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 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.

1 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.

2 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 `H5VLintrospect_get_cap_flags()`. This is considered a "pass-through VOL connector" API call vs. a user-level API call and is declared in `H5VLconnector_passthru.h`.

3 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.

4 The Problem of Getting the Flags in the First Place

One problem that we have with VOL connector flags, is that getting them is complicated.

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

The existing feature flags scheme is acceptable for a passthrough VOL connector querying the underlying connectors, but is not particularly well suited to applications trying to figure out if a connector stack meets the needs of the software.

There are two reasons for this:

1. API call parameters are more suited for pass-through connectors than applications. For example, `H5VLintrospect_get_cap_flags()` takes a void pointer for the info struct.
2. Situational. The existing scheme was designed for passthrough connectors that are in the midst of opening an object and have access to the object, the fapl, and know about the underlying VOL connector.

These are basically fixable problems, but fully addressing the second may involve advising applications to check a connector's flags using either a fapl or connector ID to see if it's even possible for a VOL connector to meet the application's needs (connectors would have to be configured to emit "best possible" flags when queried directly), then check the flags again after file create/open.

5 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].
- 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 `H5VLget_cap_flags()` prototype will move from `H5VLconnector_passthru.h` to `H5VLpublic.h`.
- We'll investigate modifying `H5VLget_cap_flags()` to take a fapl with a VOL stack instead of a VOL connector ID. It will also be modified to use the introspect callback instead of just returning the top-level connector's flags.
- 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.

5.1 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`	

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

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_ref count
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.

6 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. Get the current VOL connector's capability flags
3. 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;

H5VLget_cap_flags(plist_id /* or open object */, &vol_flags);

if (req_flags != (req_flags & vol_flags))
      exit(EXIT_FAILURE); /* or skip tests… */
```

As mentioned earlier in this document, the difficulty here is with passthrough connectors.

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.