RFC: A Plugin Interface for HDF5 Virtual File Drivers

Jake Smith

This RFC proposes a general-purpose API to load and run HDF5 Virtual File Drivers (VFDs) dynamically at runtime. The potential benefits include: “hot-swappable” VFD updates and releases, and ability to release VFDs in binary form. The VFD plugin API will make use of existing Plugin architecture in the HDF5 library.

# Introduction

We want to:

* Simplify the release and maintenance of VFDs.
  + Exotic testing setups not required by VFD non-users, e.g., Hadoop Distributed File System.
* Sell proprietary VFDs – source kept hidden, binaries distributed.
* Maintain binary compatibility between the library and releases of, or updates to, VFDs.
  + An update to a VFD does not require a rebuild of the library.
  + TODO: Replacing a VFD plugin might be “hot-swappable” while the library is in use?
* Keep tools working comfortably with or without any given VFD plugin.
  + TODO: What about help message and accepted arguments?
* Accommodate arbitrary "passthrough" VFD stacking, including both built-in and pluggable VFDs.

To do it, we will:

* Extend the existing Plugin API (H5PL) to support dynamically-loadable VFD plugins.
  + TODO: no new function calls -- all internal details?
* Extend the Property List API (H5P) to provide general-purpose VFD plugin selection and configuration routines.
  + H5Pget/set\_fapl\_vfd\_plugin()
    - Uniform interface for VFD plugin FAPL-set.
    - Receives a string containing configuration information, the contents of which are interpreted as appropriate by the VFD plugin.
  + H5Pget/set\_fapl\_vfd() as a wrapper for \_all\_ VFD FAPL-set calls, including built-in VFDs.
  + When using passthrough plugin VFDs, the user will no longer be responsible for child VFD/FAPL management.
    - The "top level" VFD plugin is configured via the configuration string which contains the configuration information of all (nested) child VFDs. Upon VFD-open, this child configuration information is used to instantiate the child VFD/FAPLs; on VFD-close, these child VFD/FAPLs are also closed and released recursively.
    - Child VFD configuration string will be passed into the new API call, H5Pset\_fapl\_vfd(), which will parse and redirect the configuration information appropriately.
    - Built-in VFDs may still be configured with their dedicated FAPL-set functions (e.g., H5Pset\_fapl\_sec2()).
    - Built-in passthrough VFDs may still be populated with user-created child VFD/FAPL IDs (e.g., application/user creates child VFD/FAPLs and passes them into the passthrough VFD's dedicated FAPL-set call as appropriate).
    - Any plugin passthrough VFD which might use a built-in passthrough VFD *must* have a provision for handling the built-in VFD's FAPL-set, managing its child VFD/FAPLs.
      * The more elegant solution is to update the built-in passthrough VFD to a plugin.
* Extend the Virtual File Layer API (H5FD) to manage VFD plugins in memory.
  + TODO: this will probably be invisible to the user. “Cache” loaded VFD plugins, like H5Z?
* Create an example terminal VFD plugin for testing and demonstration (e.g., *sec2plugin* or *stdioplugin*).
* Create an example passthrough (or, “stackable”) VFD plugin (e.g., *splitter* or *logger*).

