RFC: Terminal VOL Connector Feature Flags

Dana Robinson

Jordan Henderson

The HDF5 Virtual Object Layer (VOL) provides a powerful abstraction mechanism for mapping HDF5 API calls to arbitrary storage schemes. Not all terminal VOL connectors implement all HDF5 API calls, however, and no fully-implemented scheme exists that can be used to determine if a VOL connector meets an application's needs.

This work proposes a set of VOL connector feature flags that can be used for this purpose.

# Introduction

A set of feature flags that describes which HDF5 capabilities a VOL connector implements would solve two problems:

* Matching HDF5 applications to suitable VOL connectors
* Indicating which tests in the vol-test repository suite should be run for a VOL connector

The first problem is critical – Since there is no standard for what makes an "acceptable" VOL connector, HDF5 software that uses non-native VOL connectors will need to be able to query connectors to see if they exhibit the correct functionality.

# Existing/Related Infrastructure

The H5VL\_class\_t struct already has a cap\_flags field (type: unsigned). The flags that are currently supported are:

#define H5VL\_CAP\_FLAG\_NONE 0 /\* No special connector capabilities \*/

#define H5VL\_CAP\_FLAG\_THREADSAFE 0x01 /\* Connector is threadsafe \*/

#define H5VL\_CAP\_FLAG\_ASYNC 0x02 /\* Connector performs operations asynchronously\*/

#define H5VL\_CAP\_FLAG\_NATIVE\_FILES 0x04 /\* Connector produces native file format \*/

An application can get a VOL connector's flags using several API calls:

* H5Pget\_vol\_cap\_flags (H5Ppublic.h)
* H5VLget\_cap\_flags (H5VLconnector\_passthrough.h)
* H5VLintrospect\_get\_cap\_flags (H5VLconnector\_passthrough.h)

H5Pget\_vol\_cap\_flags() is the ideal way for an application or test suite to obtain the flags that will be in effect when a file is opened using that fapl as it takes the full VOL stack into account.

herr\_t H5Pget\_vol\_cap\_flags(hid\_t plist\_id, unsigned \*cap\_flags)

H5VLget\_cap\_flags() is in the public API call but is not considered to be a user-level API call and is instead intended for use in pass-through VOL connectors. Since it takes a connector ID instead of a file access property list, it does not consider the entire VOL stack and is unsuitable for use in applications.

herr\_t H5VLget\_cap\_flags(hid\_t connector\_id, unsigned \*cap\_flags)

H5VLintrospect\_get\_cap\_flags() is also in the public API call but is not considered to be a user-level API call and is instead intended for use in pass-through VOL connectors. This call uses the VOL stack (and is effectively what H5Pget\_vol\_cap\_flags() will eventually call) but the info parameter is not normally available to applications, so this call is unsuitable for use in applications.

herr\_t H5VLintrospect\_get\_cap\_flags(const void \*info,

hid\_t connector\_id, unsigned \*cap\_flags)

# Coarse- or Fine-Grained Flags?

The first topic we need to address is how fine-grained we want the feature flags to be. At the most fine-grained, there would be one feature flag for most API calls as well as some broader flags like "creates native HDF5 files". Such a high level of detail would allow very precise mapping of software to VOL connectors. While applications may not need this level of precision, it would be handy for the VOL tests, which could definitely make use of it. There are several downsides, though. The first is that this would be a lot of work to implement, not only at the library and VOL connector levels, but also for applications, which would have to specify complicated compatibility flags. Another is that we'd easily exceed the 64 flags we can pack into the largest common unsigned integer type, requiring a more complicated flag structure and more extensive changes to existing VOL connectors and applications.

An alternative is to use a more coarse-grained scheme, where flags specify larger chunks of functionality. This would be much simpler to implement, the flags would likely fit into a single integer type, and could (mostly) use existing infrastructure. The downside is that applications/tests will not precisely map their desired functionality to VOL connectors.

# The Problem of Getting the Flags in the First Place

One problem that we have with VOL connector flags, is that getting them has some complications.

* VOL connectors can be stacked
* Some VOL features are dependent on configuration settings
* Some VOL features may not be known until runtime

This means that an application can’t simply query a VOL connector via its hid\_t ID for its properties, but must instead query the file access property list that will be used to open the file, as was noted in the discussion about existing functionality. The only call in the non-connector-author API, though, does query the fapl.

Another consideration is that any connector properties that are only discoverable at runtime would probably require a new API call that queried the file. At this time, we consider this a lower-order concern and and will not implement a new call.

# A Proposed Flag Scheme

We're going to start with a more coarse-grained capability flag system. This is mainly in the interest of simplicity, both in VOL implementation details and in the amount of code applications will have to write. If a more complicated, fine-grained system proves necessary, we'll cross that bridge when we come to it.

The proposed scheme simply piggybacks on the existing capabilities flags, albeit with a slight change to the flags type and moving some components around.

