RFC: A Plugin Interface for HDF5 Virtual File Drivers

Jake Smith
Jordan Henderson

There is increasing interest in dynamically-loadable Virtual File Drivers (VFDs) to enable easy usage of different VFDs within an HDF5 application without needing code modifications, as well as to greatly simplify the release of VFDs as modules that are separate from the HDF5 library. This RFC proposes a general-purpose API to load and run VFDs dynamically at runtime. In addition, several approaches are proposed for configuring VFD plugins, with the intent that the chosen approach(es) will supersede the existing driver configuration infrastructure.

Contents

1. Introduction

The primary goal for this RFC is to be able to release Virtual File Drivers (VFDs) separately from the HDF5 library. The direct result of this is that HDF5 must support VFDs as dynamically-loadable plugins. The indirect result of this is that the configuration process for a given VFD becomes dynamic as well: driver selection and configuration might no longer be achieved through dedicated API routines available at compile time.

Because any details about the binary representation of a driver's configuration (such as through a structure) cannot be guaranteed at compile time, a mechanism is required to identify and load a driver, and pass in a generalized configuration blob for the individual drivers to use on file-open. This configuration blob will take the form of a string with minimal syntax requirements, with runtime binary data being supported through the use of HDF5's property lists[1]. By HDF5 1.14, string-based configuration will be the only supported method for all VFDs; the current API for VFD selection and configuration will be retained for convenience, supported by the string format internally. For the transition to strings, both binary and string driver configuration internals may be supported concurrently[2].

HDF5's tools will be augmented to accept VFD configuration strings and describe their use. The library and tools should also be able to accept a VFD "default driver" and "default driver configuration" via environment variables or perhaps configuration files.

This RFC targets the HDF5 v1.14 release, with consideration given towards retaining the existing VFD API, primarily for the sake of third-party VFD authors.

[1] MPI Communicators and MPI Info objects are the only known use case of runtime data at this time.

[2] Concurrent binary and string configuration is not ideal, as it increases the short-term complexity of the implementation.

2. Approach

The following sections discuss high-level details about the general approach that will be taken to support the use of VFD plugins in an HDF5 application. Implementation-specific details are outlined in section 3.

2.1. Implementing a VFD plugin

In order to be compatible with HDF5's plugin infrastructure, an external VFD plugin must link with a relevant version of the library, implement the external H5PL functions and provide a VFL-compliant "class" structure. The external H5PL functions to implement are as follows:

```
H5PL_type_t H5PLget_plugin_type(void);
```

This function must return the `H5PL_TYPE_VFD` type enumeration, as defined in H5PLpublic.h.

```
const void * H5PLget_plugin_info(void);
```

This function must return the "class" structure of the VFD, as defined in the VFD's source code.

Any VFD plugins external to HDF5 must also make some effort to inter-operate with the HDF5 error stack using the available public H5E API. Examples of how to do so may be seen in HDF5's DAOS VOL connector.

Finally, any VFD that is to be used as a plugin should provide a reasonable default configuration (if possible) for the case where a user does not explicitly provide one. Doing so helps to make the plugin more compatible with the HDF5 ecosystem, including the HDF5 library, test suite, tools, etc.

All of HDF5's internal VFDs will be reviewed and updated with any modifications needed from above as part of the work for this RFC. Any third-party VFDs that wish to be compatible with HDF5 as a VFD plugin will need to be updated by the driver's author.

2.2. Loading VFD plugins

To load and use a VFD plugin in an HDF5 application, some effort will need to be made by the application or the user in order to tell HDF5 which driver to use and how to have that driver be configured. Below are some proposed approaches to this end; in each case, the VFD plugin loading process will search HDF5's plugin path when trying to find a plugin. More information about HDF5's plugin path can be found at Dynamic Plugins in HDF5.

2.2.1. By FAPL

To support explicit loading of a VFD plugin in an HDF5 application, this RFC proposes that two new `H5P` routines should be added to the library. These routines would allow the application to specify a VFD plugin by name or by ID, while providing the appropriate (and possibly stacked) configuration information. The application would create a FAPL, set the driver and its configuration on the FAPL with one of the new routines, and then pass that FAPL to the `H5Fcreate` or `H5Fopen` call. The proposed signatures for these routines is discussed in 3.2.

