RFC: API Contexts

Quincey Koziol

Within the HDF5 library code, there is frequently a need to have a place to store information during an API operation that is "global" to that operation. Either the information is accessed frequently from many locations or from vastly different levels of the call stack, or is a shared state that is dispersed throughout the library.

Creating an "API context" that is instantiated per API call (and is also thread-specific and re-entrant safe) allows the library to have a "bulletin board" for posting and retrieving information about the current API operation. This API context eliminates the need to pass around a property list through many layers of the internal library code, and also is higher performance than accessing properties in a property list.

1 Introduction

The HDF5 library has historically used properties in a dataset transfer property list (DXPL) to store and retrieve information about the current API operation while it was proceeding. However, there are several inherent flaws with this usage: getting / setting properties in a property list has turned out to be slower than anticipated; it's poor programming to modify a DXPL that an application passed in, so the library would duplicate non-default DXPLs when they needed to be modified; a DXPL parameter needed to be passed to / through all the internal library calls that eventually called a routine that performed I/O (since the DXPL needed to be passed to the VFD invoked), leading to increased overheard in forwarding this parameter along through many calls that didn't actually use it directly.

With this usage of DXPLs within the library, it has become essentially a "global" variable that is instantiated for each API call, for each thread, but with all the downsides listed above. Instead of using a DXPL, a separate "API context" could be created for each API call, one which provided the same functionality (a place to get / set values used at many levels within the library during an API operation), but without the downsides of passing property lists around.

This document presents an overview of the API context design and implementation, along with guidelines for using it and adding new fields to the context.

2 Approach / Design / Etc.

The API context code within the library has several requirements and design goals:

- A) Provide a location where values used during an API call can be get / set from many locations within the library, without passing an extra parameter to all the calls, and without succumbing to the software engineering downsides of global variables.

- B) Get / set values in the context quickly
- C) Have low memory overhead
- D) Accommodate re-entering the library through callbacks that make HDF5 API calls
- E) Be thread-safe

With these targets in mind, the following aspects of the API context code implementation fulfill the requirements:

- An API context is initialized / destroyed with calls to H5CX_push() / H5CX_pop() in the FUNC_ENTER_API* / FUNC_LEAVE_API* macros for API routines. Therefore it is immediately and pervasively available to all internal library code, without special setup or internal parameter passing. (A)
- API context information is only accessible through accessor ("get" & "set") routines in the H5CX interface, protecting the internal API context information from being modified arbitrarily by library code. This helps to prevent abuse of the "global" nature of the API context, and also provides better support for debugging and profiling the API context implementation. (A)
- API contexts are designed to replace the use of passing DXPLs and LAPLs around in internal library calls. They interface transparently with properties from those property lists, retrieving properties on an on-demand basis and, when necessary, updating those properties in a DXPL or LAPL when the API context is destroyed. (A)
- API context fields are designed to have high-performance access. Fields that are strictly internal and don't correspond to a property in a DXPL or LAPL don't call the H5P routines at all, and fields that correspond to a property interact with the H5P routines in a minimal way, retrieving properties only on demand. (B)
- API contexts are designed to have a minimal memory impact. Only one memory allocation occurs when an API context is initialized, and even then, the internal "free list" code is used to avoid allocating system memory more than necessary. (C)
- Because the HDF5 library allows calling HDF5 API routines from callbacks (e.g. H5Literate or H5Ovisit), the API context implementation must allow for re-entering the library without mixing the context between two calls. This is accommodated by implementing API contexts on a stack, with the context from each new entrance into the HDF5 library pushed on the top of the stack upon initialization and popped from the stack when destroyed. (D)
- API contexts are initialized after the threadsafety concurrency semaphore that protects the library during multi-threaded access is acquired for an API routine, so they don't technically require any additional protection for correct threadsafe operation. However, with an eye toward the future, where multi-threaded concurrency within the HDF5 library is a possibility, API context stacks are built on top of a thread-local variable. This means that each thread in a calling application will get a unique API context stack and will not share that state with them. Callbacks with a thread share the same API context, pushing new contexts on the stack when the library is re-entered. (E)

3 Implementation Details

In this section, the API context implementation is discussed.

3.1 API Context Struct (H5CX_t)

Each API call creates a new API context by allocating (using the internal free-list allocation (H5FL) code) a new H5CX_node_t struct, initializing it to default values (in H5CX__push_common) and pushing it onto the thread's API context stack (in H5CX_push / H5CX_push_special). This struct contains a H5CX_t struct that holds the actual information for an API context, along with a 'next' pointer to link the stack data structure together. The H5CX_t struct has 1+ members for each "field" that is accessible through the H5CX interface (the members for a field are described in section 3.5, below). The members in the struct are divided into several sections, based on their usage in the code and the structure / naming of the members is described further below.

