Making H5E Multi-Thread Safe:
A Sketch Design

John Mainzer
Lifeboat, LLC

9/25/22

Introduction

The package-by-package strategy proposed for retrofitting multi-thread support onto HDF5 requires us to:

1. Identify leaf packages – i.e., packages that do not, or at least need not, call other packages, and make these packages multi-thread safe. Repeat until all such packages have been made multi-thread safe.

2. Identify packages that only call packages that have been made multi-thread safe, and make these packages multi-thread safe. Repeat until all such packages have been made multi-thread safe.

3. Any remaining packages must participate in cycles of calls that involve other non-multi-thread safe packages. Address these cases by either re-architecting the package to avoid these cycles (thus pushing them into case 2 above), or by addressing multi-thread safety in the entire cycle simultaneously. Repeat until all packages in the HDF5 library are multi-thread safe.

The above process, if followed to its conclusion, will result in all packages in HDF5 being made multi-thread safe. Note, however, that this in itself may not be sufficient to make HDF5 multi-thread safe as a whole, as there may be lock ordering issues to be dealt with between the packages. Resolving these issues may require further re-architecting of the HDF5 library. Obviously, this potential issue can be minimized by avoiding locks to the extent possible.

It should be obvious that this strategy will result in a multi-thread safe version of the HDF5 library if sufficient resources are devoted to it. Whether this is practical or not is another question.

Fortunately, we do not need to complete the entire process to obtain useful results. As discussed in the Multi-Thread RFC, if we can make the H5E, H5I, H5P, H5CX, and H5VL packages multi-thread safe, we can move the global lock down to the native VOL and allow multi-thread

processing in VOL Connectors. (Note that H5S must be made multi-thread safe as well to do this fully, but we should be able to obtain useful results without it.)

Thus, the objective of the current exercise is to retrofit multi-thread safety on the above listed packages, both to provide support for multi-thread processing in VOLs, and to develop the experience required to determine whether the full strategy described above is practical.

Returning to the topic of this sketch design, the H5E package fits in the first category above, and thus is a reasonable starting point. This sketch design outlines the current state of H5E with reference to multi-threading, and offers an approach to retro-fitting multi-thread safety on H5E. It is circulated both for suggestion for improvement, and in the hope that reviewers will point out any misconceptions, errors, or omissions.

A Quick Overview of H5E

H5E exists to facilitate error reporting. In the typical case, when an error is detected in a HDF5 function,

- the name of the host function,
- the name of the source code file in which the function is defined,
- the source code line number on which the error was detected,
- IDs indicating the class of the error and its major and minor error numbers[1], and
- an error message.

are pushed on an error stack[2] (more exactly an instance of H5E_t), and the function returns an error. This error is detected by the calling function, which repeats the process until the error propagates up the call stack to the public HDF5 API call that occasioned the error, where the error stack is dumped (unless it is suppressed) and an error code is returned to the program that called the HDF5 library.

The net effect is to report not only where the error was detected and some indication of the nature of the error, but also at least the top of the call stack at the time the error was detected.

In the typical case, H5E is only used to report errors in the HDF5 library. However, it has API calls that allow applications to define their own error classes, major, and minor errors, and use

---

1 The class of the error is the body of code in which it occurred – in almost all cases the HDF5 library. Major and minor error codes indicate the general and more specific types of failure.

2 Usually via the HGOTO_ERROR() macro, which calls H5E_printf_stack(). That function constructs the error message, and then calls H5E__push_stack() to insert the error data into the error stack. See the end of Appendix 3 for a manual expansion of this macro.

H5E facilities to report them when appropriate. To my knowledge, VOL developers are the main users of this capability.

Multi-Thread Issues in H5E

While there appear to be no fundamental reasons why H5E can't be made multi-thread safe, the current design presents two main challenges – use of other packages in HDF5 and elements of the public and private APIs which allow multiple threads to interact with its internal data structures concurrently. 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 in H5E

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

- H5MM
- H5FL, and
- H5I

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.

In contrast, the dependency on H5I presents more difficult issues – particularly since it involves public APIs.

The first, and most obvious of these is the fact that H5I is not multi-thread safe at this point – thus our strategy requires us to duplicate the required functionality in H5E and remove the H5I calls – for now at least.[3]

In addition, storing H5E data structures in the indexes maintained by H5I makes these H5E data structures simultaneously accessible to multiple threads – albeit only through HDF5 code. This isn't as bad as it sounds, as two of the data structures exposed are effectively constant tables. However, under some circumstances, error stacks are exposed as well. As shall be seen, this issue is complicated by the H5E public API, and the fact that once inserted in H5I's indexes, H5E data structures become at least indirectly accessible via H5I calls.

H5E defines three types of IDs in H5I:

---

3 Once H5I is made multi-thread safe, this decision should be re-visited. If the multi-thread safe version of H5I is lock-less, returning to the previous state of affairs may be a viable option.

- H5I_ERROR_CLASS
- H5I_ERROR_MSG
- H5I_ERROR_STACK

(See the declaration of the H5I_type_t enumerated type in H5Ipublic.h).

H5I entries of H5I_ERROR_CLASS type are used to store instances of H5E_cls_t (defined in H5Epkg.h).

```
typedef struct H5E_cls_t {
  char *cls_name; /* Name of error class */
  char *lib_name; /* Name of library within class */
  char *lib_vers; /* Version of library */
} H5E_cls_t;
```

From review of the code, it appears that these instances are used to associate error messages with the body of code they appear in. Thus, H5E proper registers only one such instance. As shall be discussed later, there is a public API call that allow the user to create and register / un-register others – albeit indirectly.

Similarly, H5I entries of H5I_ERROR_MSG type are used to store instances of H5E_msg_t (defined in H5Epkg.h).

```
/* Major or minor message */
typedef struct H5E_msg_t {
  char *     msg;  /* Message for error */
  H5E_type_t type; /* Type of error (major or minor) */
  H5E_cls_t *cls;  /* Which error class this message
                      belongs to */
} H5E_msg_t;
```

which are in turn used to store major and minor errors with their associated strings. The HDF5 library creates a much larger number of these, but they are all known at compile time – indeed code to create them is generated automatically during the build process from H5err.txt, placed in H5Einit.h, and included and then executed in H5E_init(). The associated ID's are used in the error stack and are referenced when printing same. Note that the hid associated with the error class is not used to refer to it in H5E_msg_t – instead the structure contains a pointer to the associated instance of H5E_cls_t.

As with H5E_cls_t, there are H5E API calls to create and register/un-register instances of H5E_msg_t – albeit indirectly.

Finally, H5I entries of H5I_ERROR_STACK are used to store error stacks (instances of H5E_t)

```
/* Error stack */
struct H5E_t {
  size_t       nused;          /* Num slots currently
                               used in stack  */
  H5E_error2_t  slot[H5E_NSLOTS]; /* Array of error
                               records       */
  H5E_auto_op_t auto_op;        /* Operator for
                               'automatic' error
                               reporting */
  void *       auto_data;       /* Callback data for
                               'automatic error
                               reporting */
};

/**
* Information about an error; element of error stack
*/
typedef struct H5E_error2_t {
   hid_t cls_id;         /**< Class ID           */
   hid_t maj_num;        /**< Major error ID      */
   hid_t min_num;        /**< Minor error number   */
   unsigned line;        /**< Line in file where
                         error occurs        */
   const char *func_name; /**< Function in which
                         error occurred       */
   const char *file_name; /**< File in which error
                         occurred           */
   const char *desc;     /**< Optional supplied
                         description         */
} H5E_error2_t;

/* Some syntactic sugar to make the compiler happy with two different kinds of
callbacks */
#ifndef H5_NO_DEPRECATED_SYMBOLS
typedef struct {
   unsigned    vers;         /* Which version callback to
                            use */
   hbool_t     is_default;    /* If the printing function
                            is the library's own. */
   H5E_auto1_t func1;        /* Old-style callback, NO
                            error stack param. */
   H5E_auto2_t func2;        /* New-style callback, with
                            error stack param. */
   H5E_auto1_t func1_default; /* The saved library's
```

```
                        default function – old
                        style. */
    H5E_auto2_t func2_default; /* The saved library's
                        default function – new
                        style. */
} H5E_auto_op_t;
#else  /* H5_NO_DEPRECATED_SYMBOLS */
typedef struct {
    H5E_auto2_t func2;        /* Only the new style
                        callback function is
                        available. */
} H5E_auto_op_t;
#endif /* H5_NO_DEPRECATED_SYMBOLS */
```

in the index. In the typical case, error stacks are created on a thread specific basis, and are not registered in the index. This simplifies the problem of retrofitting multi-thread safety on H5I greatly, since in this case, each error stack is only visible to a single thread – removing any need to consider multi-thread safety for the error stacks proper. However, as mentioned above, there are public APIs that create and register/un-register instances of H5E_t.

