Making H5E Multi-Thread Safe:

A Sketch Design

John Mainzer

Lifeboat, LLC

7/13/22

# 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 it into case 2 above), or by addressing multi-thread safety in the entire cycle simultaneously. Repeat until the HDF5 library is multi-thread safe.

The above process, if followed to its conclusion, will clearly 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](https://docs.hdfgroup.org/hdf5/rfc/RFC_multi_thread.pdf), 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 RFC, 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 suggestions for improvement, and in the hope that reviewers will point out any misconceptions or errors.

# A Quick Overview of H5E

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

* the name of the function,
* the name of the source code file in which the function is defined,
* the source code line number on which the error occurred,
* identifier (ID) indicating the class of the error and its major and minor error numbers[[1]](#footnote-2), and
* an error message

are pushed on an error stack[[2]](#footnote-3) (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 and some indication of why the error occurred, but also at least the top of the call stack at the time the error was detected – which is frequently quite useful.

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 H5I facilities to report them when appropriate. To my knowledge, VOL developers are the main (and perhaps only) 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 API 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]](#footnote-4)

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:

* H5I\_ERROR\_CLASS
* H5I\_ERROR\_MSG
* H5I\_ERROR\_STACK

(See the declaration of the [H5I\_type\_t](http://docs.hdfgroup.org/hdf5/develop/_h5_ipublic_8h.html%23a13afe14178faf81b89fa2167e7ab832b) 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 allows the user to create and register / unregister 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 IDs are used in the error stack and are referenced when printing the 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/ unregister 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 \*/   
};

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 for locking on the error stacks proper. However, as above there are public APIs that create and register/ unregister 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.

This said, it is hard to come up with a scenario where multiple threads would want to access the same error stack – which is a point to consider when it comes to solutions.

With this background in place, we proceed to a discussion of the H5E public API with reference.

## Multi-thread thread issues in the H5E public API

Appendix 1 contains a list of all H5E public APIs, along with brief discussions of their function and potential multi-thread issues. While the reader is invited to review this appendix 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.

### Deletion of error classes and messages out from under other threads

The H5Eunregister\_class() call decrements the reference count on the target of H5E\_cls\_t. If this reference count drops to zero, all instances of H5E\_msg\_t are deleted from the index regardless of their reference counts, along with the target instance of H5E\_cls\_t. Needless to say, this would be inconvenient for other threads that make reference to these instances of H5E\_msg\_t either indirectly or through explicit H5E API calls (i.e. H5Eget\_class\_name(), H5Eget\_msg(), etc.).

Similarly, the H5Eclose\_msg() call decrements the application and regular reference counts on the target instance of H5E\_msg\_t – resulting in its deletion from the index if the regular reference count drops to zero. While not as potentially catastrophic as H5Eunregister\_class(), the issue must still be dealt with.

### Potential Error Stack Corruption

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 issues with H5E\_cls\_t and H5E\_msg\_t.[[4]](#footnote-5)

However, there are API calls to create and register/ unregister 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.

That said, we must keep in mind that none of these actions are an issue as long as they are executed sequentially by a single thread – which is currently enforced by the HDF5 library global lock. Further, one has to try hard to come up with a scenario in which it would be useful to allow multiple threads to operate on the same error stack concurrently. This is a point that must be considered carefully when proposing solutions to these issues.

# Solutions and their Discontents

\*\*\* working on this section now \*\*\*

# Appendix 1 – H5E public API calls

This appendix contains a list of the H5E public API calls, along with 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 descriptions were obtained via inspection of the code[[5]](#footnote-6). Multi-thread safety related issues are underlined. 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);

Create 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.

Observe that once the hid\_t associated with the new instance of H5E\_cls\_t is returned, in principle it is simultaneously accessible by all threads.   
  
\*\*\*

herr\_t H5Eunregister\_class(hid\_t class\_id);

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 itterates 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.

\*\*\*

herr\_t H5Eclose\_msg(hid\_t err\_id);

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 intance of H5E\_msg\_t proper,

After H5E\_\_close\_msg() returns, the associated entry is removed from the index.

\*\*\*

hid\_t H5Ecreate\_msg(hid\_t cls, H5E\_type\_t msg\_type,

const char \*msg);

Allocates and initializes an instance of H5E\_msg\_t, calls H5I\_register() to insert it into the index, and returns the associated hid\_t. 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.