# Approach

* Add a new public property list (H5P) functions:
  + herr\_t H5Pset\_fapl\_vfd\_plugin(hid\_t fapl\_id, const char \*plugin\_name, const void \*fapl\_config);
    - When called, sets FAPL to use VFD plugin “class” with configuration data (if any). Plugin is responsible for discerning meaning from the configuration pointer [could be a string or a pointer to a structure]. Plugin reference count is incremented.
    - If plugin is not already registered with the library, attempt to load and register plugin. TODO: initialization.
  + herr\_t H5Pget\_fapl\_vfd\_plugin(hid\_t fapl\_id, const char \*plugin\_name, void \*fapl\_config\_out);
    - Retrieves the VFD-specific configuration information stored in the FAPL.
    - Assumes that the plugin is registered; will fail if not.
  + herr\_t H5Pset\_fapl\_vfd(hid\_t fapl\_id, const char \*name, const char \*fapl\_config);
    - Wraps configuration strings for any VFD. If the VFD name is found as built-in, parses the string and calls FAPL-set function for that VFD. If not found as a built-in, calls H5Pset\_fapl\_vfd\_plugin().
  + herr\_t H5Pget\_fapl\_vfd(hid\_t fapl\_id, const char \*name, const char \*fapl\_config);
    - Retrieves VFD-specific configuration data stored in the FAPL; extracts and formats info from built-in VFDs if appropriate.
    - Included on general principle. It remains unclear how useful this will actually be, but might be relevant with nested built-in/pluggable passthrough VFDs.
* VFD plugins must implement from *H5PLextern.h*:
  + H5PL\_type\_t H5PLget\_plugin\_type(void);
    - Must return the type enumeration from *H5PLpublic.h* (H5PL\_TYPE\_VFD, e.g.)
  + const void \* H5PLget\_plugin\_info(void);
    - Returns the “class” structure of the VFD.
    - TODO: Or an “info” wrapper/subclass of said class.
* When a property list or file is closed which uses a VFD plugin, the reference count for that plugin is decremented.
  + When reference count reaches zero, the VFD is unregistered (no modifications needed) and TODO unloaded(?)... in part using the VFD's *terminate* callback.

Possible order of operations:

* A FAPL is set to use a VFD plugin through H5Pset\_fapl\_vfd\_plugin().
  + If the plugin is not already available (registered):
    - If the VFD plugin is not already loaded/cached, H5PL attempts to locate and load the plugin.
    - If loaded successfully, the VFD plugin is registered with the library (through H5I).
  + Reference count of the VFD plugin is incremented (happens automatically with register?).
  + Configuration information, if any, is copied into the FAPL.
* FAPL is used to open files.
  + VFD plugin reference count is incremented with the creation of the “virtual file” in the VFD’s open call. (TODO: confirm)
* FAPL or file (opened with FAPL/VFD plugin) is closed with H5Pclose().
  + Reference count of the VFD plugin is decremented.
  + If reference count is zero, the VFD is unregistered [and unloaded?] from the library (H5I, H5PL). Should ONLY occur when all FAPLs and virtual files are closed.
  + TODO: Can the VFD’s `terminate` function/callback properly unregister the VFD?

Possible complications:

* Getting information about the features supported by VFD plugin from within the application or library, e.g., “is this driver read-only?”
  + Library: File access support is of concern only on file open – if supported, open succeeds; else, fails. Repeated support checks are not required.
  + Might provide a wrapper structure that contains operational information alongside the actual VFD “class” that the library uses (stored in the FAPL), e.g.:

typedef struct H5FD\_plugin\_info\_t {

unsigned version; /\* informs component membership of this struct \*/

hid\_t reg\_id; /\* reserved for in-library use after registration? \*/

[. . .] /\* other pieces of information \*/

H5FD\_class\_t clas; /\* actual VFD “class” \*/

} H5FD\_plugin\_info\_t;

* Handling calls to unsupported features gracefully?
  + H5FDquery\_driver() is a good start, returning a bitmask integer value.
    - May want to add Read-Only -- or generally file access -- support flag?
* Tools’ help messages and command-line arguments.
  + Command-line VFD configuration information passed into H5Pset\_fapl\_vfd\_plugin() as string vs structure.
* It isn’t obvious how to handle VFD plugin “IDs” (generated upon registration) with “names” (used to identify/locate the plugin in the first place). Hashing, or internal map somewhere? Make this part of the H5FD extension that mimics H5Z caching?
* Can the VFD plugins interact correctly with the HDF5 error stack? HGOTO\_ERROR() and its kind are *private*. Do we care?
  + There are work-arounds with a public API to push errors onto the stack.