* The cap\_flags field will be changed to a uint64\_t from an unsigned int to increase the number of flags from 32 to 64 and so the number of flags is explicit and not system-dependent[[1]](#footnote-1).
* The associated capabilities flags API calls will modified to take uint64\_t parameters instead of unsigned integers.
* The existing capabilities flags will be moved from H5VLconnector.h to H5VLpublic.h.
* The H5VL\_class\_t version will be bumped by this change (this change can be combined with the multi-dataset VOL changes).

In this proposed flag scheme, we're assuming that the number of capabilities flags will not exceed 64. Given that we're only using half of our bits to cover the entire HDF API, this seems like a reasonable assumption. If a more fine-grained solution that multiples the flags proves necessary, we're probably going to have to rework the flags field into something more complicated anyway.

## Proposed Compatibility Flags

Keeping with the existing naming scheme, flags will be named H5VL\_CAP\_FLAG\_<THING> according to the following table.

|  |  |
| --- | --- |
| **<THING>** | **Description** |
| ATTRIBUTE\_BASIC | H5Acreate(\_by\_name)/delete(\_by\_name)/exists(\_by\_name)/  open/close/read/write |
| ATTRIBUTE\_MORE | H5Aget\_info(\_by\_name)/get\_name/get\_space/get\_type/  rename(\_by\_name) |
| DATASET\_BASIC | H5Dcreate/open/close/read/write |
| DATASET\_MORE | H5Dget\_space(\_status)/get\_type/set\_extent |
| FILE\_BASIC | H5Fcreate/open/close/is\_accessible/delete |
| FILE\_MORE |  |
| GROUP\_BASIC | H5Gcreate/open/close |
| GROUP\_MORE | H5Gget\_info |
| LINK\_BASIC | H5Lexists/delete |
| LINK\_MORE | H5Lcopy/move/get\_info/get\_name/get\_val |
| OBJECT\_BASIC | H5Oopen/close/exists |
| OBJECT\_MORE | H5Ocopy/get\_file/get\_name/get\_type/get\_info/incr|decr\_refcount |
| REFERENCES\_BASIC | H5Rdestroy, at least one of the OBJ|REG|ATTR\_REF flags |
| REFERENCES\_MORE | H5Rget\_type/ copy/get\_file\_name |
| OBJ\_REF | H5Rcreate\_object/open\_object |
| REG\_REF | H5Rcreate\_region/open\_region |
| ATTR\_REF | H5Rcreate\_attr/open\_attr |
| STORED\_DATATYPES | H5Tcommit/open |
| CREATION\_ORDER | H5Pset\_(attr|link)\_creation\_order (as applied to other packages) |
| ITERATE | H5Aiterate, H5Lvisit, et al.  (Specific calls depend on package flags) |
| STORAGE\_SIZE | H5Aget\_storage\_size, et al.  (Specific calls depend on package flags) |
| BY\_IDX | H5Adelete\_by\_idx/get\_info\_by\_idx/get\_name\_by\_idx, et al.  (Specific calls depend on package flags) |
| GET\_PLIST | H5Aget\_create\_plist, et al.  (Specific calls depend on package flags) |
| FLUSH\_REFRESH | H5Dflush/refresh, H5Gflush/refresh |
| EXTERNAL\_LINKS | H5Lcreate\_external |
| HARD\_LINKS | H5Lcreate\_hard |
| SOFT\_LINKS | H5Lcreate\_soft |
| UD\_LINKS | H5Lcreate\_ud |
| TRACK\_TIMES | H5Pset\_obj\_track\_times (as applied to other packages) |
| MOUNT | H5F(un)mount, H5G(un)mount |
| FILTERS | Filter pipelines |
| FILL VALUES | Supports dataset fill values |

It might also be useful to define useful bitwise OR flag sets. For example, a BASIC flag set might be used to indicate that a VOL connector could serve as a general purpose VOL connector in common use cases, such as running IOR.

# Use By Applications / VOL Test Suite

Utilizing the capabilities flags will be straightforward.

1. Create a desired flag set by combining the capabilities you need using bitwise OR
2. Create a fapl and set up whatever VOL connectors you need
3. Get the current VOL connector's capability flags from the fapl you created
4. Compare required flags with bitwise AND of required and VOL flags

uint64\_t req\_flags = H5VL\_CAP\_FLAG\_BASIC | H5VL\_CAP\_FLAG\_FILTERS;

uint64\_t vol\_flags = H5VL\_CAP\_FLAG\_NONE;

hid\_t fapl\_id = H5I\_INVALID\_HID;

fapl\_id = H5Pcreate(H5P\_FILE\_ACCESS);

/\* Set VOL connector(s) on fapl here \*/

H5Pget\_cap\_flags(fapl\_id, &vol\_flags);

if (req\_flags != (req\_flags & vol\_flags))

exit(EXIT\_FAILURE); /\* or skip tests… \*/

# Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

# Revision History

|  |  |
| --- | --- |
| *August 19, 2022:* | Version 1 circulated for comment within The HDF Group. |
| *September 15, 2022* | Version 2 adds information about H5Pget\_vol\_cap\_flags() and updates the section on checking capabilities flags. |

1. Even though ILP64 and SILP64 systems are rare and it's been decades since 16-bit integers were common. [↑](#footnote-ref-1)