While the previously discussed H5I index entries are practically speaking constants, error stacks are typically modified and eventually discarded. Since inserting them into the index allows multiple threads to act on them simultaneously, we need some method to prevent subsequent corruption.

With this background in place, we proceed to a discussion of the H5E public and private APIs with reference to multi-thread safety.

Multi-thread thread issues in the H5E public and private APIs

Appendicles 1 and 2 contains a list of all H5E public and private APIs, along with call trees and brief discussions of their function and potential multi-thread issues. While the reader is invited to review these appendicles for background data, in this section it should be sufficient to list the multi-thread safety issues not specifically related to H5I that must be addressed.

Multi-thread issues with the H5E public and private APIs divide into two categories:

- Potential race conditions between API calls.

- Potential data structure corruption resulting from concurrent execution of public API calls

These issues are discussed in the following sections:

API Race Conditions

The first issue is endemic to the API – for example, at present, there is nothing to prevent one thread from deleting an error class and all of its error messages just before another thread flags one of these errors or tries to print an error stack that references some of these errors. Unfortunately, this is unsolvable without semantic changes to the API. Even with such changes, it would still be possible for one thread to delete an error stack just before another one tried to print of modify it.

Absent API changes, all that H5E can do is handle dangling IDs gracefully, and otherwise keep the H5E data structures in a consistent state and, to the extent possible, appear to execute operations in some order,

Beyond this, it will be the responsibility of the client to either avoid race conditions of the above type, or to handle them gracefully.

Potential H5E Data Structure Corruption

Since the lists of error messages and error classes are essentially constant, with the only operations being creation and deletion, the only real issue here is error stacks, which are subject to a variety of operations.

Error stacks are typically thread specific – as long as this is the case it is hard to see any multi-thread safety issues beyond the above-mentioned possibility of error classes and messages being deleted out from under an error stack.[4]

However, there are API calls to create and register/un-register error stacks – with error stacks being deleted once their reference counts drop to zero. Further, there are calls to push and pop error messages, walk an error stack executing an arbitrary function call on each entry, append one error stack to another, and to get and set configuration data. All of these activities present possibilities for data structure corruption.

A further issue is the function supplied to the walk routine. The HDF5 library has no control on the activities of this function – with the obvious potential for deadlocks.

That said, we must keep in mind that none of these actions are an issue as long as they are executed sequentially – which is currently enforced by the HDF5 library global lock.

4 Entries in error stacks are maintained in an array of instance of H5E_error2_t (defined in H5Epubic.h). These in turn contain the IDs of the relevant error classes, and major and minor errors.

Solutions and their Discontents

As discussed above, our strategy for retrofitting multi-thread safety on H5E requires that we remove all calls to other HDF5 packages. H5MM and H5FL are easy enough, but H5I requires that we duplicate the necessary facilities in H5E – and make them multi-thread safe to boot.

Before outlining some possible approaches, a few comments are in order.

First, from a long-term perspective, it would be best to remove all external dependencies, and either make the thread safe H5E implementation lock free, or at worst, use locks only for small critical regions without function calls, as this would simplify maintenance and make it harder to accidentally insert deadlocks at a later date.

However, we need a working prototype soon, so we can demonstrate performance advantages.

Second, while H5E is relatively isolated, and the first cut at the sketch design for H5I is complete, reviews of H5P, H5CX, and H5VL may bring surprises that require re-thinking of the H5E sketch.

Thus, the following sketches should be regarded as straw men – they have changed greatly since the first version of this document, and they will almost certainly change again as my understanding of the issues and available resources becomes clearer.

Error classes and messages

At least in HDF5 proper, error classes and error messages are known at compile time – and thus can be viewed conceptually as entries in a constant table. While one could contrive such a case, it is hard to come up with a plausible case where this would not be true for client programs that use H5E.

Further, in addressing this issue, we should keep in mind that our solution to multi-thread safe error class and error message registration / un-registration and access may well prove temporary – if the multi-thread modifications to H5I are lock free, we may wish to return to H5I for managing the error classes and messages.

Putting all this together, we need a simple, easy to implement, lock-less solution that is easy to back out of. If possible, it should be workable in the long term. The following is an attempt to meet these requirements.

The key insight here is that if the error classes and messages were stored in a constant table constructed at compile time, there would be no multi-thread issues related to them to address. In particular, such tables would exist throughout the computation, and would not care how many threads read them at the same time. While the H5E API calls that create and delete error classes and messages at run time make this impossible absent API changes, we can come close.

To simplify this exposition, first consider only the pure HDF5 case. Here the code to construct the error messages is generated automatically at build time, and the resulting file #included in H5E_init(). Thus we know how many instances H5E_msg_t are required. Further we can calculate the total length of the required strings. Similarly, we can calculate the space required for the singleton instance of H5E_cls_t and its strings.

Given this data, declare static arrays of H5E_cls_t, H5E_msg_t, and char of sufficient size to represent all the expected error class and error message creations, and their associated strings. Modify H5Eregister_class() and H5Ecreate_msg() to copy data into the static arrays, returning an hid_t whose low order bits are indexes into the static arrays.

If we assume that all calls to H5Eregister_class() and H5Ecreate_msg() will be made in a single thread, that all these will complete before any other H5E calls, and that all threads will complete before H5E is shut down, we are done,[5] as the static tables allow us to ignore reference counts, etc.

However, this is not the case, so we have to extend the above solution.

If multiple threads may be involved in registering entries in the statically allocate tables, we need some mechanism to ensure that each entry in these tables is written at most once. Do this with an atomic integer for each table and use atomic fetch and modify call to allocate unique indexes into the target table.

Similarly, we need a mechanism for preventing access to entries before they are fully initialized. For this, create an atomic integer for each entry in each of the statically allocated tables. Initialize these integers to special value (say negative max int) to indicate that the entry is undefined. After the entry is initialized and ready for use, set the associated atomic integer to one and use it as a reference count on the associated entry. Since the tables are effectively constant, the reference counts are largely irrelevant. However, to maintain existing behavior, treat the associated entry as deleted if its reference count drops to zero.

Finally, we must extend this sketch to support definition of error classes and messages by client software. Do this by doubling or tripling the size of the static tables, and throwing an error it the available capacity is exceeded. Applications that exceed this limit will have to increase the size of the relevant tables and re-compile the HDF5 library. This is a bother, but doesn't seem excessive given the limited use of the facility.

---

5 Here I ignore the problem of integrating this new implementation of the error class and error message index into H5E. While I will address this later, the basic idea is to keep the existing internal APIs as unchanged as possible by adding code to re-direct H5I calls addressing IDs of H5I_ERROR_CLASS and H5I_ERROR_MSG type to new H5E calls to perform the necessary operations on the statically allocated tables.

Alternatively, the above solution could be extended to use dynamically allocated memory and support an arbitrary number of error messages and classes – albeit at the cost of significantly increased complexity.

Error Stacks

Error stacks are a more difficult problem than error classes and error messages, as they are dynamic data structures that are created, modified, read, and discarded at run time. Thus, in addition to the indexing problem, there is the matter of maintaining mutual exclusion on the error stacks so as to avoid data structure corruption.

The notion of solving this problem by passing it up to the application is tempting – indeed the HDF5 library does similar things in its MPI build. Unfortunately, the consequences of an applications failure to keep its end of the bargain include potential heap corruption.

Thus, we are pushed towards some mechanism for enforcing mutual exclusion on error stacks. While a lock free solution would be preferable, modifications to error stacks are frequently both numerous and widely distributed – making this approach problematic without significant re-architecting.

This leaves us leaves us with the option of associating a mutex with each error stack that is visible to all threads – that is, all error stacks that are not thread local.

My impression is that this is a sufficiently improbable case that performance is not an issue. However, there are a number of correctness issues that must be addressed:

Lock ordering in the H5Eappend_stack() API call

H5Eappend_stack() operates on two error stacks that have been registered, and are thus visible to all threads – which presents a potential lock ordering issue. Solve this by locking the source and destination stacks in increasing ID order. Use the same ordering in any similar cases.

Lock Ordering Issues with the Error Class and Error Message Indexes

Many API calls manipulating error stacks perform lookups on error message and error class IDs. At present, these indexes are maintained via H5I. While it remains to be seen how multi-thread safety will be implemented in H5I, current thinking for the first cut involves placing a recursive R/W lock on each index. Further, it is possible that a lookup will fail, bringing the thread of execution back into H5E.

At present, I don't see how the thread of execution could lead back to an error stack that is registered and thus requires a mutex, as all error calls in the HDF5 library proper appear to involve only the thread local error stack – and even if this is not the case, we could make the mutex recursive.

