RFC: A Plugin Interface for HDF5 Virtual File Drivers

Jake Smith

There is increasing interest in dynamically-loadable Virtual File Drivers (VFDs) to greatly simplify the release of VFDs, especially proprietary VFDs where the source code should not be publicly visible. This RFC proposes a general-purpose API to load and run VFDs dynamically at runtime.

# Introduction

With the creation of proprietary VFDs, the release pipeline for the HDF5 library became very complicated, involving separate code patches applied to the main release branch and imposing exotic and fickle testing environments. By making VFDs accessible dynamically (“pluggable”), these issues are largely avoided: the same library may use – or not – proprietary VFD plugins released as binaries; these VFD plugin binaries can be tested internally, no longer forcing the end user to accommodate odd testing environments.

Additional benefits include the ability to update VFD plugins without requiring a rebuild of the library.

VFD plugins will be engineered to handle “passthrough VFDs”, which defer or relay operations to one or more “child VFD”.

Tools will also be augmented to interpret VFD plugin configuration and use.

This RFC targets the HDF5 v1.12 release.

# Proposed Changes

## New VFD-related FAPL API Calls

Two new getter/setter functions will be added to the API: H5Pget/set\_fapl\_vfd\_plugin(), and H5Pget/set\_fapl\_vfd(). Both deal with a top-level FAPL id, a string name of a VFD, and a string containing configuration information of the VFD.

### H5Pget/set\_fapl\_vfd\_plugin()

The configuration information pointer will be of type void, as we will not impose that all VFD plugins require a string – creators of VFD plugins may have valid reasons for using some other means of passing in configuration information.

#### H5Pset\_fapl\_vfd\_plugin()

The setter will have the following signature:

herr\_t H5Pset\_fapl\_vfd\_plugin(hid\_t fapl\_id, const char \*name, const void \*config);

Its purpose will be to load the plugin (if not already loaded) through the HDF5 Plugin API (H5PL) and to copy the driver class and its configuration information into the FAPL.

#### H5Pget\_fapl\_vfd\_plugin()

The getter will have the following signature:

herr\_t H5Pget\_fapl\_vfd\_plugin(hid\_t fapl\_id, const char \*name, void \*config\_out);

Its purpose will be to verify that the plugin within the FAPL corresponds with the given plugin name and to retrieve the configuration information as originally set.

### H5Pget/set\_fapl\_vfd()

This new family of calls will relate to the change of responsibility for child VFD configuration from the current convention.

At present, such as with the *multi* or *family* VFDs, the programmer creates the FAPL for the child, terminal VFD, and then sets that FAPL ID in the top-level “parent” VFD. Upon cleanup, the programmer is likewise responsible for closing both FAPLs. This approach is sensible because the programmer, at compile time, is aware of all the relevant VFD configuration calls – H5Pset\_fapl\_<driver>() – and has access to any structure or specially-defined types that a driver’s FAPL-set function requires.

With VFD plugins, these assumptions are completely invalidated. As such, any passthrough VFD must be able to configure, retrieve, and close any child VFD, plugin or otherwise. This generic function will wrap both built-in and plugin VFD FAPL-set calls, to be called as needed as a VFD initializes itself (probably on call to VFD-open).

The user will be responsible only for the top-level FAPL.

Built-in VFDs will (should) still be set primarily with their dedicated FAPL-set function.

Any plugin passthrough VFD which might use a built-in passthrough VFD (e.g., *splitter* VFD plugin using *family* VFD for read-write channel) *must* have a provision for handling the built-in VFD's FAPL-set, managing its child VFD/FAPLs. The more elegant solution will be, if possible, to update the built-in passthrough VFD to a plugin.

#### H5Pset\_fapl\_vfd()

The setter will have the following signature:

herr\_t H5Pset\_fapl\_vfd(hid\_t fapl\_id, const char \*name, const void \*config);

In the case that the name does not match a built-in VFD, it will redirect to H5Pset\_fapl\_vfd\_plugin(). If, instead, the name is known as a built-in VFD, this function will handle any relevant processing of the configuration information and will pass it into that VFD’s FAPL-set function, e.g. H5Pset\_fapl\_core().

#### H5Pget\_fapl\_vfd()

The getter will have the following signature:

herr\_t H5Pget\_fapl\_vfd(hid\_t fapl\_id, const char \*name, void \*config\_out);

The inverse of the setter – will check that the VFD name matches that in the FAPL; will then redirect to the H5Pget\_fapl\_vfd\_plugin() if not recognized, or else get information from built-in VFD’s FAPL-get function and compose it as appropriate.

## VFD Configuration String Syntax

Below is the proposed syntax for VFD configuration strings, and examples with built-in and proposed plugin VFDs, both passthrough and terminal. The top-level parse of VFD\_name, configuration\_value, is not consistent with the FAPL-set functions in 2.1 at this time – this parse would be done prior to H5Pset\_fapl\_vfd[\_plugin](), rather than inside it.

VFD plugins would not be beholden to this syntax, but all VFDs created by THG should be as consistent as possible.

The built-in VFD example configuration strings could be used with the wrapper H5Pset\_fapl\_vfd(), but are provided more as case studies than intended for application.

Examples with “missing” parameters –as seen with *core*, *family*, and *mirror*, below – would use an internally-defined default value.

