HDF5 Virtual Object Layer (VOL) Connector Author Guide

HDF5 1.12.0

The HDF Group

5th December 2019

Contents

1 Introduction **1**

2 Creating a New Connector **2**
2.1 Overview . . . 2
2.2 Using The New Connector . . . 2
2.3 Testing . . . 3
2.4 Internal Connectors . . . 3
2.4.1 Implementing an Internal Connector . . . 3
2.4.2 Using an Internal Connector . . . 4

3 VOL Connector Interface Reference **5**
3.1 Mapping the API to the Callbacks . . . 6
3.2 Connector Information Callbacks . . . 8
3.2.1 info: size . . . 8
3.2.2 info: copy . . . 8
3.2.3 info: cmp . . . 9
3.2.4 info: free . . . 9
3.2.5 info: to_str . . . 9
3.2.6 info: from_str . . . 9
3.3 Object Wrap Callbacks . . . 10
3.3.1 wrap: get_object . . . 10
3.3.2 wrap: get_wrap_ctx . . . 10
3.3.3 wrap: wrap_object . . . 10
3.3.4 wrap: unwrap_object . . . 10
3.3.5 wrap: free_wrap_ctx . . . 11
3.4 The Attribute Function Callbacks . . . 11
3.4.1 attr: create . . . 11
3.4.2 attr: open . . . 12
3.4.3 attr: read . . . 12
3.4.4 attr: write . . . 12
3.4.5 attr: get . . . 13
3.4.6 attr: specific . . . 14
3.4.7 attr: optional . . . 15
3.4.8 attr: close . . . 15
3.5 Dataset Callbacks . . . 15
3.5.1 dataset: create . . . 16
3.5.2 dataset: open . . . 16
3.5.3 dataset: read . . . 17
3.5.4 dataset: write . . . 17
3.5.5 dataset: get . . . 17
3.5.6 dataset: specific . . . 18
3.5.7 dataset: optional . . . 19
3.5.8 dataset: close . . . 19
3.6 Datatype Callbacks . . . 20
3.6.1 datatype: commit . . . 20
3.6.2 datatype: open . . . 21
3.6.3 datatype: get . . . 21
3.6.4 datatype: specific . . . 22
3.6.5 datatype: optional . . . 23
3.6.6 datatype: close . . . 23
3.7 File Callbacks . . . 23
3.7.1 file: create . . . 24
3.7.2 file: open . . . 24
3.7.3 file: get . . . 24
3.7.4 file: specific . . . 25
3.7.5 file: optional . . . 27
3.7.6 file: close . . . 27
3.8 Group Callbacks . . . 27
3.8.1 group: create . . . 28

3.8.2 group: open . . . 28
3.8.3 group: get . . . 28
3.8.4 group: specific . . . 29
3.8.5 group: optional . . . 30
3.8.6 group: close . . . 30
3.9 Link Callbacks . . . 30
3.9.1 link: create . . . 31
3.9.2 link: copy . . . 31
3.9.3 link: move . . . 32
3.9.4 link: get . . . 32
3.9.5 link: specific . . . 33
3.9.6 link: optional . . . 34
3.10 Object Callbacks . . . 34
3.10.1 object: open . . . 35
3.10.2 object: copy . . . 35
3.10.3 object: get . . . 36
3.10.4 object: specific . . . 36
3.10.5 object: optional . . . 37
3.11 Request (Async) Callbacks . . . 38
3.11.1 request: wait . . . 38
3.11.2 request: notify . . . 38
3.11.3 request: cancel . . . 39
3.11.4 request: specific . . . 39
3.11.5 request: optional . . . 39
3.11.6 request: free . . . 39
3.12 Blob Callbacks . . . 40
3.12.1 blob: put . . . 40
3.12.2 blob: get . . . 40
3.12.3 blob: specific . . . 40
3.12.4 blob: optional . . . 41
3.13 Optional Generic Callback . . . 41

4 New VOL API Routines 41
4.1 H5VLpublic.h . . . 42
4.1.1 H5VLregister_connector . . . 42
4.1.2 H5VLregister_connector_by_name . . . 42
4.1.3 H5VLregister_connector_by_value . . . 42
4.1.4 H5VLis_connector_registered . . . 42
4.1.5 H5VLget_connector_id . . . 43
4.1.6 H5VLget_connector_name . . . 43
4.1.7 H5VLclose . . . 43
4.1.8 H5VLunregister_connector . . . 43
4.2 H5VLconnector.h . . . 44
4.2.1 H5VLobject_register . . . 44
4.2.2 H5VLget_object . . . 44

Appendix A Mapping of VOL Callbacks to HDF5 API Calls 45

Appendix B Callback Wrapper API Calls for Passthrough Connector Authors 48

1 Introduction

The Virtual Object Layer (VOL) is an abstraction layer in the HDF5 library which intercepts all API calls that could potentially access objects in an HDF5 container and forwards those calls to object drivers referred to as *VOL connectors*. The architecture of this feature is described in the VOL User Guide and VOL Architecture and Internals Documentation and will not be duplicated here.

This guide is for people who are interested in developing their own VOL connector for the HDF5 library. It is assumed that the reader has good knowledge of the VOL architecture obtained by reading the VOL architectural design document [**?**].

2 Creating a New Connector

External plugins are developed outside of the HDF5 library and do not use any internal HDF5 private functions. They do not require to be shipped with the HDF5 library, but can just link to it from userspace just like an HDF5 application. If an external VOL plugin is installed on the system as a shared library or a DLL, the library can search for it and load it dynamically to be used by an application.

2.1 Overview

Developing an external plugin is similar to developing an internal plugin. Refer to Section 3 for details on plugin creation. The important thing to keep in mind is that external plugins cannot use internal HDF5 features. The "value" field for an external plugin should be a positive integer greater than 128. The name should be unique across all plugins registered with the library.

For the HDF5 library to be able to load an external plugin dynamically, the plugin developer has to define two public routines with the following name and signature:

```
H5PL_type_t H5PLget_plugin_type(void)
const void *H5PLget_plugin_info(void)
```

`H5PLget_plugin_type` should return the library type which should always be `H5PL_TYPE_VOL`. `H5PLget_plugin_info` should return a pointer to the plugin structure defining the VOL plugin with all the callbacks. For example, consider an external plugin defined as:

```
static const H5VL_class_t H5VL_log_g = {
    1,  /* version */
    502,       /* value */
    "log",     /* name */
    ...
}
```

The plugin would implement the two routines as:

```
H5PL_type_t H5PLget_plugin_type(void) {return H5PL_TYPE_VOL;}
const void *H5PLget_plugin_info(void) {return &H5VL_log_g;}
```

Implementing iteration and visit callbacks in external plugins requires the plugin invoking the user defined callback function with an HDF5 identifier (`hid_t`) on the location object. The current HDF5 H5I API does not allow creating and dereferencing identifiers that point to HDF5 object types. Therefore, two new API routines have been added (see Section 4 for more details) to get around that. `H5VLobject_register()` is used to create an HDF5 identifier for a VOL object handle (file, group, dataset, named datatype, or attribute). `H5VLget_object()` is used to derefernce the HDF5 identifier back to the VOL object handle.

2.2 Using The New Connector

Unlike internal plugins, external plugins cannot create an API routine for applications to use to set the VOL plugin in the file access property list. External plugin developers can, however, provide a wrapper routine that registers the plugin with `H5VLregister()` instead of having the application call that directly with the pointer to the plugin class structure. Both ways are possible if the external plugin used is in the application's userspace. External plugins provided by third party vendors that are not in the application's userspace but are installed on the system with a shared library or DLL can be registered by the application using `H5VLregister_name()` provided that the application knows the plugin's name in advance.

The registration operation will return a global identifier for the registered plugin. Applications can query the library for that identifier using the plugin name with `H5VLget_plugin_id()`. This identifier is used to initialize the plugin if it requires an initialization phase. A third party library could use that ID to create a plugin that stacks or mirrors on top of an internal plugin using that identifier (refer to Section **??** for more details on stacking and mirroring).

Some plugins could require an initialization phase before using the plugin with some input parameters from users. This process is done by creating a VOL initialization property list and setting a public property that should be defined by the internal plugin to the hostname of the remote machine. The user then calls the initialize routine (`H5VLintialize()`) with the plugin_id and the property list to initialize the plugin. An external plugin could abstract those operations into a wrapper routine that wraps the property list creation, property settings, and plugin initialization into one call that the user makes to simplify this process.

The application then sets the plugin access property in the file access property list to the external plugin using the plugin identifier with this API routine:

```
herr_t H5Pset_vol(hid_t fapl_id, hid_t plugin_id, const void *new_vol_info);
```

where `new_vol_info` is the plugin information needed from the application. Typically it is a plugin defined structure with some fields to setup the plugin access. The external plugin could also provide a wrapper function around this to make it easier for the application to use.

Now that the plugin is initialized, the application can create or open an HDF5 container using the external plugin. After closing all objects and containers on that plugin, the application should terminate access to the selected plugin if it requires a termination phase by calling `H5VLterminate()` with the required termination properties (similar to the initialization phase).

Finally, the application is required to un-register the plugin from the library when access to the container(s) is terminated using `H5VLunregister`.

2.3 Testing

2.4 Internal Connectors

Internal plugins are developed internally with the HDF5 library and are required to ship with the entire library to be used. Typically those plugins need to use internal features and functions of the HDF5 library that are not available publicly from the user application.

2.4.1 Implementing an Internal Connector

The first step to implement an internal plugin is to implement all the callbacks defined in the VOL class as described in Section 3. After implementing the VOL class, the next step would be to allow users to select this plugin to be used. This is done by creating a new API routine to set the plugin on the file access property list. For example, if we create an internal plugin called "dummy" that needs an MPI communicator and info object as information from the user, that routine signature should look like:

```
herr_t H5Pset_fapl_dummy(hid_t fapl_id, MPI_Comm comm, MPI_Info info);
```

The implementation for the above routine should use the internal function:

```
herr_t H5P_set_vol(H5P_genplist_t *plist, H5VL_class_t *vol_cls, const
void *vol_info);
```

that will set the file access using that `fapl_id` to go through the "dummy" plugin. It will also call the copy callback of the "dummy" plugin on the info object (`comm and info`).

A sample implementation for the `H5Pset_fapl_dummy()` would look like this:

```
/* DUMMY-specific file access properties */
typedef struct H5VL_dummy_fapl_t {
    MPI_Comm          comm;    /* communicator     */
    MPI_Info          info;    /* MPI information */
} H5VL_dummy_fapl_t;

herr_t
H5Pset_fapl_dummy(hid_t fapl_id, MPI_Comm comm, MPI_Info info)
{
```

```
10      H5VL_dummy_fapl_t fa;
11      H5P_genplist_t *plist;    /* Property list pointer */
12      herr_t         ret_value;
13
14      FUNC_ENTER_API(FAIL)
15
16      if(fapl_id == H5P_DEFAULT)
17          HGOTO_ERROR(H5E_PLIST, H5E_BADVALUE, FAIL, "can't set values in default property list")
18
19      if(NULL == (plist = H5P_object_verify(fapl_id, H5P_FILE_ACCESS)))
20          HGOTO_ERROR(H5E_ARGS, H5E_BADTYPE, FAIL, "not a file access property list")
21
22      if(MPI_COMM_NULL == comm)
23          HGOTO_ERROR(H5E_PLIST, H5E_BADTYPE, FAIL, "not a valid communicator")
24
25      /* Initialize driver specific properties */
26      fa.comm = comm;
27      fa.info = info;
28
29      ret_value = H5P_set_vol(plist, &H5VL_dummy_g, &fa);
30
31  done:
32      FUNC_LEAVE_API(ret_value)
33  } /* end H5Pset_fapl_dummy() */
```

At this point, the internal plugin is ready to be used. For more information on how to implement an internal plugin, the native plugin for the HDF5 library is a comprehensive plugin that implements all features of the library and can be used as a guide.

2.4.2 Using an Internal Connector

An internal plugin is registered automatically with the HDF5 library when the library gets initialized. Users should not attempt to register an internal plugin. The internal registration operation creates a global identifier for each internal plugin. Applications can query the library for that identifier using the plugin name with `H5VLget_plugin_id()`. This identifier is used to initialize the plugin if it requires an initialization phase. A third party library could use that ID to create a plugin that stacks or mirrors on top of an internal plugin using that identifier (refer to Section **??** for more details on stacking and mirroring).

Some plugins could require an initialization phase before using the plugin with some input parameters from users. For example, a remote plugin would need to setup a connection to a remote machine with a specific hostname that the user knows and the HDF5 library does not know about. This process is done by creating a VOL initialization property list and setting a public property that should be defined by the internal plugin to the hostname of the remote machine. The user then calls the initialize routine (`H5VLintialize()`) with the plugin_id and the property list to initialize the plugin. Some plugins could create a public API routine that wraps the property list creation, property settings, and plugin initialization into one routine that the user calls to simplify this process for the users. The native HDF5 plugin does not require an initialization phase.

Now that the plugin is initialized, the application can create or open an HDF5 container using the internal plugin by creating a file access property list and calling the corresponding API routine that the internal plugin should define for setting the VOL plugin property and container parameters on the file access property list. That FAPL can be passed to the `H5Fcreate()` or `H5Fopen` calls to access the container through the VOL plugin. Subsequently, all other HDF5 calls on the container and objects created or opened in that container will automatically go to the selected VOL plugin.