However, the point here is that there is a great deal of complexity and thus opportunity for error, making it very easy to insert a deadlock later even if the initial implementation avoids them. Thus, I view this as a strong argument for implementing some sort of a lock free index for the error classes and error messages as described above. If we do this, the lock ordering issue with H5I vanishes.

This does not mean that we are home free – it is still possible that we will detect an error while holding a lock on an error stack. In this case we must either drop the lock before flagging the error, or ensure that the error stack on which the error will be recorded is thread local and thus not subject to locking.

Indexing Error Stacks

We must also deal with indexing error stacks. As argued above, it would be best to avoid H5I for this purpose, so as to remove the dependency.

This leaves us with the problem of designing a multi-thread safe index for error stacks, with insert, delete, and lookup operations. To maintain the current API, we would also need a reference count.

While a lock free data structure would be preferable for this purpose, we are already using uthash in H5I. Further, uthash claims to be multi-thread safe if the uthash macros are suitably protected with a R/W lock. While there are issues with uthash error reporting as detailed in the H5I sketch, it would likely be sufficient for at least an initial implementation, and perhaps indefinitely.

Public API Changes

While we want to think long and hard before we make public API changes, the review of H5I has provided some strong incentives to at least consider the idea. In particular, H5I's iterate calls are very inconvenient from a multi-thread perspective – raising the thought of disabling them in the multi-thread case, and providing more multi-thread friendly alternatives that would function in both the multi-thread and single thread regimes.

While the arguments are not so strong in the H5E case, turning API calls that delete error classes and messages into NO-OPs in the multi-thread case may simplify matters considerably. Similarly, replacing the H5Ewalk1() and H5Ewalk2() calls with something more multi-thread friendly may allow us to sidestep the dangers of the unconstrained function that is executed on each entry in the target error stack.

While I don't propose any immediate action on this, it is something to be thinking about as I continue my scan through the target packages.

Appendix 1 – H5E public API calls

This appendix contain a list of the H5E public API calls, along with call trees[6] and brief discussions describing their behavior with particular reference to multi-thread safety related issues. The list of public calls is taken from H5Ipublic.h, and the call trees and descriptions were obtained via inspection of the code – and thus likely contain errors and omissions. Corrections are welcome. Issues seeming unrelated to multi-thread safety are covered lightly, if at all.

```
hid_t H5Eregister_class(const char *cls_name,
                        const char *lib_name,
                        const char *version);
```

```
H5Eregister_class()
+-H5E__register_class()
|  +-H5FL_CALLOC()
|  +-H5MM_xctrdup()
|  +-H5E__free_class()
|    +-H5MM_xfree()
+-H5I_register()
  +- …
```

In a nutshell:

Create instance of H5E_cls_t and register it (with H5I). Return an ID allowing it subsequent lookup.

In greater detail:

Allocate new instance of H5E_cls_t, initialize it with the supplied data, register it (via H5I_register()) with H5I as an index entry of H5I_ERROR_CLASS, and return the new hid_t.

Multi-thread concerns:

Once it is registered, the new instance of H5E_cls_t is potentially accessible to all threads.

To this, add any multi-thread issues inherited from H5I_register().

***

6 Sections of call trees involving calls to H5I are omitted unless they involve callbacks into H5E. Please see the H5I multi-thread sketch design for the elided details of H5I calls.

```
herr_t H5Eunregister_class(hid_t class_id);

H5Eunregister_class()
 +-H5I_get_type()
 |  +- …
 +-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)
     |  |  ||
     |  | H5E__unregister_class() // in this case
     |  |  +-H5I_iterate()
     |  |  | +-H5I__iterate_cb()
     |  |  |   +-H5I__unwrap() // a NO-OP in this case
     |  |  |   |  +- …
     |  |  |   +-op()
     |  |  |     ||
     |  |  |    H5E__close_msg_cb() // in this case
     |  |  |     +-H5E__close_msg()
     |  |  |     |  +-H5MM_xfree()
     |  |  |     |  +-H5FL_FREE()
     |  |  |     +-H5I_remove()
     |  |  |       +-H5I__remove_common()
     |  |  +-H5E__free_class()
     |  +-H5I__remove_common()
     +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nutshell:

Decrement the reference count on the target error class. If this reference count drops to zero, the error class and all associated error messages are deleted.

In greater detail:

H5Eunregister_class() first calls H5I_get_type() to verify that the supplied ID is that of an error class. It then calls H5I_dec_app_ref() which calls H5I__dec_app_ref(),

H5I__dec_app_ref() calls H5I__dec_ref(), which decrements the regular reference count on the indicated instance of H5E_cls_t in the index. If the regular reference count drops to zero, H5E__unregister_class() is called.

H5E__unregister_class iterates over all entries of type H5I_ERROR_MSG in the index, calling H5E__close_msg_cb() on each. H5E__close_msg_cb tests to see if the target entry is associated with the target error class – if it is, it:

- calls H5E__close_msg() to free the string containing the message string associated with the instance of H5E_msg_t,
- calls H5I_remove() to remove it from the index, and finally
- frees the target instance of H5E_msg_t.

Finally, H5E__unregister_class() calls H5E__free_class(), which deletes the target instance of H5E_cls_t.

If the target instance of H5E_cls_t still exists after the call to H5I__dec_ref(), H5I__dec_app_ref() decrements the application reference count.

Multi-thread concerns:

H5Eunregister_class() may delete data structures that are visible to other threads – specifically the target instance of H5E_cls_t and any associated instances of H5E_msg_t.

To this, add any multi-thread issues associated with the H5I calls.

***

```
herr_t H5Eclose_msg(hid_t err_id);

H5Eclose_msg()
 +-H5I_get_type()
 |  +- …
 +-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)
|           // H5E__close_msg() in this case
|  +-H5I__remove_common()
+-H5I__find_id()
   +-(id_info->realize_cb)()
   +-H5I__remove_common()
   +-(id_info->discard_db)()
```

In a nutshell:

Decrement the reference count on the target error message. If the reference counts drops to zero, discard the message.

In greater detail:

Calls H5I_dec_app_ref() on the target error message.

H5I__dec_app_ref() calls H5I__dec_ref(), which decrements the regular reference count on the indicated instance of H5E_msg_t in the index. If the regular reference count drops to zero, H5E__close_msg() is called.

H5E__close_msg() frees the string containing the message string associated with the instance of H5E_msg_t, and then frees the target instance of H5E_msg_t proper,

After H5E__close_msg() returns, the associated entry is removed from the index.

Multi-thread concerns:

H5Eclose_msg() may delete a data structures that is visible to other threads – specifically the target instance of H5E_msg_t.

To this, add any multi-thread issues associated with the H5I calls.

***

```
hid_t H5Ecreate_msg(hid_t cls, H5E_type_t msg_type,
```

```
                const char *msg);

H5Ecreate_msg()
 +-H5I_object_verify()
 |  +- …
 +-H5E_create_msg()
 |  +-H5FL_MALLOC()
 |  +-M5MM_xstrdup()
 |  +-H5E__close_msg() // error case
 +-H5I_register()
   +- …
```

In a nutshell:

Create an error message of the specified type and class, insert it into the appropriate index, and return the associated ID.

In greater detail:

Call H5I_object_verify to obtain a pointer to the instance of H5E_cls_t associated with the supplied error class.

Call H5E_create_msg() to allocate and initialize an instance of H5E_msg_t,

Call H5I_register() to insert the new instance of H5E_msg_t into the H5I_ERROR_MSG index, and return the ID associated with the new error message to the caller.

Note that the supplied class_id is verified, and a pointer to the associated instance of H5E_cls_t is included in the H5E_msg_t.

Multi-thread concerns:

Once it is registered, the new instance of H5E_msg_t is potentially accessible to all threads.

To this add any multi-thread issues inherited from H5I_register().

***

```
hid_t H5Ecreate_stack(void);

H5Ecreate_stack()
```

```
+-H5FL_CALLOC()
+-H5E__set_default_auto()
+-H5I_register()
  +- …
```

In a nutshell:

Create and register an error stack and return its ID.

In greater detail:

Allocate and initialize an error stack (i.e. and instance of H5E_t) and register it in the index via a call to H5I_register(). Return the hid_t assigned to the new instance of H5E_t.

Multi-thread concerns:

Once it is registered, the new instance of H5E_t is potentially accessible to all threads.

To this add any multi-thread issues inherited from H5I_register().

***

```
hid_t H5Eget_current_stack(void);