Observe that once the hid\_t associated with the new instance of H5E\_msg\_t is returned, in principle it is simultaneously accessible by all threads.

\*\*\*

hid\_t H5Ecreate\_stack(void);

Allocate and initialize an error stack (i.e., an 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.

Observe that once the hid\_t associated with the new error stack is returned, in principle it is simultaneously accessible by all threads.  
  
 \*\*\*

hid\_t H5Eget\_current\_stack(void);

Create a new error stack (instance of H5E\_t) and copy the current error stack into it, register it in the index via call to H5I\_register(), and return the hid\_t. The current error stack is cleared in passing.

In the MT case, the current error stack will be thread specific. Thus copying it into the new error stack and clearing the current error stack should have no mult-thread significance.

Observe that once the hid\_t associated with the new error stack is returned, in principle it is simultaneously accessible by all threads.

\*\*\*

herr\_t H5Eappend\_stack(hid\_t dst\_stack\_id,

hid\_t src\_stack\_id,

hbool\_t close\_source\_stack);

Operates on two error stacks that are identified by supplied hid\_t’s.   
  
Append the contents of the src\_stack to dst\_stack – if there are not enough empty slots in dst\_stack, excess entries from the src\_stack are not copied into the dst\_stack.   
  
As each error message is copied from the src to dst stack, the ref counts of the class id, major error id, and minor error id are all incremented via call to H5I\_inc\_ref().   
  
If directed, the ref count on src\_stack is decremented via a call to H5I\_dec\_app\_ref(). If this ref count is decremented to zero, H5E\_\_close\_stack() will be called – see discussion of H5Eclose\_stack() for details.  
  
\*\*\*

herr\_t H5Eclose\_stack(hid\_t stack\_id);

Decrement the ref count on the supplied stack via a call to H5I\_dec\_app\_ref().

H5I\_\_dec\_app\_ref() 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.

H5E\_clear\_stack() in turn calls H5E\_\_clear\_entries() with the a pointer to the target instance of H5E\_t, and nentries (the number of entries to clear) set to the number of entries in the error stack.

Starting from the top of the stack, H5E\_\_clear\_entries() scans each entry, using H5I\_dec\_ref() to decrement the reference count on the minor error, major error, and error class of each entry until the specified number of entries have been cleared. If an entry has a description string, that is freed as well. Finally, H5E\_\_clear\_entries() updates the nused field of the error stack to reflect the number of entries cleared – in this case, all of them.

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.  
  
\*\*\*

ssize\_t H5Eget\_class\_name(hid\_t class\_id, char \*name,

size\_t size);

Accessor routine that looks up the instance of H5E\_cls\_t referenced by the supplied hid\_t (via H5I\_object\_verify), and returns the length of the target's cls\_name field. If supplied, and if it is large enough, copies the cls\_name into the supplied name buffer.   
  
\*\*\*

herr\_t H5Eset\_current\_stack(hid\_t err\_stack\_id);

Discards the contents of the current (i.e. thread specific) stack via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.

Then copies the contents of the error stack indicated by the supplied err\_stack\_id into the current error stack -- incrementing the ref counts on the error class, and major, and minor errors of each entry in passing, and also duplicating entry description strings where present.   
  
In the MT build, the current error stack is thread specific, thus there are no issues with either clearing it or copying entries into it. However, this is not the case with the source error stack, since it is registered in the index.

\*\*\*

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, ...);

Clear the current error stack (i.e. the thread local error stack) if the supplied err\_stack parameter is not H5E\_\_DEFAULT).   
  
Format and then push the supplied error on to the supplied error stack via a call to H5E\_\_push\_stack(). If the err\_stack parameter is H5E\_DEFAULT, the supplied error stack is the current (or thread local) stack. Otherwise, the supplied stack is obtained via a call to H5I\_object\_verify().

Note that H5E\_\_push\_stack() increments the ref count on the supplied class, major, and minor errors, and duplicates the supplied description string. If the stack is full, the error message is discarded.  
  
\*\*\*

herr\_t H5Epop(hid\_t err\_stack, size\_t count);

Pop the specified number of entries of the specified stack. Do this via a call to H5E\_\_pop() – which calls H5E\_\_clear\_entries() to perform this operation. See discussion of H5Eclose\_stack() for a discussion of this function.

\*\*\*

herr\_t H5Eprint2(hid\_t err\_stack, FILE \*stream);

If the supplied err\_stack parameter is not H5E\_\_DEFAULT, discard the contents of the current stack (i.e. the thread local stack) via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.  
  