After closing all objects and containers on that plugin, the application should terminate access to the selected plugin if it requires a termination phase by calling `H5VLterminate()` with the required termination properties (similar to the initialization phase). The native HDF5 plugin does not require a termination phase.

The internal plugin is unregistered automatically by the HDF5 library on library termination, and users should not attempt to unregister it themselves.

3 VOL Connector Interface Reference

Each VOL connector should be of type `H5VL_class_t`, Listing 1.

```
/* Class information for each VOL driver */
typedef struct H5VL_class_t {
    /* Overall connector fields & callbacks */
    unsigned int version;               /* VOL connector class struct version # */
    H5VL_class_value_t value;           /* Value to identify connector          */
    const char *name;                   /* Connector name (MUST be unique!)     */
    unsigned cap_flags;                 /* Capability flags for connector       */
    herr_t (*initialize)(hid_t vipl_id); /* Connector initialization callback   */
    herr_t (*terminate)(void);          /* Connector termination callback       */

    /* VOL framework */
    H5VL_info_class_t     info_cls;     /* VOL info fields & callbacks */
    H5VL_wrap_class_t     wrap_cls;     /* VOL object wrap / retrieval callbacks */

    /* Data Model */
    H5VL_attr_class_t     attr_cls;     /* Attribute (H5A*) class callbacks */
    H5VL_dataset_class_t  dataset_cls;  /* Dataset (H5D*) class callbacks */
    H5VL_datatype_class_t datatype_cls; /* Datatype (H5T*) class callbacks */
    H5VL_file_class_t     file_cls;     /* File (H5F*) class callbacks   */
    H5VL_group_class_t    group_cls;    /* Group (H5G*) class callbacks  */
    H5VL_link_class_t     link_cls;     /* Link (H5L*) class callbacks   */
    H5VL_object_class_t   object_cls;   /* Object (H5O*) class callbacks */

    /* Infrastructure / Services */
    H5VL_request_class_t  request_cls;  /* Asynchronous request class callbacks */
    H5VL_blob_class_t     blob_cls;     /* 'blob' callbacks */

    /* Catch-all */
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments); /* Optional
        callback */
} H5VL_class_t;
```

Listing 1: VOL connector class, H5VLpublic.h

The `version` field is the version of the `H5VL_class_t` struct. This is identical to how the `version` field is used in the `H5Z_class2_t` struct for filters.

The `value` field is a unique integer identifier that should be greater than 256 for external, non-library connectors. Setting it in the VOL structure is required.

The `name` field is a string that uniquely identifies the VOL connector name. Setting it in the VOL structure is required.

The `cap_flags` field will be used to hold bitwise capability/feature flags that can be used with a future query API call to determine which operations and capabilities are supported by a the VOL connector. This feature is currently under development.

The `initialize` field is a function pointer to a routine that a connector implements to set up or initialize access to the connector. Implementing this function by the connector is not required since some connectors do not require any set up to start accessing the connector. In that case, the value of the function pointer should be set to NULL. Connector specific variables that are required to be passed from users should be passed through the VOL initialize property list. Generic properties can be added to this property class for user-defined connectors that cannot modify the HDF5 library to add internal properties. For more information consult the property list reference manual pages.

The `terminate` field is a function pointer to a routine that a connector implements to terminate or finalize access to the connector. Implementing this function by the connector is not required since some connectors do not require any termination phase to the connector. In that case, the value of the function pointer should be set to NULL. Connector specific variables that are required to be passed from users should be passed

through the VOL terminate property list. Generic properties can be added to this property class for user-defined connectors that cannot modify the HDF5 library to add internal properties. For more information consult the property list reference manual pages.

The rest of the fields in the `H5VL_class_t` struct are "subclasses" that define all the VOL function callbacks that are mapped to from the HDF5 API layer. Those subclasses are categorized into two categories, Data Model and Services. Data Model classes are those that provide functionality for accessing an HDF5 container and objects in that container as defined by the HDF5 data model. Service classes are those that provide services for users that are not related to the data model specifically. Asynchronous operations, for example, are a service that most connectors can implement, so we add a class for it in the VOL structure. If a service becomes generic enough and common among many connectors, a class for it should be added to the VOL structure. However, many connectors can/will provide services that are not shared by other connectors. A good way to support these services is through an optional callback in the VOL structure which can be a hook from the API to the connector that provides those services, passing any necessary arguments needed without the HDF5 library having to worry about supporting that service. A similar API operation to allow users to use that service will be added. This API call would be similar to an "ioctl" call where any kind of operation can be supported and passed down to the connector that has enough knowledge from the user to interpret the type of the operation. All classes and their defined callbacks will be detailed in the following sub-sections.

3.1 Mapping the API to the Callbacks

The callback interface defined for the VOL has to be general enough to handle all the HDF5 API operations that would access the file. Furthermore, it has to capture future additions to the HDF5 library with little to no changes to the callback interface. Changing the interface often whenever new features are added would be discouraging to connector developers since that would mean reworking their VOL connector structure. To remedy this issue, every callback will contain two parameters:

- A data transfer property list (DXPL) which allows that API to put some properties on for the connectors to retrieve if they have to for particular operations, without having to add arguments to the VOL callback function.
- A pointer to a request (`void **req`) to handle asynchronous operations if the HDF5 library adds support for them in future releases (beyond the 1.8 series). That pointer is set by the VOL connector to a request object it creates to manage progress on that asynchronous operation. If the `req` is `NULL`, that means that the API operation is blocking and so the connector would not execute the operation asynchronously. If the connector does not support asynchronous operations, it needs not to worry about this field and leaves it unset.

In order to keep the number of the VOL object classes and callbacks concise and readable, it was decided not to have a one-to-one mapping between API operation and callbacks. Furthermore, to keep the callbacks themselves short and not cluttered with a lot of parameters, some of the parameters are passed in as properties in property lists included with the callback. The value of those properties can be retrieved by calling the public routine (or its private version if this is an internal connector):

```
1 herr_t H5Pget(hid_t plist_id, const char *property_name, void *value);
```

The property names and value types will be detailed when describing each callback in their respective sections.

The HDF5 library provides several routines to access an object in the container. For example, to open an attribute on a group object, the user could use `H5Aopen()` and pass the group identifier directly where the attribute needs to be opened. Alternatively, the user could use `H5Aopen_by_name()` or `H5Aopen_by_idx()` to open the attribute, which provides a more flexible way of locating the attribute, whether by a starting object location and a path or an index type and traversal order. All those types of accesses usually map to one VOL callback with a parameter that indicates the access type. In the example of opening an attribute, the three API open routine will map to the same VOL open callback but with a different location parameter. The same applies to all types of routines that have multiple types of accesses. The location parameter is a structure defined in Listing 2.

```c
/*
 * Structure to hold parameters for object locations.
 * either: BY_SELF, BY_NAME, BY_IDX, BY_TOKEN
 */

typedef struct H5VL_loc_params_t {
    H5I_type_t obj_type; /* The object type of the location object */
    H5VL_loc_type_t type; /* The location type */
    union { /* parameters of the location */
        H5VL_loc_by_token_t   loc_by_token;
        H5VL_loc_by_name_t    loc_by_name;
        H5VL_loc_by_idx_t     loc_by_idx;
    }loc_data;
} H5VL_loc_params_t

/*
 * Types for different ways that objects are located in an
 * HDF5 container.
 */
typedef enum H5VL_loc_type_t {
    /* starting location is the target object */
    H5VL_OBJECT_BY_SELF,

    /* location defined by object and path in H5VL_loc_by_name_t */
    H5VL_OBJECT_BY_NAME,

    /* location defined by object, path, and index in H5VL_loc_by_idx_t */
    H5VL_OBJECT_BY_IDX,

    /* location defined by token (e.g. physical address) in H5VL_loc_by_token_t */
    H5VL_OBJECT_BY_TOKEN,
} H5VL_loc_type_t;

typedef struct H5VL_loc_by_name {
    const char *name; /* The path relative to the starting location */
    hid_t lapl_id; /* The link access property list */
}H5VL_loc_by_name_t;

typedef struct H5VL_loc_by_idx {
    const char *name; /* The path relative to the starting location */
    H5_index_t idx_type; /* Type of index */
    H5_iter_order_t order; /* Index traversal order */
    hsize_t n; /* Position in index */
    hid_t lapl_id; /* The link access property list */
}H5VL_loc_by_idx_t;

typedef struct H5VL_loc_by_token {
    void *token; /* arbitrary token (physical address of location in native VOL) */
}H5VL_loc_by_token_t;
```

Listing 2: Structure to hold parameters for object locations, H5VLpublic.h

Other API routines that would make a one-to-one mapping difficult are:

- the `Get` operations that retrieve something from an object; for example a property list or a datatype of a dataset, etc.
- API routines are specific to each class of callbacks (attributes, files, etc...). Having a callback for each one would clutter the VOL callback structure and reduce user-friendliness.
- API routines that make sense to only implement in the native HDF5 file format and would not be possible to implement in other connectors. Adding a callback for those routines would be confusing to connector developers.

To handle that large set of API routines, each class in the Data Model category has three generic callbacks, `get`, `specific`, and `optional` to handle the three set of API operations outline above respectively. The callbacks will have a `va_list` argument to handle the different set of parameters that could be passed in. The `get` and `specific` callbacks also have an `op_type` that contain an `enum` of the type of operation that is requested to be performed. Using that type, the `va_list` argument can be parsed. The `optional` callback is a free for all callback where anything from the API layer is passed in directly. This callback is used to support connector specific operations in the API that other connectors should or would not know about. More information about types and the arguments for each type will be detailed in the corresponding class arguments.

3.2 Connector Information Callbacks

This section's callbacks involve the connector-specific information that will be associated with the VOL in the fapl via `H5Pset_fapl()` et al. This data is copied into the fapl so the library needs these functions to manage this in a way that prevents resource leaks.

The `to_str` and `from_str` callbacks are used to convert the connector-specific data to and from a configuration string. There is no official way to construct VOL configuration strings, so the format used (JSON, XML, getopt-style processing, etc.) is up to the connector author. These connector configuration strings can be used to set up a VOL connector via mechanisms like command-line parameters and environment variables.

```
1 typedef struct H5VL_info_class_t {
2     size_t size;
3     void * (*copy)(const void *info);
4     herr_t (*cmp)(int *cmp_value, const void *info1, const void *info2);
5     herr_t (*free)(void *info);
6     herr_t (*to_str)(const void *info, char **str);
7     herr_t (*from_str)(const char *str, void **info);
8 } H5VL_info_class_t;
```

Listing 3: Info class for connector information routines, H5VLconnector.h

3.2.1 info: size

The `size` field indicates the size required to store any special information that the connector needs.

If the connector requires no special information, set this field to zero.

Signature:

```
1     size_t size;
```

3.2.2 info: copy

The `copy` callback is invoked when the connector is selected for use with `H5Pset_fapl()`, the connector-specific set call, etc. Where possible, the information should be deep copied in such a way that the original data can be freed.

Signature:

```
void * (*copy)(const void *info);
```

Arguments:

`info`	(IN): The connector-specific info to copy.

3.2.3 info: cmp

The `cmp` callback is uesd to determine if two connector-specific data structs are identical and helps the library manage connector resources.

Signature:

```
herr_t (*cmp)(int *cmp_value, const void *info1, const void *info2);
```

Arguments:

`cmp_value`	(OUT): A strcmp-like compare value.
`info1`	(IN): The 1st connector-specific info to copy.
`info2`	(IN): The 2nd connector-specific info to copy.

3.2.4 info: free

The `free` callback is used to clean up the connector-specific information that was copied when set in the fapl via the `copy` callback.

Signature:

```
herr_t (*free)(void *info);
```

Arguments:

`info`	(IN): The connector-specific info to free.

3.2.5 info: to_str

The `to_str` callback converts a connector-specific information structure to a connector-specific configuration string. It is the opposite of the `from_str` callback.

Signature:

```
herr_t (*to_str)(const void *info, char **str);
```

Arguments:

`info`	(IN): The connector-specific info to convert to a configuration string.
`str`	(OUT): The constructed configuration string.

3.2.6 info: from_str

The `to_str` callback converts a connector-specific configuration string to a connector-specific information structure. It is the opposite of the `to_str` callback.

Signature:

```
herr_t (*from_str)(const char *str, void **info);
```

Arguments:

`str`	(IN): The connector-specific configuration string.
`info`	(OUT): The connector-specific info generated from the configuration string.

3.3 Object Wrap Callbacks

The object wrap callbacks are used by passthrough connectors to wrap/unwrap objects and contexts when passing them up and down the VOL chain.

```
typedef struct H5VL_wrap_class_t {
    void * (*get_object)(const void *obj);
    herr_t (*get_wrap_ctx)(const void *obj, void **wrap_ctx);
    void * (*wrap_object)(void *obj, H5I_type_t obj_type, void *wrap_ctx);
    void * (*unwrap_object)(void *obj);
    herr_t (*free_wrap_ctx)(void *wrap_ctx);
} H5VL_wrap_class_t;
```

Listing 4: Wrap class for object wrapping routines, H5VLconnector.h

3.3.1 wrap: get_object

Retrieves an underlying object.