H5Eget_current_stack()
 +-H5E__get_current_stack()
 |  +-H5E__get_my_stack()
 |  +-H5FL_CALLOC()
 |  +-H5I_inc_ref()
 |  |  +- …
 |  +-H5MM_xstrdup()
 |  +-H5E_clear_stack()
 |     +-H5E__get_my_stack()
 |     +-H5E__clear_entries()
 |     |  +-H5I_dec_ref()
 |     |     +- …
 |     +-H5MM_xfree_const()
 +-H5I_register()
   +- …
```

In a nutshell:

Create a new error stack (H5E_t), copy the current (thread specific error stack into the new error stack, and clear the current error stack. Register the new error stack, and return its ID for future lookup.

In greater detail:

H5Eget_current_stack() first calls H5E__get_current_stack() to obtain the new stack, and then calls H5I_register to insert the new stack into the error stack index (type = H5I_ERROR_STACK). Absent errors, it returns the ID returned by H5I_register().

H5E__get_current_stack() first calls H5E__get_my_stack to obtain a pointer to the thread specific error stack, and then allocates a new error stack (instance of H5E_t) via H5FL_CALLOC().

The function then walks the current and new stacks in tandem, copying entries from the current stack to the equivalent entry on the new stack. In passing, the function calls H5I_inc_ref() on the IDs in the cls_id, maj_num, and min_num fields. It uses H5MM_xstrdup() to copy the strings pointed to by the desc field.

Finally, it calls H5E_clear_stack() to clear the current stack, and then returns a pointer to the new stack.

H5E__clear_stack() calls H5E__get_my_stack to obtain a pointer to the thread specific error stack, and then calls H5E__clear_entries() to clear all entries in the thread specific stack.

H5E_clear_entries() walks the target error stack from the top down to clear the specified number of entries – all of them in this case. For each entry, it calls H5I_dec_ref() on the IDs in the cls_id, maj_num, and min_num fields, and H5MM_xfree_const() on the string pointer in the desc field to discard the string, and then decrements the number of entries in the stack.

Multi-thread concerns:

Creating the new stack, copying it from the current (thread local) stack, and clearing the thread local stack should not have any multi-thread issues.

Once it is registered, the new instance of H5E_t is potentially accessible to all threads.

To this add any multi-thread issues inherited from H5I_register(), H5I_inc_ref() and H5I_dec_ref().

***

```
herr_t H5Eappend_stack(hid_t dst_stack_id,
                       hid_t src_stack_id,
                       hbool_t close_source_stack);

H5Eappend_stack()
 +-H5I_object_verify()
 |  + …
 +-H5E__append_stack()
 |  +-H5I_inc_ref()
 |  |  +- …
 |  +-H5MM_xstrdup()
 +-H5I_dec_app_ref()
   +- …
```

In a nutshell:

Append the contents of the source error stack to the destination error stack. If directed, decrement the reference count on the source error stack – which will cause it to be deleted if its reference count drops to zero.

In greater detail:

H5Eappend_stack() calls H5I_object_verify() to obtain pointers to the source and destination error stacks (instance of H5E_t). It then passes these pointers to H5E__append_stack() to do the actual append. Finally, if the close_source_stack parameter is TRUE, it calls H5I_dec_app_ref() on the source stack ID to decrement its reference count – will will result in a call to H5E__close_stack() if the reference count drops to zero. See discussion of H5Eclose_stack() below for details on H5E__close_stack()

H5E__append_stack() scans through the source error stack, copying each entry onto the next free entry in the destination error stack. Any excess entries in the source stack are skipped. The function calls H5I_inc_ref() on the contents of the cls_id, maj_num, and min_num fields as they are copied, and uses H5MM_xstrdup() to copy the strings pointed to by the desc fields.

Multi-thread concerns:

H5Eappend_stack() operates on error stacks that have been registered in the H5I_ERROR_STACK index – and are therefore visible to all threads.

To this add any multi-thread issues inherited from H5I_object_verify(), H5I_inc_ref() and H5I_dec_app_ref().

***

herr_t H5Eclose_stack(hid_t stack_id);

```
H5Eclose_stack()
 +-H5I_get_type()
 |  +- …
 +-H5I_dec_app_ref()
   +-H5I__dec_app_ref()
     +-H5I__dec_ref()
     |  +-H5I__find_id()
     |  |  +-…
     |  +-(type_info→cls→free_func)
     |  |  ||      ((void *)info->object, request)
     |  |  ||
     |  | H5E__close_stack() // in this case
     |  |  +-H5E__close_stack()
     |  |    +-H5E_clear_stack()
     |  |      +-H5E__get_my_stack()
     |  |      +-H5E__clear_entries()
     |  |        +-H5I_dec_ref()
     |  |        |  +- …
     |  |        +-H5MM_xfree_const()
     |  |  +-H5FL_FREE()
     |  +-H5I__remove_common()
     +-H5I__find_id()
       +- …
```

In a nutshell:

Decrement the ref count on the supplied stack via a call to H5I_dec_app_ref() – which may cause it to be deleted.

In greater detail:

H5Eclose_stack() first calls H5I_get_type() to verify that the supplied ID refers to an error stack.

If it does, the function calls H5I__dec_ref(), which decrements the regular reference count on the indicated instance of H5E_t in the index. If the regular reference count drops to zero, H5E__close_stack() is called with a pointer to the target instance of H5E_t (this is important, as otherwise H5E__close_stack() would operate on the current stack).

In this context, H5E__close_stack() calls H5E_clear_stack() with a pointer to the target instance of H5E_t. See H5Eget_current_stack() above for a discussion of H5E_clear_stack().

After H5E_clear_stack() returns, H5E__close_stack() frees the target instance of H5E_t.

After H5E__close_stack() returns, the target entry is removed from the index.

Multi-thread concerns:

H5Eclose_stack() operates on an error stack that has been registered in the H5I_ERROR_STACK index – and is therefore visible to all threads.

To this add any multi-thread issues inherited from H5I_get_type(), and H5I_dec_app_ref().

***

```
ssize_t H5Eget_class_name(hid_t class_id, char *name,
                          size_t size);

H5Eget_class_name()
 +-H5I_object_verify()
 |  +- …
 +-H5E__get_class_name()
```

In a nutshell:

Lookup the error class indexed by the supplied ID and return the length of its name. If the supplied buffer exists, copy up to size bytes of this name into *name.

In greater detail:

H5Eget_class_name() calls H5I_object_verify to obtain a pointer to the instance of H5E_cls_t indexed by the supplied ID, passed this pointer to H5E__get_class_name(), and returns whatever that function returns.

H5E__get_class_name() computes the length of the string pointed to by the cls_name field. If the supplied buffer is non-NULL, it copies the string pointed to by the cls_name field into the supplied buffer up to the supplied size – thus the returned class name may be truncated. Finally, it returns the length of the string.

Multi-thread concerns:

H5Eget_class_name() operates on an instance of H5E_cls_t that have been registered in the H5I_ERROR_CLASS index – and is therefore visible to all threads.

To this add any multi-thread issues inherited from H5I_object_verify().

***

herr_t H5Eset_current_stack(hid_t err_stack_id);

```
H5Eset_current_stack()
 +-H5I_object_verify()
 |  +- …
 +-H5E__set_current_stack()
 |  +-H5E__get_my_stack()
 |  +-H5E_clear_stack()
 |  |  +-H5E__get_my_stack()
 |  |  +-H5E__clear_entries()
 |  |  |  +-H5I_dec_ref()
 |  |  |     +- …
 |  |  +-H5MM_xfree_const()
 |  +-H5I_inc_ref()
 |  |  +- …
 |  +-H5MM_xstrdup()
 +-H5I_dec_app_ref()
   +- …
```

In a nutshell:

Clear the thread specific error stack, and then copy the contents of the supplied error stack into it.

In greater detail:

H5Eset_current_stack() calls H5I_object_verify() to obtain a pointer to the supplied error stack (instance of H5E_t). This pointer is passed to H5E__set_current_stack() to clear the thread specific error stack, and then copy the contents of the supplied error stack into it. On return, H5Eset_current_stack() calls H5I_dec_app_ref() to decrement the reference count on the supplied error stack – which may result in the deletion of the supplied stack.

H5E__set_current_stack() calls H5E__get_my_stack() to get a pointer to the thread specific stack, and then clears it with a call to H5E_clear_stack() – see discussion of H5Eget_current_stack() for details.

This done, the function then walks the thread local and supplied stacks in tandem, copying entries from the supplied stack into the thread local stack. In passing, it calls H5I_inc_ref() on the IDs in the cls_id, maj_num, and min_num IDs fields. It also uses H5MM_xstrdup() to copy the strings pointed to by the desc field.

Multi-thread concerns:

The supplied error stack in calls to H5Eset_current_stack() must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads.

To this add any multi-thread issues inherited from H5I_object_verify(), H5I_inc_ref(), and H5I_dec_app_ref().

***