There will be major changes to the user's interaction with FAPL driver manipulation and the user's responsibility of child VFD configuration. At present, such as with the multi or family VFDs, the programmer creates the FAPL for any child VFD, and then sets that FAPL ID in the parent VFD. Upon cleanup, the programmer is likewise responsible for closing all FAPLs. This approach is possible because the user, at compile time, is aware of all the relevant VFD configuration calls – e.g., `H5Pset_fapl_<driver>()` – and has access to any structure or specially-defined types that a driver's FAPL driver-set function requires. With VFD plugins, however, neither dedicated FAPL driver-set functions nor configuration information definitions are guaranteed at compile time. As such, any passthrough VFD must be able to configure, retrieve, and close itself and any child VFD, plugin or otherwise, without the user's direct intervention.

A generic FAPL driver-set function must therefore wrap both built-in and plugin VFD FAPL driver-set calls, with a single "stackable" configuration element.

Using the family driver as an example, the programmer would call the generic FAPL driver-set function with a configuration element that includes both the family driver and any child (member) driver information, and the resulting FAPL would be used to create the complete VFD stack on file open. The user will be responsible solely for this "top-level" FAPL. Similarly, the user must be able to use a generic get function and receive a complete configuration element, including any VFD stacking – this should be possible regardless of how the top-level VFD was set. Built-in VFDs may still be set with their dedicated FAPL driver-set function, but the provision must be made for configuring with a string, in order to inter-operate with passthrough VFD plugins. These dedicated FAPL manipulation functions will be maintained as a convenience to the user.

2.2.2. By Environment Variable

For convenience, it is proposed that the `HDF5_DRIVER` environment variable should also be available to specify the name of a VFD plugin[3]. By using an environment variable to specify the VFD name, no code modifications will need to be made to an HDF5 application, allowing one to easily switch between different VFDs before running the application. In the same manner as HDF5's `HDF5_VOL_CONNECTOR` environment variable, setting `HDF5_DRIVER` would replace the default VFD for File Access Property Lists. Thus, any HDF5 application that passes `H5P_DEFAULT` for an HDF5 API's `fapl_id` parameter, or which supplies a FAPL that has not specifically had a VFD set on it, will use the driver specified by this environment variable.

If this environment variable is implemented, one would naturally want a way to specify configuration information for a VFD in a similar manner, so as to avoid needing application code modifications. Provisions for this are discussed in sections 2.3.2 and 2.3.3.

[3] Note that the `HDF5_DRIVER` environment variable already exists in the HDF5 library, but is currently only used for testing code; this proposal would simply re-purpose the environment variable.

2.3. Configuring VFD plugins

The following sections contain details about configuring VFD plugins and discusses the particular uses for each of the proposed approaches that could be implemented. In each case, configuration is done completely via configuration strings; configuring a VFD with binary runtime data is discussed in 2.3.4. Further, covered here is simply a high-level overview of how the configuration string should be passed into HDF5 and, subsequently, the underlying VFD structure. Discussion around the format of these configuration strings can be found in 3.1. While only one of the proposed approaches below must be adopted for the purpose of this feature, it may good to consider implementing more (or all) of these methods in order to give an HDF5 library user more freedom on how to configure a VFD plugin.

2.3.1. By Generic H5P API Routine

As mentioned previously, this RFC proposes that two new `H5P` routines should be added to HDF5 to facilitate specifying a VFD by name or by ID. Those routines would accept a configuration string as a parameter and would be the generally preferred method for passing configuration strings to HDF5. The configuration string would be stored as part of the FAPL specified and would be later retrieved from the FAPL internally.

2.3.2. By Configuration File

Configuration strings might also be passed in via configuration files residing in storage. A configuration file would simply contain the entire configuration string, formatted in the manner decided upon in 3.1. Parsing of a configuration file could either be done by HDF5 itself (which may introduce a library dependency), or could be deferred to the top-level VFD in the VFD stacking structure. To allow HDF5 to locate a configuration file at runtime, several approaches could again be used:

- A standard file name/location pair could be decided upon, which HDF5 would look for at runtime
- A new `H5P` routine could be introduced to point to the file
- An environment variable, `HDF5_DRIVER_CONFIG_FILE` could be used to point to the configuration file

2.3.3. By Environment Variable

For convenience, it is proposed that the environment variable `HDF5_DRIVER_CONFIG` should be available for use. This would contain the entire configuration string needed by the application and would mostly be helpful in simple cases where the string itself is not very large or complicated. This environment variable is meant to serve as a companion to the previously-mentioned `HDF5_DRIVER` environment variable, allowing a compiled HDF5 application to easily switch between different VFDs without needing to be modified.