Call H5E\_\_print2() to print the contents of the supplied error stack (which will be the thread specific error stack if the supplied err\_stack parameter is H5E\_\_DEFAULT) to the specified file.

H5E\_\_print2() call H5E\_\_print() with the bk\_compatible flag set to FALSE.

In this context, H5E\_\_print() walks the stack via a call to H5E\_\_walk() – which in turn calls H5E\_\_walk2\_cb() on each entry starting from the top of the stack.

H5E\_\_walk2\_cb() prints the contents of the target stack entry to the supplied file, or to stderr if no file is provided. Note that in doing so, it must look up the listed error class, and major and minor errors in the index.  
  
\*\*\*

herr\_t H5Ewalk2(hid\_t err\_stack, H5E\_direction\_t direction,

H5E\_walk2\_t func, void \*client\_data);

If the supplied err\_stack parameter is not H5E\_\_DEFAULT, discard the contents of the current stack (i.e. the thread local stack) via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.   
  
Walk the supplied stack (which will be the thread specific error stack if the supplied err\_stack parameter is H5E\_\_DEFAULT) in the indicated direction applying the supplied operation.

Do this via a call to H5E\_\_walk(), which in turn calls the supplied function of each entry in supplied error stack, starting at the top or bottom of the stack as directed.  
  
\*\*\*

herr\_t H5Eget\_auto2(hid\_t estack\_id, H5E\_auto2\_t \*func,

void \*\*client\_data);

If the supplied err\_stack parameter is not H5E\_\_DEFAULT, discard the contents of the current stack (i.e. the thread local stack) via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.

Obtain a pointer to the specified error stack (i.e. instance of H5E\_t) -- which will be the thread specific error stack if the supplied err\_stack parameter is H5E\_\_DEFAULT. Call this pointer estack.

If the func parameter is not NULL, set \*func = estack->auto\_op.func2.

If the client\_data parameter is not NULL, set \*client\_data = estack->auto\_data;

Whether there are multi-thread issues with this function depends on the estack\_id parameter. If it is H5E\_DEFAULT (indicating that the thread local error stack is the target), there should be no issues. If not, there certainly are.  
  
\*\*\*

herr\_t H5Eset\_auto2(hid\_t estack\_id, H5E\_auto2\_t func,

void \*client\_data);

If the supplied err\_stack parameter is not H5E\_\_DEFAULT, discard the contents of the current stack (i.e. the thread local stack) via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.

Obtain a pointer to the specified error stack (i.e. instance of H5E\_t) -- which will be the thread specific error stack if the supplied err\_stack parameter is H5E\_\_DEFAULT. Call this pointer estack.

Allocate an instance of H5E\_auto\_op\_t – call this instance op. Initialize op.func2 = func.[[6]](#footnote-7)

Set estack->auto\_op = op, and estack->auto\_data = client\_data.

Whether there are multi-thread issues with this function depends on the estack\_id parameter. If it is H5E\_DEFAULT (indicating that the thread local error stack is the target), there should be no issues. If not, there certainly are.

\*\*\*

herr\_t H5Eclear2(hid\_t err\_stack);

If the supplied err\_stack parameter is not H5E\_\_DEFAULT, discard the contents of the current stack (i.e. the thread local stack) via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.

Clear the supplied error stack – which will be the thread specific error stack if the supplied err\_stack parameter is H5E\_\_DEFAULT. Do this via a call to H5E\_clear\_stack().

Multi-thread safety issues depend on the value of the err\_stack parameter.

\*\*\*

H5\_DLL herr\_t H5Eauto\_is\_v2(hid\_t err\_stack,

unsigned \*is\_stack);

If the supplied err\_stack parameter is not H5E\_\_DEFAULT, discard the contents of the current stack (i.e. the thread local stack) via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details.

Obtain a pointer to the specified error stack (i.e. instance of H5E\_t) -- which will be the thread specific error stack if the supplied err\_stack parameter is H5E\_\_DEFAULT. Call this pointer 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 safety issues depend on the value of the err\_stack parameter.

\*\*\*

ssize\_t H5Eget\_msg(hid\_t msg\_id, H5E\_type\_t \*type,

char \*msg, size\_t size);

Obtain a pointer to the instance of H5E\_msg\_t indexed by the supplied msg\_id. Call this pointer msg.