```
herr_t H5Epush2(hid_t err_stack, const char *file,
        const char *func, unsigned line,
        hid_t cls_id, hid_t maj_id, hid_t min_id,
        const char *msg, ...);

H5Epush2()
 +-H5E_clear_stack()
 |  +-H5E__get_my_stack()
 |  +-H5E__clear_entries()
 |    +-H5I_dec_ref()
 |    |  +- …
 |    +-H5MM_xfree_const()
 +-H5I_object_verify()
 |  +- …
 +-H5E__push_stack()
   +-H5E__get_my_stack()
   +-H5I_inc_ref()
     +- …
```

In a nutshell:

If the supplied stack is not H5E__DEFAULT, clear the thread local stack. Then format the supplied error and push it on the supplied error stack, which is the thread local stack if H5E__DEFAULT is supplied.

In greater detail:

H5Epush2() check to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Epush2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack). If the supplied stack ID is H5E_DEFAULT, estack is set to NULL – which indicates that the thread local stack is the target.

Next, H5Epush2() constructs the error description string via a call to HDvasprintf(), and then calls H5E__push_stack(). Here, estack indicates the target error stack. When H5E__push_stack() returns, H5Epush2() discards the buffer allocated by Hdvasprintf() and returns.

On entry, H5E__push_stack tests to see if estack (the pointer to the target error stack) is NULL. If it is, it calls H5E__get_my_stack() to obtain a pointer to the thread local error stack and stores this value in estack.

This done, it examines *estack to find the next available entry on the stack, and copies the supplied data into it. It calls H5E_inc_ref() on the IDs copied into the cls_id, maj_id, and min_id fields, and calls H5MM_xstrdup() and stores a pointer to the new string in the desc field. Finally, it increments the number of entries on the stack and returns. If there is no next available entry on the stack, the function call becomes a NO-OP.

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If it is H5E_DEFAULT, there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Epush2() inherits any multi-thread issues present in H5I_object_verify(), H5I_inc_ref(), and H5I_dec_ref().

***

herr_t H5Epop(hid_t err_stack, size_t count);

H5Epop()

```
+-H5E__get_my_stack()
+-H5I_object_verify()
|  +- …
+-H5E__pop()
  +-H5E__clear_entries()
    +-H5I_dec_ref()
    |  +- …
    +-H5MM_xfree_const()
```

In a nutshell:

If the supplied stack is not H5E__DEFAULT (i.e. the thread local stack), clear the thread local stack. Then pop the specified number of entries off the supplied error stack.

In greater detail:

H5Epop() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Epop() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack). If the supplied stack ID is H5E_DEFAULT, estack is set to NULL – which indicates the thread local stack.

This done, H5Epop() calls H5E__pop() with estack pointing to the target error stack, and the supplied number of entries to pop, and returns.

H5E__pop() simply calls H5E__clear_entries() with the target stack and the number of entries to clear from the top of the stack. See the discussion of H5Eget_current_stack() for the details of H5E__clear_entries().

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If it is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Epop() inherits any multi-thread issues present in H5I_object_verify(), and H5I_dec_ref().

***

```
herr_t H5Eprint2(hid_t err_stack, FILE *stream);
```

```
H5Eprint2()
 +-H5E__print2()
   +-H5E__get_my_stack()
   +-H5I_object_verify()
   |  +- …
   +-H5E__print()
     +-H5E__walk()
       +-H5E__walk2_cb() // in this case
         +-H5I_object_verify()
```

```
typedef struct H5E_print_t {
  FILE *    stream;
  H5E_cls_t cls;
} H5E_print_t;
```

In a nutshell:

Print the specified error stack to the specified file. If the specified error stack is H5E__default, the thread specific error stack is printed.

In greater detail:

H5Eprint2() simply calls H5E__print2() with its parameters and returns the result.

H5E__print2() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Eprint2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack).

If the supplied stack ID is H5E_DEFAULT, estack is set to NULL – which indicates the thread local stack.

Having obtained a pointer to the target error stack, H5E__print2() then calls H5E__print() with the bk_compatible flag set to FALSE to print the error stack, and returns whatever that call returns.

H5E__print() allocates an instance of H5E_walk_op_t (walk_op) on the stack, and (since the bk_compatible flag is FALSE) initializes as follows:

```
walk_op.vers    = 2;
walk_op.u.func2 = H5E__walk2_cb;
```

It then calls H5E__walk() with walk_op as one of its parameters, and returns.

With the above initialization of walk_op, H5E__walk() calls H5E__walk2_cb() on each entry in the error stack.

H5E__walk2_cb() calls H5I_object_verify() on the IDs in cls_id, maj_num, and min_num fields to obtain the necessary strings, and then prints the contents of the target entry in the error stack.

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If it is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Eprint2() inherits any multi-thread issues present in H5I_object_verify().

***

```
herr_t H5Ewalk2(hid_t err_stack, H5E_direction_t direction,
        H5E_walk2_t func, void *client_data);

H5Ewalk2()
 +-H5E__get_my_stack()
 +-H5E_clear_stack()
 |  +-H5E__get_my_stack()
 |  +-H5E__clear_entries()
 |    +-H5I_dec_ref()
 |    |  +- …
 |    +-H5MM_xfree_const()
 +-H5I_object_verify()
 |  +- …
 +-H5E__walk()
   +-(op→u.func2)() // the func parameter passed to
            // H5Ewalk2() in this case.
```

In a nutshell:

Walk the specified error stack in the specified direction calling the specified function on each entry in the stack.

In greater detail:

H5Ewalk2() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Ewalk2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack).

If the supplied stack ID is H5E_DEFAULT, H5Ewalk2() calls H5E__get_my_stack() to get a pointer to the thread local stack and stores it in estack.

H5Ewalk2() then initializes an instance of H5E_walk_op_t as follows:

```
op.vers    = 2;
op.u.func2 = func;
```

where stack_func is the function passed into H5Ewalk2() as a parameter.

Finally, H5Ewalk2() calls H5E__walk() with estack, op, and the parameters it received. When H5E__walk() returns, H5Ewalk2 returns whatever it returns.

H5E__walk() walks the supplied stack in the specified error stack in the specified direction, and calls (op→u.func2)() on each entry in the error stack. In this case, func2 is the func passed into H5Ewalk2().

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If it is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Ewalk2() inherits any multi-thread issues present in H5I_object_verify(), H5I_dec_ref(), and the func parameter passed to H5Ewalk2()..

***

```
herr_t H5Eget_auto2(hid_t estack_id, H5E_auto2_t *func,
        void **client_data);
```

```
H5Eget_auto2()
 +-H5E__get_my_stack()
```

```
+-H5E_clear_stack()
|  +-H5E__get_my_stack()
|  +-H5E__clear_entries()
|    +-H5I_dec_ref()
|    |  +- …
|    +-H5MM_xfree_const()
+-H5I_object_verify()
|  +- …
+-H5E__get_auto()
```

In a nutshell:

Lookup the indicated error stack, and set:

```
*func       = estack->auto_op.func2
*client_data = estack->auto_data
```

if the supplied func and client_data parameters are not NULL.

In greater detail:

H5Eget_auto2() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Eget_auto2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack).

If the supplied stack ID is H5E_DEFAULT, H5Eget_auto2 calls H5E__get_my_stack to get a pointer to the thread local stack and stores it in estack.

H5Eget_auto2() then calls H5E__get_auto() with the estack, &op, and client_data as parameters. Here op is an instance of H5E_auto_op_t that is allocated on the stack.

Assuming op and client_data are not NULL, H5E__get_auto() sets:

```
*op         = estack->auto_op;
*client_data = estack->auto_data;
```

and returns.

After H5E__get_auto() returns, H5Eget_auto2() checks to see if func is not VOID, and if so, it sets *func = op.func2 before returning.

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If it is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Eget_auto2() inherits any multi-thread issues present in H5I_object_verify(), and H5I_dec_ref().

***

```
herr_t H5Eset_auto2(hid_t estack_id, H5E_auto2_t func,
                    void *client_data);

H5Eset_auto2()
 +-H5E__get_my_stack()
 +-H5E_clear_stack()
 |  +-H5E__get_my_stack()
 |  +-H5E__clear_entries()
 |    +-H5I_dec_ref()
 |    |  +- …
 |    +-H5MM_xfree_const()
 +-H5I_object_verify()
 |  +- …
 +-H5E__get_auto()
 +-H5E__set_auto()
```

In a nutshell:

Lookup the indicated error stack, and set:

```
estack->auto_op.func2 = func
estack->auto_data     = client_data
```

where func and client_data are the supplied parameters.

In greater detail:

H5Eset_auto2() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Eset_auto2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack).

If the supplied stack ID is H5E_DEFAULT, H5Eset_auto2() calls H5E__get_my_stack() to get a pointer to the thread local stack and stores it in estack.

This done, H5Eset_auto2() calls H5E__get_auto() to get the current value of the target error stacks auto_oo fields and store it in the local variable op (an instance of H5E_auto_op_t), It then modified op as follows[7]:

```
if (func != op.func2_default)
   op.is_default = FALSE;
else
   op.is_default = TRUE;

op.vers = 2;

op.func2 = func;
```

and then calls H5E__set_auto(estack, &op, client_data), and returns.

H5E__set_auto() performs the assignments:

```
estack->auto_op    = *op;
estack->auto_data = client_data;
```

and then returns.

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If it is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Eset_auto2() inherits any multi-thread issues present in H5I_object_verify(), and H5I_dec_ref(). Further, the function pointed to by the func parameter may also have multi-thread safety issues.

---

7 This is the most complex case. If H5_NO_DEPRECATED_SYMBOLS is defined, this code is significantly simpler.

***

herr_t H5Eclear2(hid_t err_stack);

```
H5Eclear2()
 +-H5E_clear_stack()
 |  +-H5E__get_my_stack()
 |  +-H5E__clear_entries()
 |    +-H5I_dec_ref()
 |    |  +- …
 |    +-H5MM_xfree_const()
 +-H5I_object_verify()
   +- …
```

In a nutshell:

Clear all entries in the specified error stack.

In greater detail:

H5Eclear2() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Eclear2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack). If the supplied stack ID is H5E_DEFAULT, estack is set to NULL – which indicates the thread local stack.

H5Eclear2() then passes estack to H5E_clear_stack() to clear the target stack and returns.

See H5Eset_current_stack() above for the details of H5E_clear_stack().

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If the supplied error stack is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Eclear2() inherits any multi-thread issues present in H5I_object_verify(), and H5I_dec_ref().

***

```
H5_DLL herr_t H5Eauto_is_v2(hid_t err_stack,
                    unsigned *is_stack);

H5Eauto_is_v2()
 +-H5E__get_my_stack()
 +-H5E_clear_stack()
 |  +-H5E__get_my_stack()
 |  +-H5E__clear_entries()
 |    +-H5I_dec_ref()
 |    |  +- …
 |    +-H5MM_xfree_const()
 +-H5I_object_verify()
   +- …
```

In a nutshell:

Determines if the error auto reporting function for an error stack conforms to the H5E_auto_stack_t typedef or the H5E_auto_t typedef. *is_stack is set to 1 for the first case and 0 for the latter case.

In greater detail:

H5Eauto_is_v2() checks to see if the supplied stack ID is H5E_DEFAULT.

If it is not, H5Eauto_is_v2() calls H5E_clear_stack() to clear the thread local stack (see discussion of H5Eget_current_stack() for details), and then calls H5I_object_verify() to obtain a pointer to the specified error stack (estack).

If the supplied stack ID is H5E_DEFAULT, H5Eauto_is_v2() calls H5E__get_my_stack() to get a pointer to the thread local stack and stores it in estack.

If H5_NO_DEPRECATED_SYMBOLS is undefined, and is_stack is not NULL, set *is_stack = (estack->auto_op.vers > 1).

Otherwise, if is_stack is not NULL, set *is_stack = 1;

Multi-thread concerns:

If the supplied error stack is not H5E__DEFAULT, the supplied error stack must be registered in the H5I_ERROR_CLASS_STACK index – and is therefore visible to all threads. If the supplied error stack is H5E__DEFAULT there appear to be no H5E specific multi-thread issues.

Regardless of the above, H5Eauto_is_v2() inherits any multi-thread issues present in H5I_object_verify(), and H5I_dec_ref().

***

```
ssize_t H5Eget_msg(hid_t msg_id, H5E_type_t *type,
        char *msg, size_t size);

H5Eget_msg()
 +-H5I_object_verify()
 |  +- …
 +-H5E__get_msg()
```

In a nutshell:

Retrieve the string associated with the supplied error message ID.

In greater detail:

Call H5I_object_verify() to obtain a pointer (msg) to the instance of H5E_msg_t associated with the supplied message ID.

Then call H5E__get_msg(msg, type, msg_str, size), and return whatever that function returns.

H5E__get_msg() first determines the length of the error message (msg→msg). If the msg_str parameter is not NULL, the function copies up to size bytes of msg→msg into *msg_str. Note that this means that the string in *msg_ptr may contain a truncated version of msg→msg. Finally, H5E__get_msg() returns the length of msg→msg.

Multi-thread concerns:

IH5Eget_msg() requires the ID of an entry in the H5I_ERROR_CLASS_MSG as input – thus the target instance of H5E_msg_t must be visible to all threads.

In addition, H5Eget_msg() inherits any multi-thread issues present in H5I_object_verify().

***

herr_t H5Eclear1(void);

H5Eclear1()
+-H5E_clear_stack()
  +-H5E__get_my_stack()
  +-H5E__clear_entries()
    +-H5I_dec_ref()
    | +- …
    +-H5MM_xfree_const()

In a nutshell:

Clear the thread local stack.

In greater detail:

H5Eclear1() is essentially a pass through function that calls H5E_clear_stack(NULL) to clear the thread local stack, and then returns.
See the discussion of H5Eget_current_stack() for the details of H5E_clear_stack().

Multi-thread concerns:

IH5Eclear1()doesn't touch any data structures that are visible to other threads, and thus has no indigenous multi-thread safety issues.

That said, it calls H5I_dec_ref(), and thus inherits any multi-thread issues that call may have.

***

H5_DLL herr_t H5Eget_auto1(H5E_auto1_t *func,
                    void **client_data);

H5Eget_auto1()
+-H5E__get_my_stack()
+-H5E__get_auto()

In a nutshell:

Returns the current settings for the automatic error stack traversal function and its data for the

thread specific error stack. Either (or both) arguments may be null in which case the value is not returned.

In greater detail:

H5Eget_auto1() first calls H5E__get_my_stack() to obtain a pointer to the thread local error stack (estack). It then calls H5E__get_auto(estack, &auto_op, client_data) where auto_op is an instance of H5E_auto_op_t.

When H5E__get_auto() returns, H5Eget_auto1() performs some sanity checking, sets *func = auto_op.func1, and returns.

See the discussion of H5Eget_auto2() for details on H5E__get_auto().

Multi-thread concerns:

None.

***

```
herr_t H5Epush1(const char *file, const char *func,
        unsigned line, H5E_major_t maj,
        H5E_minor_t min, const char *str);

H5Epush1()
 +-H5E__push_stack()
   +-H5E__get_my_stack()
   +-H5I_inc_ref()
     +- ...
```

In a nutshell:

Format and then push the supplied error on to the thread local error stack.

In greater detail:

H5Epush1() is a pass through functions – it calls H5E__push_stack() and returns.

See H5Epush2() above for a discussion of H5E__push_stack().

Multi-thread concerns:

IH5Epush1() doesn't touch any data structures that are visible to other threads, and thus has no indigenous multi-thread safety issues.

That said, it calls H5I_inc_ref(), and thus inherits any multi-thread issues that call may have.

***

```
herr_t H5Eprint1(FILE *stream);

H5Eprint1()
 +-H5E__get_my_stack()
 +-H5E__print()
   +-H5E__walk()
       +-H5E__walk1_cb() // in this case
         +-H5I_object_verify()
```

In a nutshell:

Walk the thread local error stack and print it contents to the specified file.

In greater detail:

H5Eprint1() first calls H5E__get_my_stack() to obtain a pointer to the thread local error stack (estack). It then calls H5E__print() with the bk_compatible flag set to TRUE, and returns.

H5E__print() allocates an instance of H5E_walk_op_t (walk_op) on the stack, and (since the bk_compatible flag is TRUE) initializes as follows:

```
walk_op.vers    = 1;
walk_op.u.func1 = H5E__walk1_cb;
```

It then calls H5E__walk() with walk_op as one of its parameters, and returns.

With the above initialization of walk_op, H5E__walk() calls H5E__walk1_cb() on each entry in the error stack.

H5E__walk1_cb() calls H5I_object_verify() on the IDs in maj_num, and min_num fields to obtain the necessary strings, and then prints the contents of the target entry in the error stack. Note that unlike H5E__walk2_cb(), H5E__walk1_cb() obtains a pointer to the instance of H5E_cls_t

associated with each entry on the error stack via the pointer to H5E_cls_t stored in the instance of H5E_msg_t associated with the major error ID.

Multi-thread concerns:

H5Eprintf1 operates on the thread local stack, and thus has no H5E specific multi-thread issues.

That said, H5Eprint1() inherits any multi-thread issues present in H5I_object_verify().

***