2.3.4. Storing and Accessing Runtime Configuration Data

As a VFD may need access to information that exists at runtime and cannot be easily encoded as a string, it is important to provide a mechanism to store this runtime information. The primary known use case for storing this kind of opaque runtime data pertains to MPI Communicator and Info objects. To retain compatibility with existing HDF5 applications, MPI Communicator and Info objects should continue to be set on a FAPL within the application. This is done via either the `H5Pset_fapl_mpio()` API routine if the MPI I/O VFD is being used, or via the `H5Pset_mpi_params()` API routine if a different VFD is being used. A VFD may later retrieve the MPI Communicator and Info objects by using the `H5Pget_mpi_params()` API routine. For convenience and simplicity, a VFD plugin may also choose to translate common values from representative values in a given configuration string, e.g. ”MPI_COMM_WORLD”, ”MPI_COMM_SELF”, ”MPI_INFO_NULL”, etc.

Other runtime information that needs to be passed to a VFD should be inserted as a new property on a FAPL via the `H5Pinsert()` API routine. That runtime information can be retrieved later by the VFD via the `H5Pget()` API routine.

2.3.5. Remote Configuration

Any driver that has child operations "over the wire", such as a remote VFD with mirror_sock (a socket-based mirror VFD), will be responsible for communicating with the remote process. This includes transmitting the child configuration on file open, directing remote operations (read/write) throughout the file's lifespan, and handling remote file close. The remote process must be able to respond appropriately to the parent process's instructions.

3. Implementation Details

The following sections contain lower-level details about how support for VFD plugins will be implemented.

3.1. Configuration String Format

Configuration strings must contain all information for a VFD's operation, including the driver's name, its settings and the information needed for any child file driver(s), such as the underlying driver as part of the family or multi drivers. Unlike filters, the primary user of VFD selection will be the user, rather than the library, which puts a premium on the ability of the programmer to recognize and/or remember which driver they intend, and how it operates. Keeping this in mind, an important choice must be made about the format that configuration strings will be expected to conform to.

The first approach would be to adopt an existing configuration format standard, such as JSON, YAML, etc. While this approach is convenient in that several libraries exist for both parsing and generating configuration strings/files in these

standardized formats, it also means that a dependency on one of these libraries would have to be added to either HDF5 itself, or to each VFD that will interact with configuration strings. This approach also carries the risk that the chosen format may fall out of use in the future, leading to questions about how far into the future this style of VFD configuration can be supported.

If the previous approach proves unfeasible, then configuration strings may instead follow the grammar proposed in Appendix A. Examples of how configuration strings might be formatted for each existing HDF5 VFD are provided in Appendix B. New parsing routines would need to be added to the HDF5 library and would be available as public API routines that any VFD would have access to. The associated API would deal specifically with string management for VFD configuration strings, but could be generalized in the future to handle VOL connector configuration as well. Given the tuple-based syntax proposed in Appendix A, the proposed API for parsing and inspecting elements of a configuration string is described with two simple ”parse” and ”cleanup” routines:

```
herr_t H5FDconfig_str_unwrap(const char *str,
                             H5FD_config_tuple_t **unwrapped);
herr_t H5FDconfig_free(H5FD_config_tuple_t *unwrapped);
```

Given a configuration string, `H5FDconfig_str_unwrap` would unwrap the ’top level’ tuple of the string, extracting all values in the tuple. For each value extracted from the tuple, if that element is another tuple, a subsequent call to `H5FDconfig_str_unwrap` will need to be made to parse that tuple and extract its elements. The value elements extracted from a tuple (represented by the `H5FD_config_tuple_t` type) would be allocated and would therefore need to be freed with the related `H5FDconfig_free` routine.

The `H5FD_config_tuple_t` would be a structure defined as follows:

```
typedef struct H5FD_config_tuple_t {
    int32_t    magic;   /* for sanity checking */
    size_t     nvalues; /* number of substrings */
    char     **values;  /* list of allocated, null-terminated
```

```
                                    substring copies */
} H5FD_config_tuple_t;
```

Therefore, the `unwrapped` pointer used by the above functions contains a count of the values in the tuple, and a list of those substrings, with each substring being null-terminated.

