5I_id_info_t (info) associated with the supplied ID. See H5Iobject_verify() for a discussion of this call.

If info is NULL, or if info→app_count is zero, H5Iis_valid() returns FALSE. Otherwise, it returns TRUE.

Multi-Thread safety concerns:

See H5Iobject_verify().

```
/**
 * \ingroup H5I
 *
 * \brief Registers a "future" object under a type and returns an ID for it
 *
 * \param[in] type The identifier of the type of the new ID
 * \param[in] object Pointer to "future" object for which a new ID is created
 * \param[in] realize_cb Function pointer to realize a future object
 * \param[in] discard_cb Function pointer to destroy a future object
 *
 * \return \hid_t{object}
 *
 * \details H5Iregister_future() creates and returns a new ID for a "future" object.
 *          Future objects are a special kind of object and represent a
 *          placeholder for an object that has not yet been created or opened.
 *          The \p realize_cb will be invoked by the HDF5 library to 'realize'
 *          the future object as an actual object.  A call to H5Iobject_verify()
 *          will invoke the \p realize_cb callback and if it successfully
 *          returns, will return the actual object, not the future object.
 *
 * \details The \p type parameter is the identifier for the ID type to which
 *          this new future ID will belong. This identifier may have been created
 *          by a call to H5Iregister_type() or may be one of the HDF5 pre-defined
 *          ID classes (e.g. H5I_FILE, H5I_GROUP, H5I_DATASPACE, etc).
 *
 * \details The \p object parameter is a pointer to the memory which the new ID
 *          will be a reference to. This pointer will be stored by the library,
 *          but will not be returned to a call to H5Iobject_verify() until the
 *          \p realize_cb callback has returned the actual pointer for the object.
 *
 *          A  NULL value for \p object is allowed.
 *
* \details The \p realize_cb parameter is a function pointer that will be
 *          invoked by the HDF5 library to convert a future object into an
 *          actual object.   The \p realize_cb function may be invoked by
 *          H5Iobject_verify() to return the actual object for a user-defined
 *          ID class (i.e. an ID class registered with H5Iregister_type()) or
 *          internally by the HDF5 library in order to use or get information
 *          from an HDF5 pre-defined ID type.  For example, the \p realize_cb
 *          for a future dataspace object will be called during the process
 *          of returning information from H5Sget_simple_extent_dims().
 *
 *          Note that although the \p realize_cb routine returns
 *          an ID (as a parameter) for the actual object, the HDF5 library
 *          will swap the actual object in that ID for the future object in
 *          the future ID.  This ensures that the ID value for the object
 *          doesn't change for the user when the object is realized.
 *
 *          Note that the \p realize_cb callback could receive a NULL value
 *          for a future object pointer, if one was used when H5Iregister_future()
 *          was initially called.  This is permitted as a means of allowing
 *          the \p realize_cb to act as a generator of new objects, without
```

```
 *          requiring creation of unnecessary future objects.
 *
 *          It is an error to pass NULL for \p realize_cb.
 *
 * \details The \p discard_cb parameter is a function pointer that will be
 *          invoked by the HDF5 library to destroy a future object.  This
 *          callback will always be invoked for _every_ future object, whether
 *          the \p realize_cb is invoked on it or not.  It's possible that
 *          the \p discard_cb is invoked on a future object without the
 *          \p realize_cb being invoked, e.g. when a future ID is closed without
 *          requiring the future object to be realized into an actual one.
 *
 *          Note that the \p discard_cb callback could receive a NULL value
 *          for a future object pointer, if one was used when H5Iregister_future()
 *          was initially called.
 *
 *          It is an error to pass NULL for \p discard_cb.
 *
 * \note The H5Iregister_future() function is primarily targeted at VOL connector
 *          authors and is _not_ designed for general-purpose application use.
 *
 */
H5_DLL hid_t H5Iregister_future(H5I_type_t type, const void *object,
                                H5I_future_realize_func_t realize_cb,
                                H5I_future_discard_func_t discard_cb);

H5Iregister_future()
 +-H5I__register()
```

In a nutshell:

H5Iregister_future() inserts the supplied void pointer in the index of the indicated type, marks the entry as a future object, and decorates it with the supplied realize and discard callbacks. The function returns an ID that can be used to access this void pointer (or its realized version) at a later date.