Signature:

```
    void * (*get_object)(const void *obj);
```

Arguments:

obj	(IN): Object being unwrapped.

3.3.2 wrap: get_wrap_ctx

Get a VOL connector's object wrapping context.

Signature:

```
    herr_t (*get_wrap_ctx)(const void *obj, void **wrap_ctx);
```

Arguments:

obj	(IN): Object for which we need a context.
wrap_ctx	(OUT): Context.

3.3.3 wrap: wrap_object

Asks a connector to wrap an underlying object.

Signature:

```
    void * (*wrap_object)(void *obj, H5I_type_t obj_type, void *wrap_ctx);
```

Arguments:

obj	(IN): Object being wrapped.
obj_type	(IN): Object type (see H5Ipublic.h).
wrap_ctx	(IN): Context.

3.3.4 wrap: unwrap_object

Unwrap an object from connector.

Signature:

```
    void * (*unwrap_object)(void *obj);
```

Arguments:
`obj` (IN): Object being unwrapped.

3.3.5 wrap: free_wrap_ctx

Release a VOL connector's object wrapping context.

Signature:

```
herr_t (*free_wrap_ctx)(void *wrap_ctx);
```

Arguments:
`wrap_ctx` (IN): Context to be freed.

3.4 The Attribute Function Callbacks

The attribute API routines (H5A) allow HDF5 users to create and manage HDF5 attributes. All the H5A API routines that modify the HDF5 container map to one of the attribute callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_attr_class_t {
    void *(*create)(void *obj, const H5VL_loc_params_t *loc_params, const char *attr_name, hid_t
        type_id, hid_t space_id, hid_t acpl_id, hid_t aapl_id, hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char *attr_name, hid_t
        aapl_id, hid_t dxpl_id, void **req);
    herr_t (*read)(void *attr, hid_t mem_type_id, void *buf, hid_t dxpl_id, void **req);
    herr_t (*write)(void *attr, hid_t mem_type_id, const void *buf, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_attr_get_t get_type, hid_t dxpl_id, void **req, va_list
        arguments);
    herr_t (*specific)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_attr_specific_t
        specific_type, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*close) (void *attr, hid_t dxpl_id, void **req);
} H5VL_attr_class_t;
```

Listing 5: Structure for attribute callback routines, H5VLconnector.h

3.4.1 attr: create

The `create` callback in the attribute class creates an attribute object in the container of the location object and returns a pointer to the attribute structure containing information to access the attribute in future calls.

Signature:

```
void *(*create)(void *obj, H5VL_loc_params_t *loc_params,
    const char *attr_name, hid_t type_id, hid_t space_id,
    hid_t acpl_id, hid_t aapl_id,
    hid_t dxpl_id, void **req);
```

Arguments:

obj	(IN): Pointer to an object where the attribute needs to be created or where the look-up of the target object needs to start.
loc_params	(IN): Pointer to the location parameters as explained in Section 3.1.
attr_name	(IN): The name of the attribute to be created.
type_id	(IN): The datatype of the attribute.
space_id	(IN): The dataspace of the attribute.
acpl_id	(IN): The attribute creation property list.
aapl_id	(IN): The attribute access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.4.2 attr: open

The `open` callback in the attribute class opens an attribute object in the container of the location object and returns a pointer to the attribute structure containing information to access the attribute in future calls.

Signature:

```
1    void *(*open)(void *obj, H5VL_loc_params_t *loc_params,
2        const char *attr_name, hid_t aapl_id, hid_t dxpl_id, void **req);
```

Arguments:

obj	(IN): Pointer to an object where the attribute needs to be opened or where the look-up of the target object needs to start.
loc_params	(IN): Pointer to the location parameters as explained in Section 3.1.
attr_name	(IN): The name of the attribute to be opened.
aapl_id	(IN): The attribute access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.4.3 attr: read

The `read` callback in the attribute class reads data from the attribute object and returns an `herr_t` indicating success or failure.

Signature:

```
1    herr_t (*read)(void *attr, hid_t mem_type_id, void *buf,
2        hid_t dxpl_id, void **req);
```

Arguments:

attr	(IN): Pointer to the attribute object.
mem_type_id	(IN): The memory datatype of the attribute.
buf	(OUT): Data buffer to be read into.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.4.4 attr: write

The `write` callback in the attribute class writes data to the attribute object and returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*write)(void *attr, hid_t mem_type_id, const void *buf,
    hid_t dxpl_id, void **req);
```

Arguments:

attr	(IN): Pointer to the attribute object.
mem_type_id	(IN): The memory datatype of the attribute.
buf	(IN): Data buffer to be written.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.4.5 attr: get

The get callback in the attribute class retrieves information about the attribute as specified in the get_type parameter.It returns an herr_t indicating success or failure.

Signature:

```
herr_t (*get)(void *obj, H5VL_attr_get_t get_type, hid_t dxpl_id,
    void **req, va_list arguments);
```

The get_type argument is an enum:

```
/* types for attribute GET callback */
typedef enum H5VL_attr_get_t {
    H5VL_ATTR_GET_ACPL,                 /* creation property list          */
    H5VL_ATTR_GET_INFO,                 /* info                            */
    H5VL_ATTR_GET_NAME,                 /* access property list            */
    H5VL_ATTR_GET_SPACE,                /* dataspace                       */
    H5VL_ATTR_GET_STORAGE_SIZE,         /* storage size                    */
    H5VL_ATTR_GET_TYPE                  /* datatype                        */
} H5VL_attr_get_t;
```

Arguments:

obj	(IN): An attribute or location object where information needs to be retrieved from.
get_type	(IN): The type of the information to retrieve.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The arguments argument contains a variable list of arguments depending on the get_type parameter. The following list shows the argument list, in order, for each type:

- H5VL_ATTR_GET_ACPL, to retrieve the attribute creation property list of the attribute specified in obj:
  1. hid_t *ret_id (OUT): buffer for the identifier of the attribute creation property list.
- H5VL_ATTR_GET_INFO, to retrieve the attribute info:
  1. H5VL_loc_params_t *loc_params (IN): Pointer to the location parameters explained in Section 3.1.
  2. H5A_info_t *ainfo (OUT): info structure to fill the attribute info in.
- H5VL_ATTR_GET_NAME, to retrieve an attribute name on a particular object specified in obj:
  1. H5VL_loc_params_t *loc_params (IN): Pointer to the location parameters explained in Section 3.1. The type could be either H5VL_OBJECT_BY_SELF meaning obj is the attribute, or H5VL_OBJECT_BY_IDX meaning the attribute to retrieve the name for should be looked up using the index information on the object in obj and the index information in loc_params.

2. `size_t buf_size` (IN): the size of the buffer to store the name in.
3. `void *buf` (OUT): Buffer to store the name in.
4. `ssize_t *ret_val` (OUT): return the actual size needed to store the fill attribute name.

- `H5VL_ATTR_GET_SPACE`, to retrieve the dataspace of the attribute specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the attribute dataspace.
- `H5VL_ATTR_GET_STORAGE_SIZE`, to retrieve the storage size of the attribute specified in `obj`:
  1. `hsize_t *ret` (OUT): buffer for the storage size of the attribute in the container.
- `H5VL_ATTR_GET_TYPE`, to retrieve the datatype of the attribute specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the attribute datatype.

3.4.6 attr: specific

The `specific` callback in the attribute class implements specific operations on HDF5 attributes as specified in the `specific_type` parameter. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*specific)(void *obj, H5VL_loc_params_t *loc_params, H5VL_attr_specific_t
    specific_type, hid_t dxpl_id, void **req, va_list arguments);
```

The `specific_type` argument is an `enum`:

```
/* types for attribute SPECFIC callback */
typedef enum H5VL_attr_specific_t {
    H5VL_ATTR_DELETE,                   /* H5Adelete(_by_name/idx)          */
    H5VL_ATTR_EXISTS,                   /* H5Aexists(_by_name)              */
    H5VL_ATTR_ITER,                     /* H5Aiterate(_by_name)             */
    H5VL_ATTR_RENAME                    /* H5Arename(_by_name)              */
} H5VL_attr_specific_t;
```

Arguments:

`obj`	(IN): The location object where the operation needs to happen.
`loc_params`	(IN): A pointer to the location parameters as explained in Section 3.1.
`specific_type`	(IN): The type of the operation.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `specific_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_ATTR_DELETE`, to delete an attribute on an object:
  1. `char *attr_name` (IN): the name of the attribute to delete.
- `H5VL_ATTR_EXISTS`, to check if an attribute exists on a particular object specified in `obj`:
  1. `char *attr_name` (IN): the attribute name to check.
  2. `htri_t *ret` (OUT): existence result, 0 if false, 1 if true.
- `H5VL_ATTR_ITER`, to iterate over all attributes of an object and call a user-specified callback on each one:
  1. `H5_index_t idx_type` (IN): Type of index.

2. `H5_iter_order_t order` (IN): Order in which to iterate over index.
3. `hsize_t *idx` (IN/OUT): Initial and return offset within index.
4. `H5A_operator2_t op` (IN): User-defined function to pass each attribute to.
5. `void *op_data` (IN/OUT): User data to pass through to and to be returned by iterator operator function.

- `H5VL_ATTR_RENAME`, to rename an attribute on an object:
  1. `char *old_name` (IN): the original name of the attribute.
  2. `char *new_name` (IN): the new name to assign for the attribute.

3.4.7 attr: optional

The `optional` callback in the attribute class implements connector specific operations on an HDF5 attribute. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.4.8 attr: close

The `close` callback in the attribute class terminates access to the attribute object and free all resources it was consuming, and returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*close)(void *attr, hid_t dxpl_id, void **req);
```

Arguments:

`attr`	(IN): Pointer to the attribute object.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.5 Dataset Callbacks

The dataset API routines (H5D) allow HDF5 users to create and manage HDF5 datasets. All the H5D API routines that modify the HDF5 container map to one of the dataset callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_dataset_class_t {
    void *(*create)(void *obj, const H5VL_loc_params_t *loc_params, const char *name, hid_t
        lcpl_id, hid_t type_id, hid_t space_id, hid_t dcpl_id, hid_t dapl_id, hid_t dxpl_id, void
        **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char *name, hid_t dapl_id,
        hid_t dxpl_id, void **req);
    herr_t (*read)(void *dset, hid_t mem_type_id, hid_t mem_space_id, hid_t file_space_id, hid_t
        dxpl_id, void * buf, void **req);
    herr_t (*write)(void *dset, hid_t mem_type_id, hid_t mem_space_id, hid_t file_space_id, hid_t
        dxpl_id, const void * buf, void **req);
    herr_t (*get)(void *obj, H5VL_dataset_get_t get_type, hid_t dxpl_id, void **req, va_list
        arguments);
    herr_t (*specific)(void *obj, H5VL_dataset_specific_t specific_type, hid_t dxpl_id, void
        **req, va_list arguments);
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*close) (void *dset, hid_t dxpl_id, void **req);
} H5VL_dataset_class_t;
```

Listing 6: Structure for dataset callback routines, H5VLconnector.h

3.5.1 dataset: create

The `create` callback in the dataset class creates a dataset object in the container of the location object and returns a pointer to the dataset structure containing information to access the dataset in future calls.

Signature:

```
    void *(*create)(void *obj, H5VL_loc_params_t *loc_params, const char *name, hid_t lcpl_id,
        hid_t type_id, hid_t space_id, hid_t dcpl_id, hid_t dapl_id, hid_t dxpl_id, void **req);
```

Arguments:

`obj`	(IN): Pointer to an object where the dataset needs to be created or where the look-up of the target object needs to start.
`loc_params`	(IN): Pointer to the location parameters as explained in Section 3.1. The type can be only `H5VL_OBJECT_BY_SELF` in this callback.
`name`	(IN): The name of the dataset to be created.
`lcpl_id`	(IN): The link creation property list.
`type_id`	(IN): The datatype of the dataset.
`space_id`	(IN): The dataspace of the dataset.
`dcpl_id`	(IN): The dataset creation property list.
`dapl_id`	(IN): The dataset access property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.5.2 dataset: open

The `open` callback in the dataset class opens a dataset object in the container of the location object and returns a pointer to the dataset structure containing information to access the dataset in future calls.

Signature:

```
    void *(*open)(void *obj, H5VL_loc_params_t *loc_params, const char *name, hid_t dapl_id,
        hid_t dxpl_id, void **req);
```

Arguments:

obj	(IN): Pointer to an object where the dataset needs to be opened or where the look-up of the target object needs to start.
loc_params	(IN): Pointer to the location parameters as explained in Section 3.1. The type can be only H5VL_OBJECT_BY_SELF in this callback.
name	(IN): The name of the dataset to be opened.
dapl_id	(IN): The dataset access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.5.3 dataset: read

The read callback in the dataset class reads data from the dataset object and returns an herr_t indicating success or failure.

Signature:

```
herr_t (*read)(void *dset, hid_t mem_type_id, hid_t mem_space_id,
    hid_t file_space_id, hid_t dxpl_id, void *buf, void **req);
```

Arguments:

dset	(IN): Pointer to the dataset object.
mem_type_id	(IN): The memory datatype of the data.
mem_space_id	(IN): The memory dataspace selection.
file_space_id	(IN): The file dataspace selection.
dxpl_id	(IN): The data transfer property list.
buf	(OUT): Data buffer to be read into.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.5.4 dataset: write

The write callback in the dataset class writes data to the dataset object and returns an herr_t indicating success or failure.

Signature:

```
herr_t (*write)(void *dset, hid_t mem_type_id, hid_t mem_space_id,
    hid_t file_space_id, hid_t dxpl_id, const void *buf, void **req);
```

Arguments:

dset	(IN): Pointer to the dataset object.
mem_type_id	(IN): The memory datatype of the data.
mem_space_id	(IN): The memory dataspace selection.
file_space_id	(IN): The file dataspace selection.
dxpl_id	(IN): The data transfer property list.
buf	(IN): Data buffer to be written from.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.5.5 dataset: get

The get callback in the dataset class retrieves information about the dataset as specified in the get_type parameter.It returns an herr_t indicating success or failure.

Signature:

```
herr_t (*get)(void *dset, H5VL_dataset_get_t get_type,
```

```
        hid_t dxpl_id, void **req, va_list arguments);
```

The `get_type` argument is an `enum`:

```
/* types for dataset GET callback */
typedef enum H5VL_dataset_get_t {
    H5VL_DATASET_GET_DAPL,              /* access property list           */
    H5VL_DATASET_GET_DCPL,              /* creation property list         */
    H5VL_DATASET_GET_OFFSET,            /* offset                         */
    H5VL_DATASET_GET_SPACE,             /* dataspace                      */
    H5VL_DATASET_GET_SPACE_STATUS,      /* space status                   */
    H5VL_DATASET_GET_STORAGE_SIZE,      /* storage size                   */
    H5VL_DATASET_GET_TYPE               /* datatype                       */
} H5VL_dataset_get_t;
```

Arguments:

`dset`	(IN): The dataset object where information needs to be retrieved from.
`get_type`	(IN): The type of the information to retrieve.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `get_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_DATASET_GET_DAPL`, to retrieve the dataset access property list of the dataset specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the dataset access property list.
- `H5VL_DATASET_GET_DCPL`, to retrieve the dataset creation property list of the dataset specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the dataset creation property list.
- `H5VL_DATASET_GET_OFFSET`, to retrieve the offset of the dataset specified in `obj` in the container:
  1. `haddr_t *ret` (OUT): buffer for the offset of the dataset in the container.
- `H5VL_DATASET_GET_SPACE`, to retrieve the dataspace of the dataset specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the dataset dataspace.
- `H5VL_DATASET_GET_SPACE_STATUS`, to retrieve the information whether space has been allocated for the dataset:
  1. `H5D_space_status_t *allocation` (OUT): buffer for the space status.
- `H5VL_DATASET_GET_STORAGE_SIZE`, to retrieve the storage size of the dataset specified in `obj`:
  1. `hsize_t *ret` (OUT): buffer for the storage size of the dataset in the container.
- `H5VL_DATASET_GET_TYPE`, to retrieve the datatype of the dataset specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the dataset datatype.

3.5.6 dataset: specific

The `specific` callback in the dataset class implements specific operations on HDF5 datasets as specified in the `specific_type` parameter. It returns an `herr_t` indicating success or failure.

Signature:

```
    herr_t (*specific)(void *obj, H5VL_file_specific_t specific_type, hid_t dxpl_id, void **req,
        va_list arguments);
```

The `specific_type` argument is an `enum`:

```
/* types for dataset SPECFIC callback */
typedef enum H5VL_dataset_specific_t {
    H5VL_DATASET_SET_EXTENT,            /* H5Dset_extent                */
    H5VL_DATASET_FLUSH,                 /* H5Dflush                     */
    H5VL_DATASET_REFRESH                /* H5Drefresh                   */
} H5VL_dataset_specific_t;
```

Arguments:

`obj`	(IN): The dset where the operation needs to happen.
`specific_type`	(IN): The type of the operation.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `specific_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_DATASET_SET_EXTENT`, changes the extent of the dataset dimensions:
  1. `hsize_t size` (IN): Array containing the new magnitude of each dimension of the dataset.
- `H5VL_DATASET_FLUSH`, flushes a dataset:
  1. `hid_t dset_id` (IN): The dataset ID. Needed so that the refreshed data can be associated with the old ID.
- `H5VL_DATASET_REFRESH`, clears a dataset's buffers and reloads from disk:
  1. `hid_t dset_id` (IN): The dataset ID. Needed so that the refreshed data can be associated with the old ID.

3.5.7 dataset: optional

The `optional` callback in the dataset class implements connector specific operations on an HDF5 dataset. It returns an `herr_t` indicating success or failure.

Signature:

```
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.5.8 dataset: close

The `close` callback in the dataset class terminates access to the dataset object and free all resources it was consuming and returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*close)(void *dset, hid_t dxpl_id, void **req);
```

Arguments:

dset	(IN): Pointer to the dataset object.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.6 Datatype Callbacks

The HDF5 datatype routines (H5T) allow users to create and manage HDF5 datatypes. Those routines are divided into two categories. One that operates on all types of datatypes but do not modify the contents of the container (all in memory), and others that operate on named datatypes by accessing the container. When a user creates an HDF5 datatype, it is still an object in memory space (transient datatype) that has not been added to the HDF5 containers. Only when a user commits the HDF5 datatype, it becomes persistent in the container. Those are called named/committed datatypes. The transient H5T routines should work on named datatypes nevertheless.

All the H5T API routines that modify the HDF5 container map to one of the named datatype callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_datatype_class_t {
    void *(*commit)(void *obj, const H5VL_loc_params_t *loc_params, const char *name, hid_t
        type_id, hid_t lcpl_id, hid_t tcpl_id, hid_t tapl_id, hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char * name, hid_t
        tapl_id, hid_t dxpl_id, void **req);
    herr_t (*get)  (void *obj, H5VL_datatype_get_t get_type, hid_t dxpl_id, void **req, va_list
        arguments);
    herr_t (*specific)(void *obj, H5VL_datatype_specific_t specific_type, hid_t dxpl_id, void
        **req, va_list arguments);
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*close) (void *dt, hid_t dxpl_id, void **req);
} H5VL_datatype_class_t;
```

Listing 7: Structure for datatype callback routines, H5VLconnector.h

3.6.1 datatype: commit

The `commit` callback in the named datatype class creates a datatype object in the container of the location object and returns a pointer to the datatype structure containing information to access the datatype in future calls.

Signature:

```
void *(*commit)(void *obj, H5VL_loc_params_t *loc_params,
    const char *name, hid_t type_id, hid_t lcpl_id, hid_t tcpl_id,
    hid_t tapl_id, hid_t dxpl_id, void **req);
```

Arguments:

obj	(IN): Pointer to an object where the datatype needs to be committed or where the look-up of the target object needs to start.
loc_params	(IN): Pointer to location parameters as explained in Section 3.1. In this call, the location type is always H5VL_OBJECT_BY_SELF.
name	(IN): The name of the datatype to be created.
type_id	(IN): The transient datatype identifier to be committed.
lcpl_id	(IN): The link creation property list.
tcpl_id	(IN): The datatype creation property list.
tapl_id	(IN): The datatype access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.6.2 datatype: open

The open callback in the named datatype class opens a previously committed datatype object in the container of the location object and returns a pointer to the datatype structure containing information to access the datatype in future calls.

Signature:

```
1    void *(*open) (void *obj, H5VL_loc_params_t *loc_params,
2        const char * name, hid_t tapl_id, hid_t dxpl_id, void **req);
```

Arguments:

obj	(IN): Pointer to an object where the datatype needs to be opened or where the look-up of the target object needs to start.
loc_params	(IN): Pointer to location parameters as explained in Section 3.1. In this call, the location type is always H5VL_OBJECT_BY_SELF.
name	(IN): The name of the datatype to be opened.
tapl_id	(IN): The datatype access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.6.3 datatype: get

The get callback in the named datatype class retrieves information about the named datatype as specified in the get_type parameter.It returns an herr_t indicating success or failure.

Signature:

```
1    herr_t (*get) (void *obj, H5VL_datatype_get_t get_type,
2        hid_t dxpl_id, void **req, va_list arguments);
```

The get_type argument is an enum:

```
1  /* types for datatype GET callback */
2  typedef enum H5VL_datatype_get_t {
3      H5VL_DATATYPE_GET_BINARY,          /* get serialized form of transient type */
4      H5VL_DATATYPE_GET_TCPL             /* datatype creation property list      */
5  } H5VL_datatype_get_t;
```

Arguments:

obj	(IN): The named datatype to retrieve information from.
get_type	(IN): The type of the information to retrieve.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The arguments argument contains a variable list of arguments depending on the get_type parameter. The following list shows the argument list, in order, for each type:

- H5VL_DATATYPE_GET_BINARY, to retrieve the serialized original transient HDF5 datatype that was committed, or return the size that is required for it be serialized if the passed in buffer is NULL. The HDF5 library provides two functions to encode and decode datatypes in their transient form, H5Tencode() and H5Tdecode(). When a datatype is committed, the connector is required to keep the serialized form of the transient datatype stored somewhere in the container (which is usually the case anyway when committing a named datatype), so it can be retrieved with this call. This is needed to generate the higher level HDF5 datatype identifier that allows all the H5T "transient" routines to work properly on the named datatype.
  1. ssize_t *nalloc (OUT): buffer to store the total size of the serialized datatype.
  2. void *buf (OUT): buffer to store the serialized datatype.
  3. size_t size (IN): size of the passed in buffer.
- H5VL_DATATYPE_GET_TCPL, to retrieve the datatype creation property list:
  1. hid_t *ret_id (OUT): buffer for the identifier of the type creation property list.

3.6.4 datatype: specific

The specific callback in the datatype class implements specific operations on HDF5 named datatypes as specified in the specific_type parameter. It returns an herr_t indicating success or failure.

Signature:

```
herr_t (*specific)(void *obj, H5VL_loc_params_t *loc_params, H5VL_object_specific_t
    specific_type, hid_t dxpl_id, void **req, va_list arguments);
```

The specific_type argument is an enum:

```
/* types for datatype SPECFIC callback */
typedef enum H5VL_datatype_specific_t {
    H5VL_DATATYPE_FLUSH,
    H5VL_DATATYPE_REFRESH
} H5VL_datatype_specific_t;
```

Arguments:

obj	(IN): The container or object where the operation needs to happen.
loc_params	(IN): Pointer to location parameters as explained in Section 3.1.
specific_type	(IN): The type of the operation.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The arguments argument contains a variable list of arguments depending on the specific_type parameter. The following list shows the argument list, in order, for each type:

- H5VL_DATATYPE_FLUSH, flushes a datatype:
  1. hid_t type_id (IN): The datatype ID. Needed so that the refreshed data can be associated with the old ID.

- `H5VL_DATATYPE_REFRESH`, clears a datatype's buffers and reloads from disk:
  1. `hid_t type_id` (IN): The datatype ID. Needed so that the refreshed data can be associated with the old ID.

3.6.5 datatype: optional

The `optional` callback in the datatype class implements connector specific operations on an HDF5 datatype. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.6.6 datatype: close

The `close` callback in the named datatype class terminates access to the datatype object and free all resources it was consuming and returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*close) (void *dt, hid_t dxpl_id, void **req);
```

Arguments:

`dt`	(IN): Pointer to the datatype object.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.7 File Callbacks

The file API routines (H5F) allow HDF5 users to create and manage HDF5 containers. All the H5F API routines that modify the HDF5 container map to one of the file callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_file_class_t {
    void *(*create)(const char *name, unsigned flags, hid_t fcpl_id, hid_t fapl_id, hid_t dxpl_id,
        void **req);
    void *(*open)(const char *name, unsigned flags, hid_t fapl_id, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_file_get_t get_type, hid_t dxpl_id, void **req, va_list
        arguments);
    herr_t (*specific)(void *obj, H5VL_file_specific_t specific_type, hid_t dxpl_id, void **req,
        va_list arguments);
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*close) (void *file, hid_t dxpl_id, void **req);
} H5VL_file_class_t;
```

Listing 8: File class for file API routines, H5VLconnector.h

3.7.1 file: create

The `create` callback in the file class should create a container and returns a pointer to the file structure created by the connector containing information to access the container in future calls.

Signature:

```
1    void *(*create)(const char *name, unsigned flags, hid_t fcpl_id, hid_t fapl_id, hid_t
         dxpl_id, void **req);
```

Arguments:

`name`	(IN): The name of the container to be created.
`flags`	(IN): The creation flags of the container.
`fcpl_id`	(IN): The file creation property list.
`fapl_id`	(IN): The file access property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.7.2 file: open

The `open` callback in the file class should open a container and returns a pointer to the file structure created by the connector containing information to access the container in future calls.

Signature:

```
1    void *(*open)(const char *name, unsigned flags, hid_t fapl_id, hid_t dxpl_id, void **req);
```

Arguments:

`name`	(IN): The name of the container to open.
`flags`	(IN): The open flags of the container.
`fapl_id`	(IN): The file access property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.7.3 file: get

The `get` callback in the file class should retrieve information about the container as specified in the `get_type` parameter. It returns an `herr_t` indicating success or failure.