This approach does not carry the library dependency that a standard configuration format would, but it also means that THG must implement and maintain the format going forward. Further, external VFD developers will have to adapt to this format and will need to write any compatibility layer that they may need to translate between this format and any existing one that may be used by middle-ware that their VFD uses; the assumption here being that an existing standard configuration library may have provisions for this.

3.2. File Access Property List (H5P) Changes

The current `H5Pset/get_fapl_<driver>()` routines for each driver class will be retained. To be compatible with dynamic loading of VFD plugins, they may eventually need to be modified to internally configure the underlying VFD using configuration strings rather than binary runtime data. In that case, existing FAPL driver-set routines would take the received values, format them in a configuration string, and call (the private version of) the generic driver-set routines proposed below. On file open, the Virtual File Layer would identify the driver and dispatch the file open call, at which point the driver would then be responsible for unpacking the string to complete its operations.

Existing stackable drivers (family, multi, e.g.) would do the same, but with the added twist of pulling the configuration string out of the received child VFD FAPL and closing that FAPL (though the user might have to do so as well). The parent would then have the child's configuration substring embedded in its own, which would be stored as the driver's configuration info property in the FAPL. On file open, the child substring extraction and temporary VFD creation is done as normal, outlined earlier.

For supporting dynamically-loaded VFD plugins, three new routines will be added to the H5P interface:

```
herr_t H5Pset_driver_by_name(hid_t fapl_id,
                             const char *driver_name,
                             const char *driver_config);
herr_t H5Pset_driver_by_value(hid_t fapl_id,
                              H5FD_class_value_t driver_id,
                              const char *driver_config);
ssize_t H5Pget_driver_config_str(hid_t fapl_id,
                                 char *config_buf,
                                 size_t buf_size);
```

The first two routines will initially check if the driver specified by driver_name or driver_id [4] is already loaded. If it is not, they will attempt to load that driver as a plugin, using HDF5's current plugin path (as modified by the HDF5_PLUGIN_PATH environment variable or by HDF5's H5PL routines) to locate the plugin. Once the driver has been loaded successfully, the routines will set the driver and its (optional) configuration string in the FAPL, using the entire configuration string as the driver's configuration information. The driver will be responsible for checking for, retrieving and parsing any configuration string, self-configuring and then handling any VFD stacking on file open. A call to either routine will replace any existing driver configuration string property on the FAPL. To store the VFD configuration string inside a FAPL, the existing H5FD_driver_prop_t structure will be modified as follows:

```
typedef struct {
    hid_t       driver_id;    /* Driver's ID */
    const void *driver_info; /* Driver info, for open callbacks */
    const char *driver_config_str; /* ADDED */
} H5FD_driver_prop_t;
```

The last routine retrieves a string representation of the VFD configuration, such that calling H5Pset_driver_by_name or H5Pset_driver_by_value with

[4] Refer to 3.3.1 for more on driver IDs

the returned string would setup an equivalent VFD structure. The caller is responsible for passing in a buffer and the correct buffer size. The routine will determine the required size of the string and return that as the routine's return value. If the required size is greater than the specified buffer size, the routine will leave the buffer unaltered; the caller will be responsible for allocating a buffer of the appropriate size and calling the routine again. Otherwise, the routine will copy the configuration string into the output buffer from one of the following sources:

- If the driver stack was configured with a configuration string (by calling `H5Pset_driver_by_name` or `H5Pset_driver_by_value`), then that string will simply be copied from the FAPL's driver configuration string property
- If the driver stack was configured via binary data (by calling existing VFD calls like `H5Pset_fapl_family()`), the configuration string will need to be constructed by the driver stack before being returned.

The returned string will be NUL-terminated.

3.3. Virtual File Layer Changes

3.3.1. VFD class structure changes

Handling VFD plugins according to a specified name should be straightforward, as a driver's name is already part of its VFD "class" structure. For handling VFD plugins by ID, the `H5FD_class_value_t` type is a new type that will need to be introduced into the library. This type would mirror the functionality of the `H5VL_class_value_t` type for VOL connectors in that it would serve as a unique integer identifier for a VFD. This would allow third-party VFD plugin developers to register their VFD plugin with The HDF Group in the same way that third-party data filters and VOL connectors can be registered. This ID field would need to be added to the VFD class structure:

```
typedef struct H5FD_class_t {
    H5FD_class_value_t value;
```

```
    ...
} H5FD_class_t;
```

3.3.2. Addition of VFD class structure 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 all VFD classes to 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. By versioning drivers, developers can safely implement extensions to drivers without requiring preexisting drivers to conform to the additions which may not be relevant; similarly, preexisting features can be safely deprecated, modified and/or removed. A case is being made for the addition of a VFD "value" field in this RFC, with several other in-development features, such as Selection I/O and VFD SWMR, possibly necessitating similar changes.