```
H5_DLL herr_t H5Eset_auto1(H5E_auto1_t func,
                    void *client_data);

H5Eset_auto1()
 +-H5E__get_my_stack()
 +-H5E__get_auto()
 +-H5E__set_auto()
```

In a nutshell:

Turns on or off automatic printing of errors for the thread local error stack. When turned on (non-null func pointer) any API function which returns an error indication will first call func passing it client_data as an argument.

The default values before this function is called are H5Eprint1() with client data being the standard error stream, stderr.

Automatic stack traversal is always in the H5E_WALK_DOWNWARD direction.

In greater detail:

H5Eset_auto1() first calls H5E__get_my_stack() to obtain a pointer to the thread local error stack (estack). It then calls H5E__get_auto(estack, &auto_op, NULL), when auto_op is an instance of H5E_auto_op_t, to get the current value of estack→auto_op.

H5E_set_auto1 the modifies auto_op as follows:

```
estack→auto_op.vers      = 1;
estack→auto_op.func1     = func;
estack→auto_op.is_default = (func == auto_op.func1_default)
```

calls H5E__set_auto(estack, &auto_op, client_data) to apply the desired changes, and returns.

See the discussions of H5Eget_auto2() and H5Eset_auto2() for the details of H5E__get_auto() and H5E__set_auto() respectively.

Multi-thread concerns:

None.

***

herr_t H5Ewalk1(H5E_direction_t direction,
H5E_walk1_t func, void *client_data);

H5Ewalk1()
+-H5E__get_my_stack()
+-H5E__walk()
+-(op→u.func1)() // func in this case
+-(op→u.func2)() // not called in this case

In a nutshell:

Walk the thread local error stack in the specified direction applying the supplied function to each entry.

In greater detail:

H5Ewalk1() first calls H5E__get_my_stack() to obtain a pointer to the thread local error stack (estack). The function then initializes walk_op (an instance of H5E_walk_op_t) as follows:

walk_op.vers = 1;
walk_op.u.func1 = func;

This done, H5Ewalk1() calls

H5E__walk(estack, direction, &walk_op, client_data)

and returns.

H5E__walk() walks the supplied stack in the specified error stack in the specified direction, and calls (op→u.func1)() on each entry in the error stack. In this case, func1 is the func passed into H5Ewalk1().

Multi-thread concerns:

IH5Ewalk1() doesn't touch any data structures that are visible to other threads, and thus has no indigenous multi-thread safety issues.

This said, it will inherit any thread safety issues from the function provided by the user.

***

char *H5Eget_major(H5E_major_t maj);

H5Eget_major()
+-H5I_object_verify()
+-H5E__get_msg()
+-H5MM_malloc()
+-H5MM_xfree()

In a nutshell:

Retrieve the error message associated with the supplied error message ID. Note that the caller must discard the returned string.

In greater detail:

H5Eget_major() calls H5I_object_verify() to obtain a pointer (msg) to the instance of H5E_msg_t associated with the provided ID.

It then calls H5E__get_msg() to get the length of the message string and verify that the error message is a major error message. It then allocates a buffer of suitable size, and calls H5E__get_msg() again to copy the target message into the buffer.

Finally, it returns a pointer to the newly allocated buffer – which must be freed by the caller.

See the discussion of H5Eget_msg() for the details of H5E__get_msg().

Multi-thread concerns:

H5Eget_major() doesn't touch any data structures that are visible to other threads, and thus has no indigenous multi-thread safety issues.

This said, it will inherit any thread safety issues from H5I_object_verify().

***

```
char *H5Eget_minor(H5E_minor_t min);

H5Eget_minor()
 +-H5I_object_verify()
 +-H5E__get_msg()
 +-H5MM_malloc()
 +-H5MM_xfree()
```

In a nutshell:

Retrieve the error message associated with the supplied error message ID. Note that the caller must discard the returned string.

In greater detail:

H5Eget_minor() calls H5I_object_verify() to obtain a pointer (msg) to the instance of H5E_msg_t associated with the provided ID.

It then calls H5E__get_msg() to get the length of the message string and verify that the error message is a minor error message. It then allocates a buffer of suitable size, and calls H5E__get_msg() again to copy the target message into the buffer.

Finally, it returns a pointer to the newly allocated buffer – which must be freed by the caller.

See the discussion of H5Eget_msg() for the details of H5E__get_msg().

Multi-thread concerns:

H5Eget_minor() doesn't touch any data structures that are visible to other threads, and thus has no indigenous multi-thread safety issues.

This said, it will inherit any thread safety issues from H5I_object_verify().

Appendix 2 – H5E Private API Calls

In addition to its public API, H5E also has a small private API. Some of these calls are similar to their cognates in the public API – but there are some differences, and also some calls which offer additional capabilities.

The list of internal H5E API calls below is taken from H5Eprivate.h. Some 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 H5E_init(void);

H5E_init()
 +-H5I_register_type()
 |  +- …
 +-H5E__register_class()
 |  +-H5FL_CALLOC()
 |  +-H5MM_xstrdup()
 +-H5E__create_msg()
 |  +-H5FL_MALLOC()
 |  +-H5MM_xstrdup()
 |  +-H5E__close_msg() // error cleanup only
 |     +-H5MM_xfree()
 |     +-H5FL_FREE()
 +-H5I_register()
   +- …
```

In a nutshell:

Initialize the H5E package.

In greater detail:

H5E_init() first makes calls to H5I_register_type() to create the H5I_ERRCLS_CLS, H5I_ERRMSG_CLS, and H5I_ERRSTK_CLS indexes.

It then calls H5E__register_class() to create the HDF5 library error class, and then registers it in the H5I_ERRCLS_CLS index via a call to H5I_register().

Finally, the function #includes H5Einit.h – which contains a sequence of H5E__create_msg() / H5I_register() calls to create and register in the H5I_ERRMSG_CLS index all the error messages used by the HDF5 library.

Multi-thread Concerns:

H5E_init() initializes all the H5E data structures that are visible to all threads. Thus it must be executed only once, and it must complete before any other thread touches H5E.

***

```
H5_DLL herr_t H5E_printf_stack(H5E_t *estack,
    const char *file, const char *func, unsigned line,
    hid_t cls_id, hid_t maj_id, hid_t min_id,
    const char *fmt, ...) H5_ATTR_FORMAT(printf, 8, 9);

H5E_printf_stack()
 +-HDvasprintf()
 +-H5E__push_stack()
   +-H5E__get_my_stack()
   +-H5I_inc_ref()
     +- …
```

In a nutshell:

Format and then push the supplied error on to the error stack pointed to by the estack parameter.

In greater detail:

H5Eprintf_stack() calls H5vasprintf() to construct the error description string. It then calls H5E__push_stack() to push the error on the indicated error stack, and returns.

See H5Epush2() above for a discussion of H5E__push_stack().

Multi-thread concerns:

H5E_printf_stack() is passed a pointer to an error stack – and thus in principle can access error stacks that are visible to multiple threads. However, a review of the HDF5 source code does not reveal any case in which this parameter is not NULL (recall that a NULL estack parameter passed to H5E__push_stack() causes that call to operate on the thread local error stack).

Since it is a private API call and thus not accessible outside the library, it appears that it doesn't touch any data structures that are visible to other threads. For now, at least, it has no indigenous multi-thread safety issues.

That said, it calls H5I_inc_ref(), and thus inherits any multi-thread issues that call may have.

***

H5_DLL herr_t H5E_clear_stack(H5E_t *estack);

See H5Eget_current_stack() above for a discussion of H5E_clear_stack().

***

H5_DLL herr_t H5E_dump_api_stack(hbool_t is_api);

```
H5E_dump_api_stack()
 +-H5E__get_my_stack()
 |
 | // only one of the following calls is made
 +-(estack->auto_op.func1)(estack->auto_data))
 +-(estack->auto_op.func2)(H5E_DEFAULT, estack->auto_data)
```

In a nutshell:

This function is called at the exit from a public API function call. If the thread local error stack is configured appropriately, it calls a possibly user supplied function that typically prints the contents of the error stack to the supplied file.

In greater detail:

On entry, H5E_dump_api_stack() checks to see if the is_api parameter is TRUE – which is always is at present. If it is, it then calls H5E_get_my_stack() to obtain a pointer (estack) to the thread local error stack.

It then examines *estack to determine which of the auto_op functions (func1 or func2) to call, tests to see if that function is defined, and if so calls it.

The default configuration seems to set

```
estack->auto_op.func1 = (H5E_auto1_t)H5Eprint1;
estack->auto_op.func2 = (H5E_auto2_t)H5E__print2;
```

with the choice of which to use driven by various compilation switches. See the discussion of H5Eprint1() and H5Eprint2() above for discussion of these functions.