Signature:

```
1    herr_t (*get)(void *obj, H5VL_file_get_t get_type, hid_t dxpl_id,
2        void **req, va_list arguments);
```

The `get_type` argument is an `enum`:

```
1  /* types for all file get API routines */
2  typedef enum H5VL_file_get_t {
3      H5VL_FILE_GET_CONT_INFO,            /* file get container info          */
4      H5VL_FILE_GET_FAPL,                 /* file access property list        */
5      H5VL_FILE_GET_FCPL,                 /* file creation property list      */
6      H5VL_FILE_GET_FILENO,               /* file number                      */
7      H5VL_FILE_GET_INTENT,               /* file intent                      */
8      H5VL_FILE_GET_NAME,                 /* file name                        */
9      H5VL_FILE_GET_OBJ_COUNT,            /* object count in file             */
```

```
    H5VL_FILE_GET_OBJ_IDS               /* object ids in file               */
} H5VL_file_get_t;
```

Arguments:

obj	(IN): The container or object where information needs to be retrieved from.
get_type	(IN): The type of the information to retrieve.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The arguments argument contains a variable list of arguments depending on the get_type parameter. The following list shows the argument list, in order, for each type:

- H5VL_OBJECT_GET_CONT_INFO, get information about the container that the object belongs to:
  1. H5VL_file_cont_info_t *info (OUT): pointer to the file container info structure returned.
- H5VL_FILE_GET_FAPL, to retrieve the file access property list:
  1. hid_t *ret_id (OUT): buffer for the identifier of the file access property list.
- H5VL_FILE_GET_FCPL, to retrieve the file creation property list:
  1. hid_t *ret_id (OUT): buffer for the identifier of the file creation property list.
- H5VL_FILE_GET_FILENO, to retrieve the file's file number:
  1. unsigned long *fileno (OUT): buffer for the file number.
- H5VL_FILE_GET_INTENT, get access intent of the container:
  1. unsigned *intent_flags (OUT): buffer for the intent flags value.
- H5VL_FILE_GET_NAME, get container name an object belongs to:
  1. H5I_type_t type (IN): the object type in obj.
  2. size_t size (IN): size of the buffer for the file name.
  3. char *name (OUT): buffer for the file name.
  4. ssize_t *ret (OUT): buffer for the entire size of the file name.
- H5VL_FILE_GET_OBJ_COUNT:, to retrieve the object count in the container:
  1. unsigned types (IN): type of objects to look for.
  2. ssize_t *ret (OUT): buffer for the object count.
- H5VL_FILE_GET_OBJ_IDS:, to retrieve object identifiers in the container:
  1. unsigned types (IN): type of objects to look for.
  2. size_t max_objs (IN): maximum number of objects to open.
  3. hid_t *oid_list (OUT): buffer for the object identifiers.
  4. ssize_t *ret (OUT): buffer for the object count.

3.7.4 file: specific

The specific callback in the file class implements specific operations on HDF5 files as specified in the specific_type parameter. It returns an herr_t indicating success or failure.

Signature:

```
herr_t (*specific)(void *obj, H5VL_file_specific_t specific_type, hid_t dxpl_id, void **req,
    va_list arguments);
```

The `specific_type` argument is an `enum`:

```
/* types for file SPECIFIC callback */
typedef enum H5VL_file_specific_t {
    H5VL_FILE_POST_OPEN,                /* Adjust file after open, with wrapping context */
    H5VL_FILE_FLUSH,                    /* Flush file                                    */
    H5VL_FILE_REOPEN,                   /* Reopen the file                               */
    H5VL_FILE_MOUNT,                    /* Mount a file                                  */
    H5VL_FILE_UNMOUNT,                  /* Unmount a file                                */
    H5VL_FILE_IS_ACCESSIBLE,            /* Check if a file is accessible */
    H5VL_FILE_DELETE,                   /* Delete a file                                 */
    H5VL_FILE_IS_EQUAL                  /* Check if two files are the same */
} H5VL_file_specific_t;
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`specific_type`	(IN): The type of the operation.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `specific_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_FILE_POST_OPEN`, adjust a file after opening it:
- `H5VL_FILE_FLUSH`, flushes all buffers associated with the container to disk:
  1. `H5I_type_t obj_type` (IN): The object type of `obj` on which the flush operation was called.
  2. `H5F_scope_t scope` (IN): The scope of the flushing action.
- `H5VL_FILE_REOPEN`, reopens a file:
  1. `void *file` (IN): pointer to the reopened file (same type of pointer as is returned from the connector's create/open callbacks).
- `H5VL_FILE_MOUNT`, Mounts a file on the location object:
  1. `H5I_type_t type` (IN): the object type in `obj`.
  2. `char *name` (IN): name of the group onto which the file specified by `file` is to be mounted.
  3. `void *child` (IN): child file to be mounted (same type of pointer as is returned from the connector's create/open callbacks).
  4. `hid_t *fmpl_id` (IN): file mount property list.
- `H5VL_FILE_UNMOUNT`, un-mounts a file from the location object:
  1. `H5I_type_t type` (IN): the object type in `obj`.
  2. `char *name` (IN): name of the mount point.
- `H5VL_FILE_IS_ACCESSIBLE`, checks if a container is accessible using a specific file access property list:
  1. `hid_t *fapl_id` (IN): file access property list.
  2. `char *name` (IN): name of the container to check.
  3. `htri_t *result` (OUT): buffer for the result; 0 if no, 1 if yes.
- `H5VL_FILE_DELETE`, deletes a file:
  1. `H5I_type_t obj_type` (IN): The object type of `obj` on which the flush operation was called.
  2. `H5F_scope_t scope` (IN): The scope of the flushing action.
- `H5VL_FILE_IS_EQUAL`, check if one file is equal to another:

1. `H5I_type_t obj_type` (IN): The object type of `obj` on which the flush operation was called.
2. `H5F_scope_t scope` (IN): The scope of the flushing action.

3.7.5 file: optional

The `optional` callback in the file class implements connector specific operations on an HDF5 container. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.7.6 file: close

The `close` callback in the file class should terminate access to the file object and free all resources it was consuming, and returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*close)(void *file, hid_t dxpl_id, void **req);
```

Arguments:

`file`	(IN): Pointer to the file.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.8 Group Callbacks

The group API routines (H5G) allow HDF5 users to create and manage HDF5 groups. All the H5G API routines that modify the HDF5 container map to one of the group callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_group_class_t {
    void *(*create)(void *obj, const H5VL_loc_params_t *loc_params, const char *name, hid_t
        lcpl_id, hid_t gcpl_id, hid_t gapl_id, hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char *name, hid_t gapl_id,
        hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_group_get_t get_type, hid_t dxpl_id, void **req, va_list
        arguments);
    herr_t (*specific)(void *obj, H5VL_group_specific_t specific_type, hid_t dxpl_id, void **req,
        va_list arguments);
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*close) (void *grp, hid_t dxpl_id, void **req);
} H5VL_group_class_t;
```

Listing 9: Structure for group callback routines, H5VLconnector.h

3.8.1 group: create

The `create` callback in the group class creates a group object in the container of the location object and returns a pointer to the group structure containing information to access the group in future calls.

Signature:

```
void *(*create)(void *obj, H5VL_loc_params_t *loc_params, const char *name, hid_t gcpl_id,
    hid_t gapl_id, hid_t dxpl_id, void **req);
```

Arguments:

`obj`	(IN): Pointer to an object where the group needs to be created or where the look-up of the target object needs to start.
`loc_params`	(IN): Pointer to the location parameters as explained in Section 3.1. The type can be only `H5VL_OBJECT_BY_SELF` in this callback.
`name`	(IN): The name of the group to be created.
`dcpl_id`	(IN): The group creation property list. It contains all the group creation properties in addition to the link creation property list of the create operation (an `hid_t`) that can be retrieved with the property `H5VL_GRP_LCPL_ID`.
`gapl_id`	(IN): The group access property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.8.2 group: open

The `open` callback in the group class opens a group object in the container of the location object and returns a pointer to the group structure containing information to access the group in future calls.

Signature:

```
void *(*open)(void *obj, H5VL_loc_params_t *loc_params,
    const char*name, hid_t gapl_id, hid_t dxpl_id, void **req);
```

Arguments:

`obj`	(IN): Pointer to an object where the group needs to be opened or where the look-up of the target object needs to start.
`loc_params`	(IN): Pointer to the location parameters as explained in Section 3.1. The type can be only `H5VL_OBJECT_BY_SELF` in this callback.
`name`	(IN): The name of the group to be opened.
`dapl_id`	(IN): The group access property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.8.3 group: get

The `get` callback in the group class retrieves information about the group as specified in the `get_type` parameter. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*get)(void *obj, H5VL_group_get_t get_type, hid_t dxpl_id, void **req, va_list
    arguments);
```

The `get_type` argument is an `enum`:

```
/* types for all group get API routines */
```

```
typedef enum H5VL_group_get_t {
    H5VL_GROUP_GET_GCPL,                /* group creation property list */
    H5VL_GROUP_GET_INFO                 /* group info                   */
} H5VL_group_get_t;
```

Arguments:

obj	(IN): The group object where information needs to be retrieved from.
get_type	(IN): The type of the information to retrieve.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `get_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_GROUP_GET_GCPL`, to retrieve the group creation property list of the group specified in `obj`:
  1. `hid_t *ret_id` (OUT): buffer for the identifier of the group creation property list.
- `H5VL_GROUP_GET_INFO`, to retrieve the attribute info:
  1. `H5VL_loc_params_t *loc_params` (IN): Pointer to the location parameters explained in Section 3.1.
  2. `H5G_info_t *group_info` (OUT): info structure to fill the group info in.

3.8.4 group: specific

The `specific` callback in the group class implements specific operations on HDF5 groups as specified in the `specific_type` parameter. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*specific)(void *obj, H5VL_loc_params_t *loc_params, H5VL_object_specific_t
    specific_type, hid_t dxpl_id, void **req, va_list arguments);
```

The `specific_type` argument is an `enum`:

```
/* types for group SPECFIC callback */
typedef enum H5VL_group_specific_t {
    H5VL_GROUP_FLUSH,
    H5VL_GROUP_REFRESH
} H5VL_group_specific_t;
```

Arguments:

obj	(IN): The container or object where the operation needs to happen.
loc_params	(IN): Pointer to the location parameters as explained in Section 3.1.
specific_type	(IN): The type of the operation.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `specific_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_GROUP_FLUSH`, flushes a group:
  1. `hid_t grp_id` (IN): The group ID. Needed so that the refreshed data can be associated with the old ID.
- `H5VL_GROUP_REFRESH`, clears a group's buffers and reloads from disk:

1. `hid_t grp_id` (IN): The group ID. Needed so that the refreshed data can be associated with the old ID.

3.8.5 group: optional

The `optional` callback in the group class implements connector specific operations on an HDF5 group. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.8.6 group: close

The `close` callback in the group class terminates access to the group object and frees all resources it was consuming, and returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*close)(void *group, hid_t dxpl_id, void **req);
```

Arguments:

`group`	(IN): Pointer to the group object.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.9 Link Callbacks

The link API routines (H5L) allow HDF5 users to create and manage HDF5 links. All the H5L API routines that modify the HDF5 container map to one of the link callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_link_class_t {
    herr_t (*create)(H5VL_link_create_type_t create_type, void *obj, const H5VL_loc_params_t
        *loc_params, hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*copy)(void *src_obj, const H5VL_loc_params_t *loc_params1, void *dst_obj, const
        H5VL_loc_params_t *loc_params2, hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void **req);
    herr_t (*move)(void *src_obj, const H5VL_loc_params_t *loc_params1, void *dst_obj, const
        H5VL_loc_params_t *loc_params2, hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_link_get_t get_type, hid_t
        dxpl_id, void **req, va_list arguments);
    herr_t (*specific)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_link_specific_t
        specific_type, hid_t dxpl_id, void **req, va_list arguments);
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
} H5VL_link_class_t;
```

Listing 10: Structure for link callback routines, H5VLconnector.h

3.9.1 link: create

The `create` callback in the group class creates a hard, soft, external, or user-defined link in the container. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*create)(H5VL_link_create_type_t create_type, void *obj,
    H5VL_loc_params_t *loc_params, hid_t lcpl_id,
    hid_t lapl_id, hid_t dxpl_id, void **req);
```

The `create_type` argument is an `enum`:

```
/* link create types for VOL */
typedef enum H5VL_link_create_type_t {
    H5VL_LINK_CREATE_HARD, /* Hard Link       */
    H5VL_LINK_CREATE_SOFT, /* Soft Link       */
    H5VL_LINK_CREATE_UD    /* External / UD Link */
} H5VL_link_create_type_t;
```

Arguments:

`create_type`	(IN): type of the link to be created.
`obj`	(IN): Pointer to an object where the link needs to be created from.
`loc_params`	(IN): Pointer to the location parameters as explained in Section 3.1 for the source object.
`lcpl_id`	(IN): The link creation property list. It contains all the link creation properties in addition to other API parameters depending on the creation type, which will be detailed next.
`lapl_id`	(IN): The link access property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

As mentioned in the argument list, the `lcpl_id` contains the parameters for the link creation operation depending on the creation type:

- `H5VL_LINK_CREATE_HARD` contains two properties:
  1. `H5VL_LINK_TARGET` (with type `void*`): The target object where the hard link needs to be created to.
  2. `H5VL_LINK_TARGET_LOC_PARAMS` (with type `H5VL_loc_params_t`): The location parameters as explained in Section 3.1 for the target object.
- `H5VL_LINK_CREATE_SOFT` contains one property:
  1. `H5VL_LINK_TARGET_NAME` (with type `char*`): The target link where the soft link should point to.
- `H5VL_LINK_CREATE_UD` contains two properties:
  1. `H5VL_LINK_TYPE` (with type `H5L_type_t`): The user defined link class. `H5L_TYPE_EXTERNAL` suggests an external link is to be created.
  2. `H5VL_LINK_UDATA` (with type `void*`): User supplied link information (contains the external link buffer for external links).
  3. `H5VL_LINK_UDATA_SIZE` (with type `size_t`): size of the `udata` buffer.