A solution to this problem is to create a "clone" of the class type and introduce three self-descriptive components at the start of the structure:

```
typedef struct H5FD_class_v0_t {
    int32_t magic;          /* unique to H5FD_class_v*_t
                               (any versioned VFD class) */
    int16_t version_major; /* informs expected membership
                              of base class */
    int16_t version_minor; /* informs which extension of
                              base class */
    [. . .]                /* membership like the current
                              H5FD_class_t */
} H5FD_class_v0_t;
```

The magic number will be shared by (and approximately unique to) all H5FD (Virtual File Layer driver, or VFD) 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 revision of class the pointer should be cast to.

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 and each implementation is uniquely identified by a single major/minor version pair. Augmentations that require a re-implementation or deletion of extant members will require a new major version, which is appropriate for an HDF5 major release/modification anyway. A void pointer (for example, as returned by H5PLget_plugin_info()) will be cast to the base class – in this case, H5FD_class_v0_t. The magic number will be checked to guard against an inappropriate casting (e.g., if the pointer is for some reason not actually a VFD class). The major version will be checked to guard against an outdated implementation and/or re-cast as appropriate (previous VFD classes may be maintained/retained for legacy applications). Then, the minor version will be checked, and, if necessary, the pointer is re-cast as appropriate to a derived class prior to use (e.g., H5FD_class_v0_t → H5FD_class_v2_t). The details of transitioning to the new driver class are largely to-be-determined. Supporting both versioned and not-versioned classes simultaneously is not attractive, but should be possible if ugly.

3.4. VFD Plugin Support Changes (H5PL)

A new enumerated value, H5PL_TYPE_VFD, will be added to the H5PL_type_t structure 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, /**< Error                 */
    H5PL_TYPE_FILTER = 0,  /**< Filter                */
    H5PL_TYPE_VOL    = 1,  /**< VOL connector         */
    H5PL_TYPE_VFD    = 2,  /**< VFD                   */ /* ADDED */
    H5PL_TYPE_NONE   = 3   /**< Sentinel: This must be last! */
} 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
   set/get_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 provisions for for a VFD name or ”value”.

```
/* The key that will be used to find the plugin */
typedef union H5PL_key_t {
    int id; /* I/O 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;
    /* ADDED */
    struct {
        H5FD_get_driver_kind_t kind; /* Kind of VFD lookup to do */
        union {
            H5FD_class_value_t value; /* VFD value */
            const char *       name;  /* VFD name */
        } u;
    } vfd;
} H5PL_key_t;
```

`H5PL_load()`, `H5PL__open()` and `H5PL_iterate()` in H5PLint.c will be modified to handle the case of VFD plugins.

3.5. Third Party Driver Maintainer Responsibility

File Drivers not maintained by The HDF Group will need to be updated by their maintainers in order to support string properties in the FAPL. Each driver might need to modify or remove their FAPL ”driver set” and ”driver get” routines. Such updates will also likely involve accommodating VFD class structure changes as discussed in 3.3.

3.6. Integration with HDF5's Tools

Each of HDF5's tools will need to be updated to handle passing in of a VFD plugin name or ID, as well as the configuration strings for the VFD structure. These options will likely be a mirror of the existing options for VOL connectors.

4. Testing

For testing purposes, clones of an existing terminal (sec2 or stdio) and passthrough VFD (perhaps splitter) will be created. The development process will be used to clarify user documentation, possibly leading to an "SDK" for VFD plugins. At least cursory testing will be performed for all built-in VFDs with the proposed new `H5P` routines from 2.2.1, to provide some assurance that they can be used by passthrough VFDs. The existing VFD test framework should be suitable for testing built-in drivers as they are updated for string-based configuration.