In more detail:

Test to see if the supplied type is a library type (i.e. one used internally. Fail if it is. Similarly, verify that the realize and discard callbacks are defined. If all tests pass, call H5I__register() with the app_ref. realize_cb, and discard_cb parameters set to TRUE, and the supplied values respectively.

Further processing is as per H5Iregister(), with the exception that the is_future flag is set to TRUE, not FALSE.

Multi-Thread safety concerns:

As per H5Iregister().

Appendix 2 – H5I private API calls

In addition to its public and developer APIs, H5I also has a private API providing indexing 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 H5I multi-thread safe so it can be safely called by multiple threads in multi-thread safe VOL connectors, at first glance, the internal H5I API is not relevant to this effort. However, the internal H5I API is used by other packages – including those necessary to support multi-thread VOL connectors.

The list of internal H5I API calls below is taken from H5Iprivate.h. Most entries are annotated with a reference to the relevant public API call. Those with no public API cognate have more extensive annotations.

```
H5_DLL herr_t     H5I_register_type(const H5I_class_t *cls);
```

See H5Iregister_type()

```
H5_DLL int64_t    H5I_nmembers(H5I_type_t type);
```

See H5Inmembers()

```
H5_DLL herr_t     H5I_clear_type(H5I_type_t type, hbool_t force, hbool_t app_ref);
```

See H5Iclear_type()

```
H5_DLL H5I_type_t H5I_get_type(hid_t id);
```

See H5Iget_type()

```
H5_DLL herr_t     H5I_iterate(H5I_type_t type, H5I_search_func_t func, void *udata,
hbool_t app_ref);
```

See H5Isearch() and H5Iiterate()

```
H5_DLL int        H5I_get_ref(hid_t id, hbool_t app_ref);
```

See H5Iget_ref()

```
H5_DLL int        H5I_inc_ref(hid_t id, hbool_t app_ref);
```

See H5Iinc_ref()

```
H5_DLL int        H5I_dec_ref(hid_t id);

H5I_dec_ref()
 +-H5I__dec_ref()
    +-H5I__find_id()
    |  +-(id_info->realize_cb)()
    |  +-H5I__remove_common()
    |  +-(id_info->discard_db)()
    +-(type_info->cls->free_func)((void *)info->object, request)
    +-H5I__remove_common()
```

H5I_dec_ref() verifies that the ID is non-negative, and then calls H5I__dec_ref() with H5_REQUEST_NULL as the request parameter.

See H5Idec_ref() for further details.

```
H5_DLL int        H5I_dec_app_ref(hid_t id);
```

See H5Idec_ref()

```
H5_DLL int        H5I_dec_app_ref_async(hid_t id, void **token);

H5I_dec_app_ref_async()
 +-H5I__dec_app_ref()
    +-H5I__dec_ref()
    |  +-H5I__find_id()
    |  |  +-(id_info->realize_cb)()
    |  |  +-H5I__remove_common()
    |  |  +-(id_info->discard_db)()
    |  +-(type_info->cls->free_func)((void *)info->object, request)
    |  +-H5I__remove_common()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

H5I_dec_app_ref_async() verifies that the ID is non-negative, and then calls H5I__dec_app_ref(). This differs from calls to H5I__dec_app_ref() elsewhere in that the request parameter passed into H5I__dec_app_ref() is user supplied, and not hard coded to H5_REQUEST_NULL as in H5Idec_ref() above. This request appears to be passed into the free function from the class when it is called on the void pointer that was passed in on ID registration. It does not appear to be used elsewhere.

Otherwise, the call to H5I__dec_app_ref() seems to be as described in H5Idec_ref() above.