Possible examples of a VFD configuration string in different formats – *splitter* VFD (passthrough plugin) nesting *core* (terminal built-in) and *mirror* (terminal plugin) VFDs, with the *mirror* plugin file potentially being at an unorthodox location:

* JSON:

{

"rw-vfd": {

"name": "core",

"configuration": {

"increment": 4096,

"page-backing": false,

"write-tracking-enabled": true,

"page-size": 1048576,

},

},

"wo-vfd": {

"name": "/path/to/plugins/mirror",

"configuration": {

"handshake-port", 3000,

"server-IP": "127.0.0.12",

},

},

"wo-target": "/path/to/myfile\_writeonly.h5",

"logfile": "/path/to/splitter\_errlog.txt",

"ignore-write-channel-errors": true,

}

* Tuple-based (example *splitter* config?):

((core,(4096,0,1,1048576)),(/path/to/plugins/mirror,(3000,127.0.0.12)),/path/to/myfile\_writeonly.h5,/path/to/splitter\_errlog.txt,1)

* Tuple-based file drop-in to a tool, e.g.:

(myfile.h5, (vfd, (ros3, ()))) # uses “default” ros3 VFD to open “myfile”

Selection and setting of a VFD is done through a File Access Property List (FAPL), which is passed to relevant operations such as H5Fopen(). The configuration details of the VFD, if relevant, are set at runtime with a FAPL-specific property list set function H5Pset\_fapl\_<vfd>(). Below is a demonstration of the *sec2* file driver, a POSIX-friendly default that is built into the library – no configuration information is expected by the driver, so H5Pset\_fapl\_sec2() receives only the FAPL ID.

hid\_t fapl\_id = -1;

hid\_t file\_id = -1;

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

assert(fapl\_id > -1);

assert(H5Pset\_fapl\_sec2(fapl\_id) > -1);

file\_id = H5Fopen(“filename.h5”, H5F\_ACC\_RDWR, fapl\_id);

assert(file\_id > -1);

assert(H5Pclose(fapl\_id) > -1); /\* no longer needed; dispose \*/

[...] /\* use file \*/

assert(H5Fclose(file\_id) > -1);

Below, contrast the signature of H5Pset\_fapl\_sec2(), which requires no configuration, with that of H5Pset\_fapl\_family(), which does. The *family* file driver splits a single logical file into several smaller “file members” on the system storage – useful for storing files which exceed the filesystem’s maximum size of files on disk.

herr\_t H5Pset\_fapl\_sec2(hid\_t fapl\_id)

herr\_t H5Pset\_fapl\_family(hid\_t fapl\_id, hsize\_t msize, hid\_t memb\_fapl\_id)

TODO: something about parameter list vs a structure pointer [vs formatted string] to pass configuration information to and from H5Pget/set\_fapl\_<vfd>().

When set in a FAPL, the VFD is loaded and registered <if necessary?> and initialized; its registered value and its “class” – a structure containing pointers to supported data and function callbacks – are stored in the FAPL, to be accessed and used as needed by H5F, H5FD, and other relevant API calls. Any configuration information for the VFD is also copied into the FAPL.

TODO: Something about reference counts, or not relevant to built-in VFDs? With plugins, yes: FAPL-close will reduce reference count, and a plugin can be unloaded (probably) when no longer in use – maybe provide an override for default unloading that the programmer can pass in as a hint, but then must accept responsibility for closing when done (FAPL creation should be infrequent enough that this optimization is very much unnecessary, but that’s an early guess).

## Extension to Library Plugin Code (H5PL)

The plugin interface is well-conceived in its extensibility. We can use it to create an interface for VFD plugins. It asks that a plugin – the current examples being filters and VOL connectors – provide two functions: H5PLget\_plugin\_type() and H5PLget\_plugin\_info(). The former returns an enumerated value to indicate to the library how it is to be used; the latter returns a structure, or “class”.