3.9.2 link: copy

The `copy` callback in the link class copies a link within the HDF5 container. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*copy)(void *src_obj, H5VL_loc_params_t *loc_params1, void *dst_obj,
    H5VL_loc_params_t *loc_params2, hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void
    **req);
```

Arguments:

src_obj	(IN): original/source object or file.
loc_params1	(IN): Pointer to the location parameters for the source object as explained in Section 3.1. The type can be only H5VL_OBJECT_BY_NAME in this callback.
dst_obj	(IN): destination object or file.
loc_params1	(IN): Pointer to the location parameters for the destination object as explained in Section 3.1. The type can be only H5VL_OBJECT_BY_NAME in this callback.
lcpl_id	(IN): The link creation property list.
lapl_id	(IN): The link access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.9.3 link: move

The move callback in the link class moves a link within the HDF5 container. It returns an herr_t indicating success or failure.

Signature:

```
herr_t (*move)(void *src_obj, H5VL_loc_params_t *loc_params1, void *dst_obj,
    H5VL_loc_params_t *loc_params2, hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void
    **req);
```

Arguments:

src_obj	(IN): original/source object or file.
loc_params1	(IN): Pointer to the location parameters for the source object as explained in Section 3.1. The type can be only H5VL_OBJECT_BY_NAME in this callback.
dst_obj	(IN): destination object or file.
loc_params1	(IN): Pointer to the location parameters for the destination object as explained in Section 3.1. The type can be only H5VL_OBJECT_BY_NAME in this callback.
lcpl_id	(IN): The link creation property list.
lapl_id	(IN): The link access property list.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.9.4 link: get

The get callback in the link class retrieves information about links as specified in the get_type parameter. It returns an herr_t indicating success or failure.

Signature:

```
herr_t (*get)(void *obj, H5VL_loc_params_t *loc_params,
    H5VL_link_get_t get_type, hid_t dxpl_id, void **req,
    va_list arguments);
```

The get_type argument is an enum:

```
/* types for all link get API routines */
typedef enum H5VL_link_get_t {
    H5VL_LINK_GET_INFO,     /* link info                    */
    H5VL_LINK_GET_NAME,     /* link name                    */
```

```
    H5VL_LINK_GET_VAL       /* link value                   */
} H5VL_link_get_t;
```

Arguments:

obj	(IN): The file or group object where information needs to be retrieved from.
loc_params	(IN): Pointer to the location parameters for the source object as explained in Section 3.1. The type can be only H5VL_OBJECT_BY_NAME or H5VL_OBJECT_BY_IDX in this callback.
get_type	(IN): The type of the information to retrieve.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The arguments argument contains a variable list of arguments depending on the get_type parameter. The following list shows the argument list, in order, for each type:

- H5VL_LINK_GET_INFO, to retrieve the link info from the link specified in the loc_params:
  1. H5L_info_t *linfo (OUT): pointer to info structure to fill.
- H5VL_LINK_GET_NAME, to retrieve the name of the link specified by the index information in loc_params (loc_params is of type H5VL_OBJECT_BY_IDX only with this type):
  1. char* name (OUT): buffer to copy the name into.
  2. size_t size (IN): size of the buffer name, if 0, return only the buffer size needed.
  3. ssize_t *ret (OUT): buffer to return the length of the link name.
- H5VL_LINK_GET_VAL, to retrieve the link value from the link specified in the loc_params:
  1. void *buf (OUT): buffer to put the value into.
  2. size_t size (IN): size of the passed in buffer.

3.9.5 link: specific

The specific callback in the link class implements specific operations on HDF5 links as specified in the specific_type parameter. It returns an herr_t indicating success or failure.

Signature:

```
        herr_t (*specific)(void *obj, H5VL_loc_params_t *loc_params, H5VL_link_specific_t
            specific_type, hid_t dxpl_id, void **req, va_list arguments);
```

The specific_type argument is an enum:

```
/* types for link SPECIFIC callback */
typedef enum H5VL_link_specific_t {
    H5VL_LINK_DELETE,       /* H5Ldelete(_by_idx)           */
    H5VL_LINK_EXISTS,       /* link existence               */
    H5VL_LINK_ITER          /* H5Literate/visit(_by_name)         */
} H5VL_link_specific_t;
```

Arguments:

obj	(IN): The location object where the operation needs to happen.
loc_params	(IN): Pointer to the location parameters as explained in Section 3.1.
specific_type	(IN): The type of the operation.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `specific_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_LINK_DELETE`, to remove a link specified in the location parameter from an HDF5 container.
- `H5VL_LINK_EXISTS`, to determine whether the link specified in the `loc_params` exits (`loc_params` is of type `H5VL_OBJECT_BY_NAME` only with this type):
  1. `htri_t *ret` (OUT): buffer for the existence of the link (0 for no, 1 for yes).
- `H5VL_LINK_ITER`, to iterate over all links starting at the location provided and call a user-specified callback on each one:
  1. `hbool_t recursive` (IN): whether to recursively follow links into subgroups of the specified group.
  2. `H5_index_t idx_type` (IN): Type of index.
  3. `H5_iter_order_t order` (IN): Order in which to iterate over index.
  4. `hsize_t *idx_p` (IN/OUT): iteration position where to start and return position where an interrupted iteration may restart.
  5. `H5L_iterate_t op` (IN): User-defined function to pass each link to.
  6. `void *op_data` (IN/OUT): User data to pass through to and to be returned by iterator operator function.

3.9.6 link: optional

The `optional` callback in the link class implements connector specific operations on an HDF5 link. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.10 Object Callbacks

The object API routines (H5O) allow HDF5 users to manage HDF5 group, dataset, and named datatype objects. All the H5O API routines that modify the HDF5 container map to one of the object callback routines in this class that the connector needs to implement.

```
typedef struct H5VL_object_class_t {
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, H5I_type_t *opened_type, hid_t
        dxpl_id, void **req);
    herr_t (*copy)(void *src_obj, const H5VL_loc_params_t *loc_params1, const char *src_name, void
        *dst_obj, const H5VL_loc_params_t *loc_params2, const char *dst_name, hid_t ocpypl_id,
        hid_t lcpl_id, hid_t dxpl_id, void **req);
```

```
4     herr_t (*get)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_object_get_t get_type,
          hid_t dxpl_id, void **req, va_list arguments);
5     herr_t (*specific)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_object_specific_t
          specific_type, hid_t dxpl_id, void **req, va_list arguments);
6     herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
7 } H5VL_object_class_t;
```

Listing 11: Structure for object callback routines, H5VLconnector.h

3.10.1 object: open

The `open` callback in the object class opens the object in the container of the location object and returns a pointer to the object structure containing information to access the object in future calls.

Signature:

```
1     void *(*open)(void *obj, H5VL_loc_params_t *loc_params,
2         H5I_type_t *opened_type, hid_t dxpl_id, void **req);
```

Arguments:

`obj`	(IN): Pointer to a file or group where the objecy needs to be opened or where the look-up of the target object needs to start.
`loc_params`	(IN): Pointer to location parameters as explained in Section 3.1.
`opened_type`	(OUT): buffer to return the type of the object opened (`H5I_GROUP` or `H5I_DATASET` or `H5I_DATATYPE`).
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.10.2 object: copy

The `copy` callback in the object class copies the object from the source object to the destination object. It returns an `herr_t` indicating success or failure.

Signature:

```
1     herr_t (*copy)(void *src_obj, H5VL_loc_params_t *loc_params1,
2         const char *src_name, void *dst_obj,
3         H5VL_loc_params_t *loc_params2, const char *dst_name,
4         hid_t ocpypl_id, hid_t lcpl_id, hid_t dxpl_id, void **req);
```

Arguments:

`src_obj`	(IN): Pointer to location of the source object to be copied.
`loc_params1`	(IN): Pointer to location parameters as explained in Section 3.1. The type should only be `H5VL_OBJECT_BY_SELF` for this callback.
`src_name`	(IN): Name of the source object to be copied.
`dst_obj`	(IN): Pointer to location of the destination object.
`loc_params2`	(IN): Pointer to location parameters as explained in Section 3.1. The type should only be `H5VL_OBJECT_BY_SELF` for this callback.
`dst_name`	(IN): Name to be assigned to the new copy.
`ocpypl_id`	(IN): The object copy property list.
`lcpl_id`	(IN): The link creation property list.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.

3.10.3 object: get

The `get` callback in the object class retrieves information about the object as specified in the `get_type` parameter.It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*get)(void *obj, H5VL_loc_params_t *loc_params,
    H5VL_object_get_t get_type, hid_t dxpl_id,
    void **req, va_list arguments);
```

The `get_type` argument is an `enum`:

```
/* type for object GET callback */
typedef enum H5VL_object_get_t {
    H5VL_OBJECT_GET_FILE,          /* object's file                */
    H5VL_OBJECT_GET_NAME,          /* object name                  */
    H5VL_OBJECT_GET_TYPE           /* object type                  */
} H5VL_object_get_t;
```

Arguments:

`obj`	(IN): A location object where information needs to be retrieved from.
`loc_params`	(IN): Pointer to location parameters as explained in Section 3.1.
`get_type`	(IN): The type of the information to retrieve.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `get_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_OBJECT_GET_FILE`, to retrieve a file for a referenced object:
  1. `void **ret` (OUT): pointer to buffer to return the file.
- `H5VL_OBJECT_GET_NAME`, to retrieve a name for a referenced object:
  1. `ssize_t *ret` (OUT): buffer to return the length of the name.
  2. `char* name` (OUT): buffer to copy the name into.
  3. `size_t size` (IN): size of the buffer name, if 0, return only the buffer size needed.
- `H5VL_OBJECT_GET_TYPE`, to retrieve object type a reference points to:
  1. `H5O_type_t *type` (OUT): buffer to return the object type.

3.10.4 object: specific

The `specific` callback in the object class implements specific operations on HDF5 objects as specified in the `specific_type` parameter. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*specific)(void *obj, H5VL_loc_params_t *loc_params, H5VL_object_specific_t
    specific_type, hid_t dxpl_id, void **req, va_list arguments);
```

The `specific_type` argument is an `enum`:

```
/* types for object SPECIFIC callback */
typedef enum H5VL_object_specific_t {
    H5VL_OBJECT_CHANGE_REF_COUNT,    /* H5Oincr/decr_refcount         */
```

```
    H5VL_OBJECT_EXISTS,             /* H5Oexists_by_name            */
    H5VL_OBJECT_LOOKUP,             /* Lookup object                */
    H5VL_OBJECT_VISIT,              /* H5Ovisit(_by_name)           */
    H5VL_OBJECT_FLUSH,              /* H5{D|G|O|T}flush             */
    H5VL_OBJECT_REFRESH             /* H5{D|G|O|T}refresh           */
} H5VL_object_specific_t;
```

Arguments:

`obj`	(IN): The location object where the operation needs to happen.
`loc_params`	(IN): Pointer to location parameters as explained in Section 3.1.
`specific_type`	(IN): The type of the operation.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the get operation.

The `arguments` argument contains a variable list of arguments depending on the `specific_type` parameter. The following list shows the argument list, in order, for each type:

- `H5VL_OBJECT_CHANGE_REF_COUNT`, to update the reference count for the object in `obj`:
  1. `int ref_count` (IN): reference count to set on the object.
- `H5VL_OBJECT_EXISTS`, to check if an object with name specified in `loc_params` (type `H5VL_OBJECT_BY_NAME`) exists:
  1. `htri_t *ret` (OUT): existence result, 0 if false, 1 if true.
- `H5VL_OBJECT_LOOKUP`, Look up an object:
  1. `void *token` (IN): The object's token.
- `H5VL_OBJECT_VISIT`, to iterate over all objects starting at the location provided and call a user-specified callback on each one:
  1. `H5_index_t idx_type` (IN): Type of index.
  2. `H5_iter_order_t order` (IN): Order in which to iterate over index.
  3. `H5O_iterate_t op` (IN): User-defined function to pass each object to.
  4. `void *op_data` (IN/OUT): User data to pass through to and to be returned by iterator operator function.
  5. `unsigned fields` (IN): Fields in H5O_info_t struct to return.
- `H5VL_OBJECT_FLUSH`, flushes a object:
  1. `hid_t obj_id` (IN): The object ID. Needed so that the refreshed data can be associated with the old ID.
- `H5VL_OBJECT_REFRESH`, clears an object's buffers and reloads from disk:
  1. `hid_t obj_id` (IN): The object ID. Needed so that the refreshed data can be associated with the old ID.

3.10.5 object: optional

The `optional` callback in the object class implements connector specific operations on an HDF5 object. It returns an `herr_t` indicating success or failure.

Signature:

```
herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

obj	(IN): The container or object where the operation needs to happen.
dxpl_id	(IN): The data transfer property list.
req	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
arguments	(IN/OUT): argument list containing parameters and output pointers for the get operation.