```
H5_DLL int        H5I_dec_app_ref_always_close(hid_t id);

H5I_dec_app_ref_always_close()
 +-H5I__dec_app_ref_always_close()
    +-H5I__dec_app_ref()
    |  +-H5I__dec_ref()
    |  |  +-H5I__find_id()
    |  |  |  +-(id_info->realize_cb)()
    |  |  |  +-H5I__remove_common()
```

```
    |  |  |  +-(id_info->discard_db)()
    |  |  +-(type_info->cls->free_func)((void *)info->object, request)
    |  |  +-H5I__remove_common()
    |  +-H5I__find_id()
    |     +-(id_info->realize_cb)()
    |     +-H5I__remove_common()
    |     +-(id_info->discard_db)()
    +-H5I_remove()
       +-H5I__remove_common()
```

H5I_dec_app_ref_always_close() verifies that the supplied ID is non-negative, and then calls H5I__dec_app_ref_always_close() with the supplied ID and the request parameter set to H5_REQUEST_NULL. It returns whatever value H5I__dec_app_ref_always_close() returns.

After initial sanity checks, H5I__dec_app_ref_always_close() calls H5I__dec_app_ref() with the supplied id and request (H5_REQUEST_NULL in this case). See H5Idec_ref() for a discussion of H5I__dec_app_ref() under these circumstances.

When H5I__dec_app_ref(), H5I__dec_app_ref_always_close() checks for failure, and calls H5I_remove() if a failure is detected. This appears to be an attempt to force removal of the ID even if the free call fails -- see the following comment:

```
    /*
     * If an object is closing, we can remove the ID even though the free
     * method might fail.  This can happen when a mandatory filter fails to
     * write when a dataset is closed and the chunk cache is flushed to the
     * file.  We have to close the dataset anyway. (SLU - 2010/9/7)
     */
```

H5I_remove() is discussed in H5Iremove_verify() above.

```
H5_DLL int        H5I_dec_app_ref_always_close_async(hid_t id, void **token);
```

```
H5I_dec_app_ref_always_close_async()
 +-H5I__dec_app_ref_always_close()
    +-H5I__dec_app_ref()
    |  +-H5I__dec_ref()
    |  |  +-H5I__find_id()
    |  |  |  +-(id_info->realize_cb)()
    |  |  |  +-H5I__remove_common()
    |  |  |  +-(id_info->discard_db)()
    |  |  +-(type_info->cls->free_func)((void *)info->object, request)
    |  |  +-H5I__remove_common()
    |  +-H5I__find_id()
    |     +-(id_info->realize_cb)()
    |     +-H5I__remove_common()
    |     +-(id_info->discard_db)()
    +-H5I_remove()
       +-H5I__remove_common()
```

As per H5I_dec_app_ref_always_close() above, save that H5I_dec_app_ref_always_close_async() takes a token parameter (void **), that is passed down

to +-H5I__dec_app_ref_always_close() as its request parameter. This parameter is eventually passed to the free function for the type if the ref count drops to zero.

```
H5_DLL int        H5I_dec_type_ref(H5I_type_t type);
```

See H5Idec_type_ref()

```
H5_DLL herr_t     H5I_find_id(const void *object, H5I_type_t type, hid_t *id /*out*/);
```

```
/* User data for iterator callback for retrieving an ID corresponding to an object
pointer */
typedef struct {
    const void *object;   /* object pointer to search for */
    H5I_type_t  obj_type; /* type of object we are searching for */
    hid_t       ret_id;   /* ID returned */
} H5I_get_id_ud_t;
```

```
H5I_find_id()
 +-H5I__find_id_cb()
    +-H5I__unwrap()
       +-H5VL_object_data()
       |  +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |     +- ??? -- must investigate
       +-H5T_get_actual_type()
          +-H5VL_object_data()
             +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
                +- ??? -- must investigate
```

In a nutshell:

Scan all IDs in the target type. If an ID has a void pointer associated with it that matches the supplied void *, return this ID in *id.

In greater detail:

After initial sanity checks, and setting *id = H5I_INVALID_HID, H5I_find_id() looks up the instance of H5I_type_info_t associated with the supplied type in the H5I_type_info_array_g global array. If the target type exists and has been initialized, and has at least one entry. the function initializes an instance of H5I_get_id_ud_t as follows:

```
/* Set up iterator user data */
udata.object   = object;
udata.obj_type = type;
udata.ret_id   = H5I_INVALID_HID;
```

and then uses the HASH_ITER uthash macro to set up a for loop that visits each ID in the target type.

It calls H5I__find_id_cb() on each such entry, returning an error if H5I__find_id_cb() returns and error, and breaking out of the for loop if H5I__find_id_cb() returns H5_ITER_STOP. In either case, H5I_find_id() sets *id = udata.ret_id after it exits the for loop.

H5I__find_id_cb() calls H5I__unwrap() on the void pointer associated with the target id. It tests to see if the return value of H5I__unwrap() equals udata->object. If it does, it sets udata->ret_id = info->id, and returns H5_ITER_STOP.

See H5Isearch() above for a discussion of H5I__unwrap(). The bottom line is that it makes calls into H5VL, and then into a VOL connector callback -- with the resulting potential multithread safety issues. I am putting this issue to one side pending review of H5VL.

```
/* NOTE:    The object and ID functions below deal in non-VOL objects (i.e.;
 *          H5S_t, etc.). Similar VOL calls exist in H5VLprivate.h. Use
 *          the H5VL calls with objects that go through the VOL, such as
 *          datasets and groups, and the H5I calls with objects
 *          that do not, such as property lists and dataspaces. Datatypes
 *          are can be either named, where they will use the VOL, or not,
 *          and thus require special treatment. See the datatype docs for
 *          how to handle this.
 */

/* Functions that manipulate objects */
H5_DLL void * H5I_object(hid_t id);
```

See H5Iget_type()

```
H5_DLL void * H5I_object_verify(hid_t id, H5I_type_t type);
```

See H5Iobject_verify()

```
H5_DLL void * H5I_remove(hid_t id);
```

See H5Iremove_verify()

```
H5_DLL void * H5I_subst(hid_t id, const void *new_object);

H5I_subst()
  +-H5I__find_id()
     +-(id_info->realize_cb)()
     +-H5I__remove_common()
     +-(id_info->discard_db)()
```

In a nut shell:

Replace the void * associated with the ID with the supplied void *, returning the old void *.

In greater detail:

H5I_subst() calls H5I__find_id() to obtain the instance of H5I_id_info_t associated with the target ID. See H5Iobject_verify() above for a discussion of H5I__find_id().

Assuming that H5I__find_id() is successful, H5I_subst() sets info->object = new_object, and returns the original value of info->object.

```
H5_DLL htri_t H5I_is_file_object(hid_t id);

H5I_is_file_object()
 +-H5I_get_type()
 +-H5I_object()
 |  +-H5I__find_id()
 |     +-(id_info->realize_cb)()
 |     +-H5I__remove_common()
 |     +-(id_info->discard_db)()
 +-H5T_is_named()
```

H5I_is_file_object() calls H5I_get_type() to obtain the type of the supplied ID. If the type is either H5I_DATASET, H5I_GROUP, or H5I_MAP, the function return TRUE.

If the type is H5I_DATATYPE, it calls H5I_object() to obtain the instance of H5T_t associated with the id, calls H5T_is_named() on this instance, and returns whatever H5T_is_named() returns.

Otherwise, the function returns FALSE.

H5I_get_type() invokes the H5I_TYPE() macro to extract the type, and returns this value.

See H5Iobject_verify() for a discussion of H5I_object().

H5T_is_named() returns TRUE iff the datatype is named/committed. This is determined by examining fields of the supplied instance of H5T_t, so no special multi-thread issues beyond the usual race conditions.

```
/* ID registration functions */
H5_DLL hid_t  H5I_register(H5I_type_t type, const void *object, hbool_t app_ref);

H5I_register()
 +-H5I__register()
```

After some sanity checks H5I_register() calls H5I__register() with its parameters, and NULL for the realize_cb and discard_cp parameters.

See H5Iregister() above for a discussion of H5I__register()

```
H5_DLL herr_t H5I_register_using_existing_id(H5I_type_t type, void *object, hbool_t
app_ref,
                                             hid_t existing_id);