A new “VFD plugin class info” structure will be created, to encapsulate a VFD's “class” structure and a version number of the info structure to guard against modifications in the future.

typedef struct {

int32\_t version; /\* informs struct membership \*/

H5FD\_class\_t driver; /\* actual VFD "class" \*/

} H5FD\_plugin\_info\_t;

A new enumerated value, H5PL\_TYPE\_VFD, will be added to the list in H5PLpublic.h, which must be returned by a VFD plugin’s H5PLget\_plugin\_type() function.

typedef enum H5PL\_type\_t {

H5PL\_TYPE\_ERROR =-1;

H5PL\_TYPE\_FILTER = 0;

H5PL\_TYPE\_VOL = 1;

H5PL\_TYPE\_VFD = 2; /\* ADDED \*/

H5PL\_TYPE\_NONE = 3; /\* This must be last \*/ /\* INCREMENTED \*/

} H5PL\_type\_t;

Also in H5PLpublic.h, a new value must be defined for internal purposes of get/set loading state functions. These functions will be responsible for auto-loading (or not) of plugins automatically upon library startup.

/\* Common dynamic plugin type flags used by the H5PLget/set\_loading\_state functions \*/

#define H5PL\_FILTER\_PLUGIN 0x0001

#define H5PL\_VOL\_PLUGIN 0x0002

#define H5PL\_VFD\_PLUGIN 0x0004 /\* ADDED \*/

#define H5PL\_ALL\_PLUGIN 0xFFFF

In H5PLprivate.h, the H5PL\_key\_t structure will need to be modified to include a string pointer for a VFD “name”.

/\* The key that will be used to find the plugin \*/

typedef union H5PL\_key\_t {

int id; /\* filters \*/

struct {

H5VL\_get\_connector\_kind\_t kind; /\* Kind of VOL lookup to do \*/

union {

H5VL\_class\_value\_t value; /\* VOL connector value \*/

const char \*name; /\* VOL connector name \*/

} u;

} vol;

const char \*vfd\_name; /\* ADDED \*/

} H5PL\_key\_t;

H5PL\_load() and H5PL\_\_open() in *H5PLint.c* will both be modified to handle the new VFD plugin case.

## Definition of Plugin VFDs Within the HDF5 Library

TBD.

Something about softly mapping a VFD “name”, used by the user in H5Pget/set\_fapl\_vfd\_plugin(), e.g., to an internally registered identification number (generated through H5I?) for efficient ID checking.

## VFD Plugin Implementation

TBD. Plugin must link with a relevant version of the library, support the few H5PL external functions, and provide a VFL-compliant “class” structure.

Where the H5PL looks for VFD plugin files remains an open question. It makes sense for VFD plugins to be sorted separately from the filter plugins. but the complexity of implementing multiple locations may be a problem.

## Creating VFD Plugins

TBD

## Loading and Using VFD Plugins

Crate a FAPL and set it with H5Pset\_fapl\_vfd\_plugin(), providing the appropriate VFD name and configuration information. If successful, then proceed with file use and file- and FAPL-close as usual.

# Implementation Details

TBD

# Testing

TBD.

Create a pluggable clone of sec2, or something.

Will use development process to clarify user documentation, possibly leading to an “SDK” for VFD plugins.

# Recommendation

There are so many future plans depending on the ability to release proprietary VFDs that this has to be done eventually, sooner rather than later.

TBD

# Acknowledgements

This work was internally funded by The HDF Group.

Also “ECP”?

# Revision History

|  |  |
| --- | --- |
| *April 10, 2019:* | Version 1 drafted. |

# Appendix: Background Material

TBD

# Glossary

This section is not required, but is highly recommended if there are terms in the RFC that may not be familiar to the readers.

|  |  |
| --- | --- |
| **Child VFD** | VFD which receives direction from a “passthrough” VFD, rather than from the HDF5 library directly. |
| **File Access Property List (FAPL)** | TODO |
| **HDF5 Identifier (HID)** | TODO |
| **Passthrough VFD** | VFD which redirects operations to other “child” VFDs. |
| **Plugin** | Binary code which can be loaded and utilized at runtime. HDF5 examples include filter and VOL plugins. |
| **Terminal VFD** | VFD which performs operations on storage. |
| **Virtual File Driver (VFD)** | TODO |
| **Virtual File Layer (VFL)** | TODO |

# References

TBD