Each connector should be able to parse the `va_list arguments` if it has connector specific operations to implement and determine the type of the operation and the parameters through a predefined schema.

3.11 Request (Async) Callbacks

```
typedef struct H5VL_request_class_t {
    herr_t (*wait)(void *req, uint64_t timeout, H5ES_status_t *status);
    herr_t (*notify)(void *req, H5VL_request_notify_t cb, void *ctx);
    herr_t (*cancel)(void *req);
    herr_t (*specific)(void *req, H5VL_request_specific_t specific_type, va_list arguments);
    herr_t (*optional)(void *req, va_list arguments);
    herr_t (*free)(void *req);
} H5VL_request_class_t;
```

Listing 12: Structure for async request callback routines, H5VLconnector.h

3.11.1 request: wait

Wait (with a timeout) for an async operation to complete. Releases the request if the operation has completed and the connector callback succeeds.

Signature:

```
herr_t (*wait)(void *req, uint64_t timeout, H5ES_status_t *status);
```

The `status` argument is an `enum` (from `H5ESpublic.h`):

```
/* Asynchronous operation status */
typedef enum H5ES_status_t {
    H5ES_STATUS_IN_PROGRESS, /* Operation has not yet completed                */
    H5ES_STATUS_SUCCEED,     /* Operation has completed, successfully          */
    H5ES_STATUS_FAIL,        /* Operation has completed, but failed            */
    H5ES_STATUS_CANCELED     /* Operation has not completed and was canceled   */
} H5ES_status_t;
```

Arguments:

req	(IN): The async request on which to wait.
timeout	(IN): The timeout value.
status	(IN): The status.

3.11.2 request: notify

Registers a user callback to be invoked when an asynchronous operation completes. Releases the request if connector callback succeeds.

Signature:

```
herr_t (*notify)(void *req, H5VL_request_notify_t cb, void *ctx);
```

The cb argument is a function pointer:

```
typedef herr_t (*H5VL_request_notify_t)(void *ctx, H5ES_status_t status);
```

Arguments:

req	(IN): The async request that will receive the notify callback.
cb	(IN): The notify callback for the request.
ctx	(IN): The request's context.

3.11.3 request: cancel

Cancels an asynchronous operation. Releases the request if connector callback succeeds.

Signature:

```
herr_t (*cancel)(void *req);
```

Arguments:

req	(IN): The async request to be cancelled.

3.11.4 request: specific

Perform a specific operation on an asynchronous request.

Signature:

```
herr_t (*specific)(void *req, H5VL_request_specific_t specific_type, va_list arguments);
```

The specific_type argument is an enum:

```
/* types for async request SPECIFIC callback */
typedef enum H5VL_request_specific_t {
    H5VL_REQUEST_WAITANY,           /* Wait until any request completes */
    H5VL_REQUEST_WAITSOME,          /* Wait until at least one requesst completes */
    H5VL_REQUEST_WAITALL            /* Wait until all requests complete */
} H5VL_request_specific_t;
```

Arguments:

req	(IN): The async request on which to perform the operation.
specific_type	(IN): The specific operation to perform.
arguments	(IN/OUT): argument list containing parameters and output pointers for the operation.

3.11.5 request: optional

Perform a connector-specific operation for a request.

Signature:

```
herr_t (*optional)(void *req, va_list arguments);
```

Arguments:

req	(IN): The async request on which to perform the operation.
arguments	(IN/OUT): argument list containing parameters and output pointers for the operation.

3.11.6 request: free

Frees an asynchronous request.

Signature:

```
herr_t (*free)(void *req);
```

Arguments:

req	(IN): The async request to be freed.

3.12 Blob Callbacks

```
typedef struct H5VL_blob_class_t {
    herr_t (*put)(void *obj, const void *buf, size_t size, void *blob_id, void *ctx);
    herr_t (*get)(void *obj, const void *blob_id, void *buf, size_t size, void *ctx);
    herr_t (*specific)(void *obj, void *blob_id, H5VL_blob_specific_t specific_type, va_list
        arguments);
    herr_t (*optional)(void *obj, void *blob_id, va_list arguments);
} H5VL_blob_class_t;
```

Listing 13: Structure for blob callback routines, H5VLconnector.h

3.12.1 blob: put

Put a blob through the VOL.

Signature:

```
herr_t (*put)(void *obj, const void *buf, size_t size, void *blob_id, void *ctx);
```

Arguments:

obj	(IN): Pointer to the blob container.
buf	(IN): Pointer to the blob.
size	(IN): Size of the blob.
blob_id	(OUT): Pointer to the blob's connector-specific ID.
ctx	(IN): Connector-specific blob context.

3.12.2 blob: get

Get a blob through the VOL.

Signature:

```
herr_t (*get)(void *obj, const void *blob_id, void *buf, size_t size, void *ctx);
```

Arguments:

obj	(IN): Pointer to the blob container.
blob_id	(IN): Pointer to the blob's connector-specific ID.
buf	(IN/OUT): Pointer to the blob.
size	(IN): Size of the blob.
ctx	(IN): Connector-specific blob context.

3.12.3 blob: specific

Perform a defined operation on a blob via the VOL.

Signature:

```
herr_t (*specific)(void *obj, void *blob_id, H5VL_blob_specific_t specific_type, va_list
    arguments);
```

The `specific_type` argument is an `enum`:

```
typedef enum H5VL_blob_specific_t {
    H5VL_BLOB_DELETE,               /* Delete a blob (by ID) */
    H5VL_BLOB_GETSIZE,              /* Get size of blob */
    H5VL_BLOB_ISNULL,               /* Check if a blob ID is "null" */
    H5VL_BLOB_SETNULL               /* Set a blob ID to the connector's "null" blob ID value */
} H5VL_blob_specific_t;
```

Arguments:

`obj`	(IN): Pointer to the blob container.
`blob_id`	(IN): Pointer to the blob's connector-specific ID.
`specific_type`	(IN): The operation to perform.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the operation.

3.12.4 blob: optional

Perform a connector-specific operation on a blob via the VOL.

Signature:

```
    herr_t (*optional)(void *obj, void *blob_id, va_list arguments);
```

Arguments:

`obj`	(IN): Pointer to the blob container.
`blob_id`	(IN): Pointer to the blob's connector-specific ID.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the operation.

3.13 Optional Generic Callback

A generic optional callback is provided for services that are specific to a connector.

The `optional` callback has the following definition. It returns an `herr_t` indicating success or failure.

Signature:

```
    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req, va_list arguments);
```

Arguments:

`obj`	(IN): The container or object where the operation needs to happen.
`dxpl_id`	(IN): The data transfer property list.
`req`	(IN/OUT): A pointer to the asynchronous request of the operation created by the connector.
`arguments`	(IN/OUT): argument list containing parameters and output pointers for the operation.

4 New VOL API Routines

New API routines have been added to the HDF5 library to manage VOL connectors. This section details each new API call and explains their intended usage.

Additionally, a set of API calls that map directly to the VOL callbacks themselves have been added to aid in the development of passthrough connectors which can be stacked and/or split. For more information on usage refer to Section **??**. The routines are listed in Appendix B.

4.1 H5VLpublic.h

4.1.1 H5VLregister_connector

Signature:

```
hid_t H5VLregister_connector(const H5VL_class_t *cls, hid_t vipl_id);
```

Arguments:

cls	(IN): A pointer to the connector structure to register.
vipl_id	(IN): An ID for a VOL initialization property list (vipl).

Registers a user-defined VOL connector with the HDF5 library and returns an identifier for that connector (H5I_INVALID_HID on errors). This function is used when the application has direct access to the connector it wants to use and is able to obtain a pointer for the connector structure to pass to the HDF5 library.

4.1.2 H5VLregister_connector_by_name

Signature:

```
hid_t H5VLregister_by_name(const char *connector_name, hid_t vipl_id);
```

Arguments:

name	(IN): The connector name to search for and register.
vipl_id	(IN): An ID for a VOL initialization property list (vipl).

Registers a VOL connector with the HDF5 library given the name of the connector and returns an identifier for it (H5I_INVALID_HID on errors). If the connector is already registered, the library will create a new identifier for it and returns it to the user; otherwise the library will search the plugin path for a connector of that name, loading and registering it, returning an ID for it, if found. See the VOL User Guide for more information on loading plugins and the search paths.

4.1.3 H5VLregister_connector_by_value

Signature:

```
hid_t H5VLregister_by_value(H5VL_class_value_t connector_value, hid_t vipl_id);
```

Arguments:

connector_value	(IN): The connector value to search for and register.
vipl_id	(IN): An ID for a VOL initialization property list (vipl).

Registers a VOL connector with the HDF5 library given a value for the connector and returns an identifier for it (H5I_INVALID_HID on errors). If the connector is already registered, the library will create a new identifier for it and returns it to the user; otherwise the library will search the plugin path for a connector of that name, loading and registering it, returning an ID for it, if found. See the VOL User Guide for more information on loading plugins and the search paths.

4.1.4 H5VLis_connector_registered

Signature:

```
htri_t H5VLis_connector_registered(const char *name);
```

Arguments:
`name` (IN): The connector name to check for.

Checks if a VOL connector is registered with the library given the connector name and returns TRUE/FALSE on success, otherwise it returns a negative value.

4.1.5 H5VLget_connector_id

Signature:

```
hid_t H5VLget_connector_id(const char *name);
```

Arguments:
`name` (IN): The connector name to check for.

Given a connector name that is registered with the library, this function returns an identifier for the connector. If the connector is not registered with the library, a negative value is returned. The identifier must be released with a call to `H5VLclose()`.

4.1.6 H5VLget_connector_name

Signature:

```
ssize_t H5VLget_connector_name(hid_t id, char *name/*out*/, size_t size);
```

Arguments:
`id` (IN): The object identifier to check.
`name` (OUT): Buffer pointer to put the connector name. If NULL, the library just returns the size required to store the connector name.
`size` (IN): the size of the passed in buffer.

Retrieves the name of a VOL connector given an object identifier that was created/opened with it. On success, the name length is returned.

4.1.7 H5VLclose

Signature:

```
herr_t H5VLclose(hid_t connector_id);
```

Arguments:
`connector_id` (IN): A valid identifier of the connector to close.

Shuts down access to the connector that the identifier points to and release resources associated with it.

4.1.8 H5VLunregister_connector

Signature:

```
herr_t H5VLunregister(hid_t connector_id);
```

Arguments:
`connector_id` (IN): A valid identifier of the connector to unregister.

Unregisters a connector from the library and return a positive value on success otherwise return a negative value. That native VOL connector cannot be unregistered (this will return a negative herr_t value).

4.2 H5VLconnector.h

4.2.1 H5VLobject_register

Signature:

```
1  hid_t H5VLobject_register(void *obj, H5I_type_t obj_type, hid_t connector_id);
```

Arguments:

obj	(IN): pointer to the object to register.
obj_type	(IN): The type of the object (H5I_FILE, H5I_DATASET, H5I_GROUP, H5I_ATTRIBUTE, H5I_DATATYPE).
connector_id	(IN): identifier of the vol connector the object is associated with.

Creates an HDF5 identifier for an object given its type and an identifier of the connector that was used in creating it. This function is typically used by external connectors wanting to wrap an hid_t around a connector specific object to return to a user callback for functions like H5Literate, H5Lvisit, H5Ovisit, H5Aiterate etc... A Negative value is returned on failure.

4.2.2 H5VLget_object

Signature:

```
1  herr_t H5VLget_object(hid_t obj_id, void **obj);
```

Arguments:

obj_id	(IN): identifier of the object to dereference.
obj	(OUT): buffer to store the pointer of the VOL object associated with the object identifier.

Retrieves a pointer to the VOL object from an HDF5 file or object identifier.