3.2 Initialization of Cached API Context Fields

The H5CX interface initialization routine (H5CX__init_package) populates static structures containing cached default values for the DXPL and LAPL properties that interface with the API context interface. Once initialized, this cached data is constant for the duration of the application and is not modified further, even at interface shutdown. When a default DXPL or LAPL is used for an API operation, this cached value is used (on demand) to populate the corresponding field in the API context instead of looking up the property value in the default DXPL / LAPL.

3.3 API Context Setup

When an API routine in the library is entered, that routine may set non-default DXPL and LAPL values for the API context (which is created in its FUNC_ENTER_API macro), with H5CX_set_apl, H5CX_set_loc, H5CX_set_dxpl, and H5CX_set_lapl:

- H5CX_set_apl is the most complicated of these, and performs three functions: setting the access property list (LAPL) for both the API routine itself and the API context, checking if the application has requested collective metadata reads for this operation (and setting the API context field indicating this), and optionally sanity checking parallel HDF5 API operations that are collective. H5CX_set_apl should be called for all API routines that have an access property list as a parameter.
- H5CX_set_loc is next in complexity and sets the API context field for collective metadata reads and optionally sanity checks collective parallel HDF5 API operations. H5CX_set_loc should be called from all HDF5 API routines that modify file metadata but don't pass an access property list parameter to the API routine.
- H5CX_set_lapl is a supplement to the H5CX_set_apl routine and should be called for all API routines which pass both an access property list for the object's type (i.e. attribute, dataset, or group) and a link access property list to the API routine. In this case, the object's access property list should be passed to H5CX_set_apl (to set up collective access, etc) and then H5CX_set_lapl can be called to set the actual LAPL for the API context.
- H5CX_set_dxpl should be called for all API routines that have a DXPL parameter. This API context routine sets the DXPL to use for the operation to a non-default value, from the application.

Note that all of the API context routines above that set internal property lists do not copy the property list, but use it directly.

3.4 "Special" Push & Pop

H5CX_push_special and H5CX_pop_special are only used one place within the HDF5 library: in the library termination routine (H5_term_library). They are identical to H5CX_push / H5CX_pop except that they use malloc() and free() to allocate and release an API context struct (instead of the library's free-list allocation and release routines) so that they don't depend on other library interfaces (which are about to be shut down).

3.5 API Context fields

"Fields" in the H5CX interface correlate to get / set routines for the interface (in src/H5CXprivate.h), may correspond to a property in a DXPL or LAPL, and can be composed of several members in the H5CX_t struct. For example, the "btree_split_ratio" field in the API context can be interacted with using the H5CX_get_btree_split_ratios / H5CX_set_btree_split_ratios accessor routines from other modules in the library, corresponds to the H5D_XFER_BTREE_SPLIT_RATIO_NAME property in a DXPL, and is composed of the btree_split_ratio and btree_split_ratio_valid members of the H5CX_t struct.

There are four types of fields in the H5CX_t struct:

- Reference Property Lists: The property lists to use for the operation, either default or application set. The DXPL and LAPL ID in the H5CX_t struct are either the default property list for the library (if the API routine doesn't have a DXPL or access property list parameter), or the property list that the application passed in (which could be the default property list). The 'dxpl' and 'lapl' members of the H5CX_t struct point to the underlying property list for the corresponding ID, and are just used for faster access to the property list when multiple properties are retrieved. Non-default DXPLs and LAPLs for an API context are not copied, but are used directly. Reference property lists are retrieved from the API context to pass to application-defined callbacks in several interfaces (H5FD, external links, etc).
- Internal Fields: Internal API context fields are used for managing various state within the library and don't correspond to a DXPL or LAPL property. Internal fields are typically set somewhere high in the library and consumed or retrieved at a much lower level within the library. For example, the metadata "tag" for an object is typically set in the group or dataset code, and then checked in the metadata cache. Since these fields don't correspond to properties, they may only use one member in the H5CX_t struct and don't have a particular naming scheme.
- Cached Fields: Cached fields in the API context struct correspond to a property in the DXPL or LAPL. These fields are usually retrieved from the underlying property list into the API context on demand. However, if the underlying property list is a library default list, the value used for the field is retrieved from the cached property values retrieved from the default property list on library initialization, to avoid performance impact from interacting with the property list code. Any changes to a cached field are **not** reflected back into the original property list. Cached fields have a particular naming scheme: the struct member(s) that holds the field value in the H5CX_t struct must be paired with a boolean field that is named "<field name>_valid" for the macros that interact with the property to work correctly. Cached fields may have both

a ‘get’ and ‘set’ routine, or just a ‘get’ routine, if they are never set within the library.