H5I_register_using_existing_id()
```

```
 +-H5I__find_id()
    +-(id_info->realize_cb)()
    +-H5I__remove_common()
    +-(id_info->discard_db)()
```

In a nutshell:

Register the supplied void * in the specified index with the specified ID. The function will fail if the ID is already in use, or if it doesn't belong to the specified index.

In greater detail:

H5I_register_using_existing_id() first verifies that the supplied id is not in use via a call to H5I__find_id(). (see H5Iobject_verify() for a discussion of H5I__find_id()). It then verifies that the supplied type is valid, and that the supplied id belongs to the supplied type. As part of these sanity checks, it looks up the instance of H5I_type_info_t associated with the supplied type and stores its address in type_info. In so doing, it reads the global H5I_next_type_g and the global H5I_type_info_array_g array.

If all these sanity checks pass, the function allocates an instance of H5I_id_info_t via H5FL_CALLOC() storing its address in info. (note the thread safety issue), initializes it, and then uses the uthash HASH_ADD() to insert it into the hash table of the specified index with the specified id. Before exiting, it sets type_info->last_id_info = info.

From a multi-thread safety perspective, H5I_register_using_existing_id() seems to have the same issues as H5I__register().

```
/* Debugging functions */
H5_DLL herr_t H5I_dump_ids_for_type(H5I_type_t type);
```

This is a debugging function, and thus can be skipped for now.

Appendix 3 – Fields modified by API Calls

The following table lists the fields in the relevant instances of H5I_type_info_t and H5I_id_info_t that are modified by the listed API calls. Note that this data is derived from inspection of the code, and thus some errors should be expected.

Note also that H5Iget_file_id() and H5Iget_name() have been omitted from this table.

Operation(s)	Field modified in H5I_type_info_t	Fields modified in H5I_id_info_t
H5Iregister() H5Iregister_future() H5I_register()	nextid, id_count, last_id_info hash_table	all – allocate instance and initialize
H5Iobject_verify() H5I_object_verify() H5I_object() (no future objects)	last_id_info	none
H5Iobject_verify() H5I_object_verify() H5I_object() (future objects possible)	last_id_info, hash_table, id_count	all (delete one index entry, modify another)
H5Iremove_verify() H5I_remove()	hash_table, last_id_info, id_count	all (free instance) or marked (if H5I_marking_g is TRUE)
H5Iget_type() H5I_get_type() (no future objects)	none	none
H5Iget_type() H5I_get_type() (future objects possible)	hash_table, id_count	all (delete one index entry, modify another)
H5Iinc_ref() H5I_inc_ref() (no future objects)	none	count, app_count
H5Iinc_ref() H5I_inc_ref() (future objects possible)	hash_table, id_count	all (delete one index entry, modify another)
H5Idec_ref() H5I_dec_ref() H5I_dec_app_ref_async() H5I_dec_app_ref_always_close() H5I_dec_app_ref_always_close_async()	hash_table, last_id_info, id_count	all

H5Iget_ref() H5I_get_ref() (no future objects)	none	none
H5Iget_ref() H5I_get_ref() (future objects possible)	hash_table, id_count	all (delete and free one index entry, modify another)
H5Iregister_type	all – allocate and initialize	none
H5Iclear_type() H5I_clear_type()	hash_table, id_count	all – delete and free most entries
H5Idestroy_type()	all – de-allocate when done	All – delete and free all entries
H5Iinc_type_ref()	init_count	none
H5Idec_type_ref() H5I_dec_type_ref()	just init_count if it remains positive, or all with de-allocation when done if init_count drops to zero	none, if init_count remains positive. otherwise all – delete and free all entries
H5Iget_type_ref()	none	none
H5Isearch() H5Iiterate() H5I_iterate() H5I_find_id()	potentially hash_table, id_count, last_id_info	potentially all
H5Inmembers() H5I_nmembers()	id_count	none
H5Itype_exists()	none	none
H5Iis_valid() (no future objects)	none	none
H5Iis_valid() (future objects possible)	hash_table, id_count	all (delete one index entry, modify another)
H5I_subst() (no future objects)	last_id_info	object
H5I_subst() (future objects possible)	last_id_info, hash_table, id_count	all (delete one index entry, modify another)
H5I_is_file_object() (no future objects)	none	none
H5I_is_file_object() (future objects possible)	hash_table, id_count	all (delete one index entry,

		modify another)
H5I_register_using_existing_id() (no future objects)	id_count, last_id_info hash_table	all – allocate instance and initialize
H5I_register_using_existing_id() (future objects possible)	id_count, last_id_info hash_table	all – either allocate an instance and initialize, or delete one index entry, modify another