VFD\_CONFIG ::= KV\_PAIR

KV\_PAIR ::= '(' KEY [ VALUE ] ')' ; matched parentheses enclosing a KEY and an optional VALUE separated by whitespace

KEY ::= identifier

VALUE ::= identifier

| number

| quoted\_string ; "..." Enclosed quotes must be escaped

| KV\_PAIR

| LIST

LIST ::= '(' VALUE { VALUE } ')' ; matched parentheses enclosing VALUEs separated by whitespace

[ ... ] :: 0 or 1 ("optional")

{ ... } :: zero or more

"|" :: "OR"

Examples with current VFDs

==========================

(core ((increment 1048))) ; no page-backing

(core ((backing 1) (increment 1048)))

(core ((increment 1048) (write\_tracking 1) (page\_size 4096)))

(direct ((cbuf\_size 8192) (boundary 512) (block\_size 4096)))

(family ((member\_size 1024))

(family ((terminal\_vfd (sec2)) (member\_size 1024)))

(family

( (terminal\_vfd

(direct

( (cbuf\_size 8192)

(boundary 512)

(block\_size 4096)

)

)

)

(member\_size 1024)

)

)

(log ((logfile log\_vfd\_out.log) (flags 1048575) (buffer\_size 4096)))

; 1: SUPER, 2: BTREE, 3: DRAW, 4: GHEAP, 5:LHEAP, 6:OHDR

; TODO: maxaddr 'k'?

; no specified terminal VFDs for any constituent file -- uses default

(multi

( (memory ((3 3) (2 2) (4 4)))

(names

( (1 "multi\_file-s.h5")

(2 "multi\_file-b.h5")

(4 "multi\_file-g.h5")

(3 "multi\_file-r.h5")

)

)

(maxaddrs ((1 0) (2 k/4) (3 k/2) (4 k\*3/4)))

)

)

; Or, using identifiers in place of enum index

(multi

( (memory ((draw 3) (btree 2) (gheap 4)) )

(names

( (super "multi\_file-s.h5")

(btree "multi\_file-b.h5")

(gheap "multi\_file-g.h5")

(draw "multi\_file-r.h5")

)

)

(maxaddrs ((super 0) (btree k/4) (draw k/2) (gheap k\*3/4)))

)

)

(sec2)

(stdio)

(windows)

Examples with in-progress VFD plugins

=====================================

(ros3) ; anonymous access

(ros3 ((aws\_profile test-hdf5-aws))) ; credentials stored locally

(ros3 ((aws\_region us-east-1) (access\_key\_id "TBD") (secret\_access\_key "TBD")))

(mirror ((ip 127.0.0.12))) ; default port

(mirror ((port 8080) (ip 127.0.0.10)))

(splitter

( (write\_only\_channel (mirror ((ip localhost) (port 3000))))

(read\_write\_channel (sec2))

(logfile splitter.log)

(ignore\_write\_channel\_errors 1)

)

)

(splitter ; "default" r/w channel; no logging; register errors

( (write\_only\_channel (mirror ((ip localhost) (port 3000))))

)

)

## Plugin (H5PL) Augmentations

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 will be defined for internal purposes, used by H5PLget/set\_loading\_state() – these functions are responsible for automatic loading (or not) of plugins by type 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.

## VFL (H5FD) Augmentations

The Plugin-implemented function, H5Pget\_plugin\_info() returns a void pointer– in the case of VFD plugins, it will return a pointer to a “class” containing the VFD’s implementation much in the manner of current, built-in VFD “classes” (H5FD\_class\_t). While the details of this class structure are not precisely relevant to the VFD plugin project, the current implementation is very inflexible, and an update is proposed in the next subsection.

### VFD Class Versioning

At the time of this RFC, the VFD “class” structure has no self-descriptive information beyond its name. This is a serious problem, as it forces that all VFD classes share identical components, even as those components are changed or added to, to meet the needs of a small number of VFDs; any change in the class structure requires that all other VFD implementations accommodate this change.

A solution to this problem is to overhaul the class implementation once with three components at the start of the structure:

typedef struct H5FD\_class\_v0\_t {

int32\_t magic; /\* unique to H5FD\_class\_v\*\_t \*/

int16\_t version\_major; /\* informs expected membership of base class \*/

int16\_t version\_minor; /\* informs which extension of base class \*/

[. . .] /\* membership of the current H5FD\_class\_t \*/

} H5FD\_class\_v0\_t;

The *magic* number will be a shared by (and unique to) all H5FD classes. The *major* version will be used to identify the *base* membership of the class – in this case, the membership of the current H5FD\_class\_t. The *minor* version will be used to inform which subsequent version of class the pointer should be cast.

As a result, subsequent *minor* versions of a class may be created, so long as they only add to the implementation of the previous class minor version. Augmentations that require a re-implementation of extant members will require a new *major* version, which is appropriate with an HDF5 major release/modification anyway.

In use, the returned void pointer is cast to the base class – in this case, H5FD\_class\_v0\_t. The magic number will be checked to guard against an inappropriate casting. The major version will be checked to guard against an outdated implementation. Then the minor version is checked, and, in necessary, the pointer is re-cast to a subsequent class (e.g., H5FD\_class\_v2\_t) prior to use

# Miscellany (cruft, leftovers, placeholders)

* 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:

* Tools’ help messages and command-line arguments.
  + Command-line VFD configuration information passed into H5Pset\_fapl\_vfd\_plugin() as string vs structure.
  + Generic “(vfd\_name <vfd\_specific\_config>)” helper?
* 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.

## 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 plugin clone of terminal VFD, *sec2* or *stdio*, and a passthrough VFD plugin, perhaps *splitter*.

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, 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. |
| *April 24, 2019* | Version 2 updated to prose; restructured; adds notes on configuration and VFD class versioning. |

# 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