- Return-Only Fields: Return-only fields in the API context correspond to a property in the DXPL or LAPL, but are never retrieved within the library and are only set with values to return to the application. Values set for these fields are cached within the API context until it is being destroyed (when leaving an API return) and then any changed field values are copied back into the corresponding property in the DXPL or LAPL. Return-only fields have a particular naming scheme as well: the struct member(s) that holds the field value in the H5CX_t struct must be paired with a boolean field that is named “<field name>_valid” for the macros that interact with the property to work correctly. Return-only fields may have both a ‘get’ and ‘set’ routine, or just a ‘set’ routine, if they are never retrieved within the library.

3.6 Adding New API Context Fields

Adding a new field to one of the types of API context fields involves several sections of the H5CX code, outlined for each kind of field:

- Reference Property Lists: Adding a new reference property list to the API context struct is straightforward: a new member within the H5CX_t struct should be defined to hold the property list ID and a generic property list pointer (H5P_genplist_t *) can also be added, if properties from that property list will correspond to new cached or return-only fields. Library-private ‘get’, ‘set’, or other support routines should be defined for setting, retrieving or interacting with non-default property list values. If properties from the new property list will be held in cached fields, a new “cache” struct should be defined to hold them, and those properties should be retrieved in the H5CX package initialization routine.
- Internal Fields: New internal fields are easily added to the API context struct, since they don’t have any correlation to properties from a property list, nor any required naming scheme. Simply add the new field to the H5CX_t struct and define accessor routines for the new field.
- Cached Fields: To add a new cached field to the API context, two (or more) members must be added to the H5CX_t struct: one (or more) to hold the actual field value(s), and another member named “<field name>_valid” that indicates that the field value has been retrieved from the corresponding property in a property list (or set internally to the library). One (or more) field must also be added to the corresponding “cache” struct, which should be identically named and typed as the field(s) added to the H5CX_t struct for the field value, and the default property for the new field must be retrieved into the cache struct in the package initialization routine. New ‘get’ (and possibly ‘set’) routines should be added for the new field, mimicing the existing routines for cached fields.
- Return-Only Fields: To add a new return-only field to the API context, two (or more) members must be added to the H5CX_t struct: one (or more) to hold the actual field value(s), and another member named “<field name>_set” that indicates that the field value has been set and should be returned to the application. These fields are generally not cached from the default property list, so the “cache” struct should not need to be modified. New ‘set’ (and possibly ‘get’) routines should be added for the new field, mimicing the existing routines for return-only fields.

3.7 Parallel HDF5

Parallel HDF5 uses API contexts in the same way as serial HDF5 and is not affected by them. The API context is used for collecting info about collective operations within the main HDF5 library and passing it to the MPI-IO VFD, but that's a strictly local operation, without calls to other MPI ranks.

3.8 Testing

The API context framework sometimes requires special handling within the library's regression tests. If a regression test only uses public HDF5 API calls, then no special actions are needed, as the library will create and destroy API contexts when entering and exiting those API routines. If a regression test calls a 'testing' routine within the library, that testing routine is not internally bracketed by FUNC_ENTER_API / FUNC_LEAVE_API macros, so it must push and pop an API context explicitly. Other "white box" library regression tests call internal library routines directly, and must include calls to H5CX_push / H5CX_pop within the test itself. If an API context is not created and pushed onto the context stack, the H5CX code will attempt to detect this and core dump when the library is built in "debug" mode.

3.9 Re-entrancy

The API context implementation is designed to handle re-entering the HDF5 API though application callbacks. Each entry into the library through an API routine will create a new API context, pushing it onto the context stack. Information from one context is isolated from changes in another context.

3.10 Thread-safety

The API context implementation is designed to isolate information in contexts in one thread of execution from other threads. Each thread has its own thread-local API context stack, set up similarly to the other thread-safe interfaces in the library (error stacks and function stacks). When entering a 'get' / 'set' routine in the H5CX interface, the head of the thread-local context stack is retrieved with the H5CX_get_my_context macro (defined in src/H5CX.c), and then the context is accessed through that pointer.

4 Summary

API contexts provide a framework for tying distant sections of the HDF5 library together, using a high-performance, thread-safe, and easy-to-understand implementation that is a significant improvement over previous methods of performing similar tasks.

Acknowledgements

This work was supported by [grant, contract, cooperative agreement] number [agreement number] from the U.S. Department of Energy (DOE). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author[s] and do not necessarily reflect the views of the Department of Energy.

Revision History

March 27, 2018: Version 1 circulated for public comment. Comments should be sent to koziol@lbl.gov.