5. Recommendation

This RFC concludes with a recommendation that The HDF Group should settle on a format, whether an in-house or external standardized convention (JSON, YAML, etc.), for strings that will be used to configure VFD plugins. Following that, the HDF5 library should be updated to support dynamic loading and configuring of VFD plugins using strings which are supplied to HDF5 applications and which are formatted in the settled-upon manner.

Acknowledgments

This work was funded by the DOE Exascale Computing Project (ECP).

Revision History

Version Number	Date	Comments
v1	Apr. 10, 2019	Version 1 drafted.
v2	Apr. 24, 2019	Version 2 updated to prose; restructured; adds notes on configuration and VFD class versioning.
v3	Jun. 19, 2019	Version 3 expands scope of changes to include configuring built-ins via strings, and discusses runtime caching of MPI Communicators.
v4	Jul. 01, 2019	Version 4 details intended change to VFL/VFD interface (open with string vs FAPL) and transitional steps for support with both approaches.
v5	Jul. 11, 2019	Version 5 addresses more precisely defines the configuration syntax and addresses some issues with strings vs binary compatibility for transition.
v6	Aug. 01, 2019	Version 6 simplifies the string configuration syntax, adds MPIO configuration description and example, and addresses a few minor consistency issues.
v7	Jul. 8, 2021	Version 7 restructures the document for readability and introduces new environment variables related to VFD plugin loading and configuration.

A. Appendix: Configuration Grammar

Below is the proposed syntax for VFD configuration strings: a very simple, stackable, tuple-based sequence with provisions for quoting and escaping. All VFD configuration strings would need to follow the format of: open parenthesis, driver identifier, optional additional configuration information, and close parenthesis (and terminated with the NULL character). The construction of configuration information is largely VFD-dependent, but must allow for correct parsing of the entire string. VFD plugins will be responsible for parsing their complete configuration string, including provisions for any child VFD. This configuration string syntax need not be exclusive to VFDs either; if applicable, the same syntax might be extended and re-purposed for VOL (Virtual Object Layer) connector configuration as well.

The formal description of the configuration string syntax is below in alphabetical order.

```
CONFIGURATION ::= '(' VFD_NAME [ VFD_OPTIONS ] ')'
                | '(' INTEGER_CONSTANT ')'
      A NULL-terminated string.
      Whitespace must separate VFD_NAME from any subsequent
      VFD_OPTIONS. In the case of an INTEGER_CONSTANT, this
      is handled by the library only when dealing with
      non-stringy VFD configuration, during the transition
      period only (HDF5 v1.12?), in which case the number
      must be a valid FAPL ID.

DECIMAL_CONSTANT ::= s[0-9] [ DECIMAL_CONSTANT ]

DOTTED_DIGITS ::= DECIMAL_CONSTANT '.'
                | DECIMAL_CONSTANT '.' DECIMAL_CONSTANT
                | '.' DECIMAL_CONSTANT

ESCAPE_CHAR ::= '\' ESCAPE_LITERAL
```

```
ESCAPE_LITERAL ::= '\' | '"'

EXPONENT ::= 'e' [ NUMERIC_SIGN ] DECIMAL_CONSTANT
           | 'E' [ NUMERIC_SIGN ] DECIMAL_CONSTANT

FLOAT_CONSTANT ::= DECIMAL_CONSTANT EXPONENT [ FLOATING_SUFFIX ]
                 | DOTTED_DIGITS [ EXPONENT ] [ FLOATING_SUFFIX ]

FLOATING_SUFFIX ::= 'f' | 'F' | 'l' | 'L'

HEX_CONSTANT ::= '0x' HEX_DIGIT_SEQUENCE
               | '0X' HEX_DIGIT_SEQUENCE

HEX_DIGIT_SEQUENCE ::= s[0-9A-Fa-f] [ HEX_DIGIT_SEQUENCE ]

INTEGER_CONSTANT ::= HEX_CONSTANT
                   | DECIMAL_CONSTANT

KEY_BODY_SEQUENCE ::= s[_0-9A-Za-z] [ KEY_BODY_SEQUENCE ]

KEY_NAME ::= s[_A-Za-z] [ KEY_BODY_SEQUENCE ]
       Sequence of alphanumeric characters and/or underscores,
       which must not begin with a numeric character.

KEY_VALUE_PAIR ::= '(' KEY_NAME VALUE ')'
      Any whitespace before KEY_NAME or after VALUE will
      be ignored (i.e., trimmed). Whitespace must separate
      KEY_NAME and VALUE.

LIST ::= '(' VALUE_SEQUENCE ')'
       | KEY_VALUE_PAIR
      Whitespace may appear between a parenthesis and
      VALUE_SEQUENCE { any such whitespace will be
```

```
      ignored (i.e., trimmed).

NUMERIC_SIGN::= '-' | '+'

NUMBER_CONSTANT ::= [ NUMERIC_SIGN ] INTEGER_CONSTANT
                  | [ NUMERIC_SIGN ] FLOAT_CONSTANT
      A numeric value suitable for conversion via
      strtol() or strtod(), e.g.

QUOTED_LITERAL ::= '"' SCHAR_SEQUENCE '"'
      The delimiting double-quote characters are
      removed during string parsing.

SCHAR ::= ESCAPE_CHAR
        | any character except backslash '\',
        double-quote '"', or NULL character '\0'

SCHAR_SEQUENCE ::= SCHAR [ SCHAR_SEQUENCE ]

VALUE ::= LIST
        | NUMBER_CONSTANT
        | QUOTED_LITERAL

VALUE_SEQUENCE ::= VALUE [ VALUE_SEQUENCE ]
      Whitespace must separate VALUE from any
      following VALUE_SEQUENCE.

VFD_NAME ::= KEY_NAME
      A human-readable, unique name of the driver.
      Should be identical to the name of the plugin
      file, and the name given in the driver
      implementation "class".

VFD_OPTIONS ::= KEY_VALUE_PAIR [ VFD_OPTIONS ]
```

```
      Whitespace must separate a KEY_VALUE_PAIR
      from any subsequent VFD_OPTIONS.

Metasyntax of Configuration Tuple Strings (EBNF-like)

::=          ::   association
|            ::   "OR"
s[ ... ]     ::   enclosed is a regex-like inclusion set;
                  'a-f' is range a|b|c|d|e|f
[ ... ]      ::   enclosed is optional (0 or 1)
' ... '      ::   enclosed is literal
```