Appendix A Mapping of VOL Callbacks to HDF5 API Calls

VOL Callback	HDF5 API Call
FILE	
create	H5Fcreate
open	H5Fopen
get	H5Fget_access_plist H5Fget_create_plist H5Fget_fileno H5Fget_intent H5Fget_name H5Fget_obj_count H5Fget_obj_ids
specific	H5Fdelete H5Fflush H5Fis_accessible H5Fis_hdf5 (deprecated, hard-coded to use native connector) H5Fmount H5Freopen H5Funmount
close	H5Fclose
GROUP	
create	H5Gcreate1 (deprecated) H5Gcreate2 H5Gcreate_anon
open	H5Gopen1 (deprecated) H5Gopen2
get	H5Gget_create_plist H5Gget_info H5Gget_info_by_idx H5Gget_info_by_name H5Gget_num_objs (deprecated)
specific	H5Gflush H5Grefresh
close	H5Gclose
DATASET	
create	H5Dcreate1 (deprecated) H5Dcreate2
open	H5Dopen1 (deprecated) H5Dopen2
read	H5Dread
write	H5Dwrite
get	H5Dget_access_plist H5Dget_create_plist H5Dextend H5Dget_offset H5Dget_space H5Dget_space_status H5Dget_storage_size H5Dget_type
specific	H5Dextend (deprecated) H5Dflush H5Drefresh H5Dset_extent
close	H5Dclose
OBJECT	
open	H5Oopen H5Oopen_by_addr H5Oopen_by_idx

	H5Oopen_by_name
copy	H5Ocopy
get	N/A
specific	H5Odecr_refcount H5Oexists_by_name H5Oflush H5O_incr_refcount H5Orefresh H5Ovisit_by_name1 (deprecated) H5Ovisit_by_name2 H5Ovisit1 (deprecated) H5Ovisit2
close	H5Oclose
LINK	
create	H5Glink (deprecated) H5Glink2 (deprecated) H5Lcreate_hard H5Lcreate_soft H5Lcreate_ud H5Olink
copy	H5Lcopy
move	H5Gmove (deprecated) H5Gmove2 (deprecated) H5Lmove
get	H5Gget_linkval (deprecated) H5Lget_info H5Lget_info_by_idx H5Lget_name_by_idx H5Lget_val H5Lget_val_by_idx
specific	H5Gunlink (deprecated) H5Ldelete H5Ldelete_by_idx H5Lexists H5Literate H5Literate_by_name H5Lvisit H5Lvisit_by_name
DATATYPE	
commit	H5Tcommit1 (deprecated) H5Tcommit2 H5Tcommit_anon
open	H5Topen1 (deprecated) H5Topen2
get	H5Tget_create_plist
specific	H5Tflush H5Trefresh
close	H5Tclose
ATTRIBUTE	
create	H5Acreate1 (deprecated) H5Acreate2 H5Acreate_by_name
open	H5Aopen H5Aopen_by_idx H5Aopen_by_name H5Aopen_idx (deprecated) H5Aopen_name (deprecated)
read	H5Aread
write	H5Awrite
get	H5Aget_get_create_plist

	H5Aget_info H5Aget_info_by_idx H5Aget_info_by_name H5Aget_name H5Aget_name_by_idx H5Aget_space H5Aget_storage_size H5Aget_type
specific	H5Adelete H5Adelete_by_idx H5Adelete_by_name H5Aexists H5Aexists_by_name H5Aiterate1 (deprecated) H5Aiterate2 H5Aiterate_by_name H5Arename H5Arename_by_name
close	H5Aclose

Table 1: Breakdown of HDF5 API calls by VOL callback

Appendix B Callback Wrapper API Calls for Passthrough Connector Authors

From H5VL_connector_passthru.h

```
1
2  /* Helper routines for VOL connector authors */
3  herr_t H5VLcmp_connector_cls(int *cmp, hid_t connector_id1, hid_t connector_id2);
4  hid_t H5VLwrap_register(void *obj, H5I_type_t type);
5  herr_t H5VLretrieve_lib_state(void **state);
6  herr_t H5VLrestore_lib_state(const void *state);
7  herr_t H5VLreset_lib_state(void);
8  herr_t H5VLfree_lib_state(void *state);
9
10 /* Pass-through callbacks */
11 void *H5VLget_object(void *obj, hid_t connector_id);
12 herr_t H5VLget_wrap_ctx(void *obj, hid_t connector_id, void **wrap_ctx);
13 void *H5VLwrap_object(void *obj, H5I_type_t obj_type, hid_t connector_id, void *wrap_ctx);
14 void *H5VLunwrap_object(void *obj, hid_t connector_id);
15 herr_t H5VLfree_wrap_ctx(void *wrap_ctx, hid_t connector_id);
16
17 /* Public wrappers for generic callbacks */
18 herr_t H5VLinitialize(hid_t connector_id, hid_t vipl_id);
19 herr_t H5VLterminate(hid_t connector_id);
20 herr_t H5VLget_cap_flags(hid_t connector_id, unsigned *cap_flags);
21 herr_t H5VLget_value(hid_t connector_id, H5VL_class_value_t *conn_value);
22
23 /* Public wrappers for info fields and callbacks */
24 herr_t H5VLcopy_connector_info(hid_t connector_id, void **dst_vol_info, void *src_vol_info);
25 herr_t H5VLcmp_connector_info(int *cmp, hid_t connector_id, const void *info1,
26     const void *info2);
27 herr_t H5VLfree_connector_info(hid_t connector_id, void *vol_info);
28 herr_t H5VLconnector_info_to_str(const void *info, hid_t connector_id, char **str);
29 herr_t H5VLconnector_str_to_info(const char *str, hid_t connector_id, void **info);
30
31 /* Public wrappers for attribute callbacks */
32 void *H5VLattr_create(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const
       char *attr_name, hid_t type_id, hid_t space_id, hid_t acpl_id, hid_t aapl_id, hid_t dxpl_id,
       void **req);
33 void *H5VLattr_open(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const char
       *name, hid_t aapl_id, hid_t dxpl_id, void **req);
34 herr_t H5VLattr_read(void *attr, hid_t connector_id, hid_t dtype_id, void *buf, hid_t dxpl_id,
       void **req);
35 herr_t H5VLattr_write(void *attr, hid_t connector_id, hid_t dtype_id, const void *buf, hid_t
       dxpl_id, void **req);
36 herr_t H5VLattr_get(void *obj, hid_t connector_id, H5VL_attr_get_t get_type, hid_t dxpl_id, void
       **req, va_list arguments);
37 herr_t H5VLattr_specific(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       H5VL_attr_specific_t specific_type, hid_t dxpl_id, void **req, va_list arguments);
38 herr_t H5VLattr_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
       arguments);
39 herr_t H5VLattr_close(void *attr, hid_t connector_id, hid_t dxpl_id, void **req);
40
41 /* Public wrappers for dataset callbacks */
42 void *H5VLdataset_create(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const
       char *name, hid_t lcpl_id, hid_t type_id, hid_t space_id, hid_t dcpl_id, hid_t dapl_id, hid_t
       dxpl_id, void **req);
43 void *H5VLdataset_open(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const
       char *name, hid_t dapl_id, hid_t dxpl_id, void **req);
44 herr_t H5VLdataset_read(void *dset, hid_t connector_id, hid_t mem_type_id, hid_t mem_space_id,
       hid_t file_space_id, hid_t plist_id, void *buf, void **req);
45 herr_t H5VLdataset_write(void *dset, hid_t connector_id, hid_t mem_type_id, hid_t mem_space_id,
       hid_t file_space_id, hid_t plist_id, const void *buf, void **req);
46 herr_t H5VLdataset_get(void *dset, hid_t connector_id, H5VL_dataset_get_t get_type, hid_t dxpl_id,
       void **req, va_list arguments);
```

```
47 herr_t H5VLdataset_specific(void *obj, hid_t connector_id, H5VL_dataset_specific_t specific_type,
       hid_t dxpl_id, void **req, va_list arguments);
48 herr_t H5VLdataset_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
       arguments);
49 herr_t H5VLdataset_close(void *dset, hid_t connector_id, hid_t dxpl_id, void **req);
50
51 /* Public wrappers for file callbacks */
52 void *H5VLfile_create(const char *name, unsigned flags, hid_t fcpl_id, hid_t fapl_id, hid_t
       dxpl_id, void **req);
53 void *H5VLfile_open(const char *name, unsigned flags, hid_t fapl_id, hid_t dxpl_id, void **req);
54 herr_t H5VLfile_get(void *file, hid_t connector_id, H5VL_file_get_t get_type, hid_t dxpl_id, void
       **req, va_list arguments);
55 herr_t H5VLfile_specific(void *obj, hid_t connector_id, H5VL_file_specific_t specific_type, hid_t
       dxpl_id, void **req, va_list arguments);
56 herr_t H5VLfile_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
       arguments);
57 herr_t H5VLfile_close(void *file, hid_t connector_id, hid_t dxpl_id, void **req);
58
59 /* Public wrappers for group callbacks */
60 void *H5VLgroup_create(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const
       char *name, hid_t lcpl_id, hid_t gcpl_id, hid_t gapl_id, hid_t dxpl_id, void **req);
61 void *H5VLgroup_open(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const
       char *name, hid_t gapl_id, hid_t dxpl_id, void **req);
62 herr_t H5VLgroup_get(void *obj, hid_t connector_id, H5VL_group_get_t get_type, hid_t dxpl_id, void
       **req, va_list arguments);
63 herr_t H5VLgroup_specific(void *obj, hid_t connector_id, H5VL_group_specific_t specific_type,
       hid_t dxpl_id, void **req, va_list arguments);
64 herr_t H5VLgroup_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
       arguments);
65 herr_t H5VLgroup_close(void *grp, hid_t connector_id, hid_t dxpl_id, void **req);
66
67 /* Public wrappers for link callbacks */
68 herr_t H5VLlink_create(H5VL_link_create_type_t create_type, void *obj, const H5VL_loc_params_t
       *loc_params, hid_t connector_id, hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void **req,
       va_list arguments);
69 herr_t H5VLlink_copy(void *src_obj, const H5VL_loc_params_t *loc_params1, void *dst_obj, const
       H5VL_loc_params_t *loc_params2, hid_t connector_id, hid_t lcpl_id, hid_t lapl_id, hid_t
       dxpl_id, void **req);
70 herr_t H5VLlink_move(void *src_obj, const H5VL_loc_params_t *loc_params1, void *dst_obj, const
       H5VL_loc_params_t *loc_params2, hid_t connector_id, hid_t lcpl_id, hid_t lapl_id, hid_t
       dxpl_id, void **req);
71 herr_t H5VLlink_get(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       H5VL_link_get_t get_type, hid_t dxpl_id, void **req, va_list arguments);
72 herr_t H5VLlink_specific(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       H5VL_link_specific_t specific_type, hid_t dxpl_id, void **req, va_list arguments);
73 herr_t H5VLlink_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
       arguments);
74
75 /* Public wrappers for object callbacks */
76 void *H5VLobject_open(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       H5I_type_t *opened_type, hid_t dxpl_id, void **req);
77 herr_t H5VLobject_copy(void *src_obj, const H5VL_loc_params_t *loc_params1, const char *src_name,
       void *dst_obj, const H5VL_loc_params_t *loc_params2, const char *dst_name, hid_t connector_id,
       hid_t ocpypl_id, hid_t lcpl_id, hid_t dxpl_id, void **req);
78 herr_t H5VLobject_get(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       H5VL_object_get_t get_type, hid_t dxpl_id, void **req, va_list arguments);
79 herr_t H5VLobject_specific(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       H5VL_object_specific_t specific_type, hid_t dxpl_id, void **req, va_list arguments);
80 herr_t H5VLobject_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
       arguments);
81
82 /* Public wrappers for named datatype callbacks */
83 void *H5VLdatatype_commit(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id,
       const char *name, hid_t type_id, hid_t lcpl_id, hid_t tcpl_id, hid_t tapl_id, hid_t dxpl_id,
       void **req);
84 void *H5VLdatatype_open(void *obj, const H5VL_loc_params_t *loc_params, hid_t connector_id, const
```

```
    char *name, hid_t tapl_id, hid_t dxpl_id, void **req);
herr_t H5VLdatatype_get(void *dt, hid_t connector_id, H5VL_datatype_get_t get_type, hid_t dxpl_id,
    void **req, va_list arguments);
herr_t H5VLdatatype_specific(void *obj, hid_t connector_id, H5VL_datatype_specific_t
    specific_type, hid_t dxpl_id, void **req, va_list arguments);
herr_t H5VLdatatype_optional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list
    arguments);
herr_t H5VLdatatype_close(void *dt, hid_t connector_id, hid_t dxpl_id, void **req);

/* Public wrappers for asynchronous request callbacks */
herr_t H5VLrequest_wait(void *req, hid_t connector_id, uint64_t timeout, H5ES_status_t *status);
herr_t H5VLrequest_notify(void *req, hid_t connector_id, H5VL_request_notify_t cb, void *ctx);
herr_t H5VLrequest_cancel(void *req, hid_t connector_id);
herr_t H5VLrequest_specific(void *req, hid_t connector_id, H5VL_request_specific_t specific_type,
    va_list arguments);
herr_t H5VLrequest_optional(void *req, hid_t connector_id, va_list arguments);
herr_t H5VLrequest_free(void *req, hid_t connector_id);

/* Public wrappers for blob callbacks */
herr_t H5VLblob_put(void *obj, hid_t connector_id, const void *buf, size_t size, void *blob_id,
    void *ctx);
herr_t H5VLblob_get(void *obj, hid_t connector_id, const void *blob_id, void *buf, size_t size,
    void *ctx);
herr_t H5VLblob_specific(void *obj, hid_t connector_id, void *blob_id, H5VL_blob_specific_t
    specific_type, va_list arguments);

/* Public wrappers for generic 'optional' callback */
herr_t H5VLoptional(void *obj, hid_t connector_id, hid_t dxpl_id, void **req, va_list arguments);
```

References