Set the return value of the function = strlen(msg->msg).

If the supplied buffer is large enough, copy msg->msg into \*msg.

If type is not NULL, set \*type = msg->type. (i.e. whether the error message is major or minor).

\*\*\*

herr\_t H5Eclear1(void);

Clear the thread local stack via a call to H5E\_clear\_stack() – see discussion of H5Eclose\_stack() for details. .   
  
\*\*\*

H5\_DLL herr\_t H5Eget\_auto1(H5E\_auto1\_t \*func,

void \*\*client\_data);

Get a pointer to the current (i.e. thread local) error stack – call this value estack.

If client\_data is not NULL, set \*client\_data = estack->autodata.

If func is not NULL, set \*func = estack->auto\_data.func1.  
  
As this call only touches the thread local error stack, there should be no multi-thread safety issues.  
  
\*\*\*

herr\_t H5Epush1(const char \*file, const char \*func,

unsigned line, H5E\_major\_t maj,

H5E\_minor\_t min, const char \*str);

Format and then push the supplied error on to the current (i.e. thread local) error stack via a call to H5E\_\_push\_stack().

Note that H5E\_\_push\_stack() increments the ref count on the supplied class, major, and minor errors, and duplicates the supplied description string. If the error stack is full, the error message is discarded.  
  
\*\*\*

herr\_t H5Eprint1(FILE \*stream);

Walk the current (i.e. thread local) error stack and print its contents.

Call H5E\_\_print() with the bk\_compatible flag set to TRUE to print the contents of the current (i.e. thread local) error stack to the specified file.

In this context, H5E\_\_print() walks the stack via a call to H5E\_\_walk() – which in turn calls H5E\_\_walk1\_cb() on each entry starting from the top of the stack.

H5E\_\_walk1\_cb() prints the contents of the target stack entry to the supplied file, or to stderr if no file is provided. Note that in doing so, it must look up the listed major and minor errors in the index. It accesses the H5E\_cls\_t pointed to by the instance of H5E\_msg\_t associated with the major error number.

\*\*\*

H5\_DLL herr\_t H5Eset\_auto1(H5E\_auto1\_t func,

void \*client\_data);

Get a pointer to the current (i.e. thread local) error stack – call this value estack.

Set:

estack→auto\_op.vers = 1;

estack→auto\_op.func1 = func;

estack→auto\_op.is\_default = (func == auto\_op.func1\_default)  
estack->auto\_data = client\_data;

Since this call only touches the thread local error stack, there should be no multi-thread safety issues.  
  
\*\*\*

herr\_t H5Ewalk1(H5E\_direction\_t direction,

H5E\_walk1\_t func, void \*client\_data);

Walk the current (i.e. the thread specific) in the indicated direction applying the supplied operation.

Do this via a call to H5E\_\_walk(), which in turn calls the supplied function of each entry in supplied error stack, starting at the top or bottom of the stack as directed.

\*\*\*

char \*H5Eget\_major(H5E\_major\_t maj);

Look up (via a call to H5I\_object\_verify() the instance of H5E\_msg\_t associated with the maj parameter (which is really an hid\_t).   
  
Allocate a string of appropriate length and load it with the error message associated with the target instance of H5E\_msg\_t. The caller must discard this buffer when done.

\*\*\*

char \*H5Eget\_minor(H5E\_minor\_t min);

Look up (via a call to H5I\_object\_verify() the instance of H5E\_msg\_t associated with the  min parameter (which is really an hid\_t).   
  
Allocate a string of appropriate length and load it with the error message associated with the target instance of H5E\_msg\_t. The caller must discard this buffer when done.

# 

# Appendix 2 – 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 relative.

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;

}

\*\* \*\*

}

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. [↑](#footnote-ref-2)
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 2 for a manual expansion of this macro,  
    [↑](#footnote-ref-3)
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 will have much to recommend it. If not, concerns for lock ordering issues may make it advisable to avoid the use of H5I in H5E. [↑](#footnote-ref-4)
4. Entries in error stacks are maintained in an array of instance of H5E\_error2\_t (see H5Epubic.h). These in turn contain the IDs of the relevant error classes, and major and minor errors. [↑](#footnote-ref-5)
5. And thus likely contain errors. Corrections are welcome. [↑](#footnote-ref-6)
6. If H5\_NO\_DEPRECIATED\_SYMBOLS is undefined, there are further initializations. [↑](#footnote-ref-7)