Multi-thread concerns:

Since H5E_dump_api_stack() operates only on the thread local stack, there should be no mult-thread safety concerns with the default functions beyond any issues inherited from H5I calls.

However, H5Eset_auto2() and H5Eset_auto1() can set arbitrary functions in the thread local error stack – which in principle can introduce multi-thread safety issues.

Appendix 3 – Manual Expansions of Macros

Much of the management of error handling in HDF5 is done via macros – most particularly the func enter and exit macros, and HGOTO_ERROR and its relatives.

These macros are constructed with a great deal of nesting, which makes them difficult to follow. As part of my preparation for this sketch design, I manually expanded several macros so that I could lay out their code in one continuous block for ease of understanding and reference. I have included these expansions here on the chance that the reader may find them useful as well.

The first of these are the FUNC_ENTER_API() and FUNC_LEAVE_API() macros that are called at the beginning and end of HDF5 API calls. The point of interest here is code used to dump the error stack in FUNC_LEAVE_API()

Note that nested macro expansions are indicated as follows:

```
** <invocation of nested macro> **
   <body of macro expansion – which may contain further macro expansion>
** **
```

Not all nested macros are expanded. HGOTO_ERROR() in particular is dealt with later in this appendix

---

```
#define FUNC_ENTER_API(err)
  {
    {
      hbool_t api_ctx_pushed = FALSE;

      ** FUNC_ENTER_API_COMMON **
        ** FUNC_ENTER_API_VARS **
          ** MPE_LOG_VARS **
            static int eventa(__func__) = -1;
            static int eventb(__func__) = -1;
            char      p_event_start[128];
          ** **
          ** H5TRACE_DECL ** // usually a no-op
            const char *RTYPE = NULL;
            double     CALLTIME;
          ** **
        ** **
        ** FUNC_ENTER_COMMON(H5_IS_API(__func__)); **
          hbool_t err_occurred = FALSE;
          FUNC_ENTER_CHECK_NAME(asrt); // asrt == H5_IS_API(__func__)
                                       // checks to see if fcn name is of API format
          // FUNC_ENTER_CHECK_NAME is a no-op in production mode.
        ** **
        ** FUNC_ENTER_API_THREADSAFE; **
          /* Initialize the thread-safe code */
          ** H5_FIRST_THREAD_INIT **  // no-op in single thread build
            pthread_once(&H5TS_first_init_g, H5TS_pthread_first_thread_init);
```

```
        ** **

        /* Grab the mutex for the library */
        ** H5_API_UNSET_CANCEL **  // no-op in single thread build
           H5TS_cancel_count_inc();
        ** **
        ** H5_API_LOCK ** // no-op in single thread build
           H5TS_mutex_lock(&H5_g.init_lock);
        ** **
      **
    ** **
    ** FUNC_ENTER_API_INIT(err); **
      /* Initialize the library */
      if (!H5_INIT_GLOBAL && !H5_TERM_GLOBAL) {
        // H5_INIT_GLOBAL == (H5_g.H5_libinit_g) if thrad-safe, (H5_libinit_g) if not
        // H5_TERM_GLOBAL == (H5_g.H5_libterm_g) if thread-safe, (H5_libterm_g) if not
        if (H5_init_library() < 0)
          HGOTO_ERROR(H5E_FUNC, H5E_CANTINIT, err, "library initialization failed")
      }

    ** **
    ** FUNC_ENTER_API_PUSH(err); **
      /* Push the name of this function on the function stack */
      ** H5_PUSH_FUNC **
        H5CS_push(__func__); // no-op if H5_HAVE_CODESTACK undefined
      ** **

      /* Push the API context */
      if (H5CX_push() < 0)
        HGOTO_ERROR(H5E_FUNC, H5E_CANTSET, err, "can't set API context")
      else
        api_ctx_pushed = TRUE;

      ** BEGIN_MPE_LOG **
        if (H5_MPEinit_g) {
          snprintf(p_event_start, sizeof(p_event_start), "start %s", __func__);
          if (eventa(__func__) == -1 && eventb(__func__) == -1) {
            const char *p_color = "red";
            eventa(__func__)    = MPE_Log_get_event_number();
            eventb(__func__)    = MPE_Log_get_event_number();
            MPE_Describe_state(eventa(__func__), eventb(__func__), __func__,
                       p_color);
          }
          MPE_Log_event(eventa(__func__), 0, p_event_start);
        }
      ** **
    ** **
    /* Clear thread error stack entering public functions */
    H5E_clear_stack(NULL);
    {


#define FUNC_LEAVE_API(ret_value)
      ;
    } /*end scope from end of FUNC_ENTER*/
    ** FUNC_LEAVE_API_COMMON(ret_value); **
      ** FINISH_MPE_LOG ** // no-op if H5_HAVE_MPE not defined
        if (H5_MPEinit_g) {
          MPE_Log_event(eventb(__func__), 0, __func__);
        }
      ** **
```

```
      ** H5TRACE_RETURN(ret_value); ** // no-op if H5_DEBUG_API no defined
        if (RTYPE) {
          H5_trace(&CALLTIME, __func__, RTYPE, NULL, V);
          RTYPE = NULL;
        }
      ** **
    ** **
    if (api_ctx_pushed) {
      (void)H5CX_pop(TRUE);
      api_ctx_pushed = FALSE;
    }
    ** H5_POP_FUNC **
      H5CS_pop(); // no-op if H5_HAVE_CODESTACK not defined
    ** **
    if (err_occurred)
      (void)H5E_dump_api_stack(TRUE);
    ** FUNC_LEAVE_API_THREADSAFE **
      ** H5_API_UNLOCK **
        5TS_mutex_unlock(&H5_g.init_lock);
      ** **
      ** H5_API_SET_CANCEL **
        H5TS_cancel_count_dec(); // no-op if H5_HAVE_THREADSAFE undefined
      ** **
    ** **
    return (ret_value);
  }
} /*end scope from beginning of FUNC_ENTER*/
```

While there are several alternative versions, the FUNC_ENTER_NOAPI() and FUNC_LEAVE_API() macros typically appear at the beginning and end of internal functions in HDF5. As can be seen, they have little to do with H5E, being primarily concerned with maintaining the call stack, and some sanity checking.

```
#define FUNC_ENTER_NOAPI(err)
  {
    ** FUNC_ENTER_COMMON(!H5_IS_API(__func__)); **
      hbool_t err_occurred = FALSE;
      FUNC_ENTER_CHECK_NAME(asrt); // asrt == H5_IS_API(__func__)
                          // checks to see if fcn name is of API format
      // FUNC_ENTER_CHECK_NAME is a no-op in production mode.
    ** **
    /* Push the name of this function on the function stack */
    ** H5_PUSH_FUNC **
      H5CS_push(__func__); // no-op if H5_HAVE_CODESTACK undefined
       +-H5CS_get_my_stack() // H5CS__get_stack() if thread safe
       |  +-H5TS_get_thread_local_value() // really pthread_getspecific()
       |  +-HDmalloc()
       |  +-H5TS_set_thread_local_value() // really pthread_setspecific()
       +-HDmalloc
    ** **
    {

#define FUNC_LEAVE_NOAPI(ret_value)
  ;
  } /*end scope from end of FUNC_ENTER*/
  ** H5_POP_FUNC **
    H5CS_pop(); // no-op if H5_HAVE_CODESTACK not defined
```

```
      +-H5CS_get_my_stack() // H5CS__get_stack() if thread safe
   ** **
   return (ret_value);
} /*end scope from beginning of FUNC_ENTER*/
```

While there are some variants, the vast majority of errors in the HDF5 library are flagged via the GOTO_ERROR() macro. As errors propagate up the call stack, the calls to H5E_printf_stack() construct the error stack that is eventually displayed.

```
/*
 * HGOTO_ERROR macro, used to facilitate error reporting between a
 * FUNC_ENTER() and a FUNC_LEAVE() within a function body.  The arguments are
 * the major error number, the minor error number, the return value, and an
 * error string.  The return value is assigned to a variable `ret_value' and
 * control branches to the `done' label.
 */
#define HGOTO_ERROR(maj, min, ret_val, ...)
   {
      ** HCOMMON_ERROR(maj, min, __VA_ARGS__); **
         ** HERROR(maj, min, __VA_ARGS__); **
            H5E_printf_stack(NULL, __FILE__, __func__, __LINE__, H5E_ERR_CLS_g,
                        maj_id, min_id, __VA_ARGS__)
         ** **
         err_occurred = TRUE;
         err_occurred = err_occurred; /* Shut GCC warnings up! */
      ** **
      ** HGOTO_DONE(ret_val) **
         {
            ret_value = ret_val;
            goto done;
         }
      ** **
   }
```