B. Appendix: Configuration Grammar Examples

The example configuration strings below begin with an investigation of the built-in VFDs, which could be used with `H5Pset_driver_by_name()`. Examples with "missing" parameters – as seen below with core, family, and mirror_sock (a socket-based mirror VFD) – would use an internally-defined default value where a parameter is absent. These strings represent the configuration as they might appear in the FAPL's driver configuration string property.

Each VFD will require a formal definition of its configuration elements, specifying optional/required elements, the expected "type" of each value (such as number, generic string, another n-tuple list, or keyword [TRUE vs FALSE, e.g.]), and default values if any. The definitions given below are provisional.

```
CORE
       backing     :: 'TRUE' | 'FALSE' (default 'FALSE')
       increment   :: INTEGER_CONSTANT (default TBD)
       page_size   :: INTEGER_CONSTANT (default TBD)
                      ignored if write_tracking is not TRUE
    write_tracking :: 'TRUE' | 'FALSE' (default 'FALSE')

(core (backing TRUE) (increment 1048))

No page backing (default):
(core (increment 1048))

Combined with write tracking:
(core (increment 1048) (write_tracking TRUE) (page_size 4096))

DIRECT
    block_size :: INTEGER_CONSTANT       (default TBD)
    bounary    :: INTEGER_CONSTANT       (default TBD)
    cbuf_size  :: INTEGER_CONSTANT       (default TBD)
```

```
(direct (cbuf_size 8192) (boundary 512) (block_size 4096))

FAMILY
    member_size   :: INTEGER_CONSTANT (default TBD)
    member_driver :: CONFIGURATION (default '(sec2)')

(family (member_size 1024))
(family (member_driver (sec2)) (member_size 1024))
(family
 (member_size 1024)
 (member_driver
     (direct (cbuf_size 8192) (boundary 512) (block_size 4096))
 )
)

LOG
    buffer_size :: INTEGER_CONSTANT (default TBD)
    flags       :: INTEGER_CONSTANT (default TBD)
    logfile     :: QUOTED_LITERAL (default: none)

(log (logfile log_vfd_out.log) (flags 1048575) (buffer_size 4096))

MPIO
      comm :: QUOTED_LITERAL (required)
              Hexadecimal key of the cached communicator
              instance OR "MPI_COMM_WORLD" | "MPI_COMM_SELF"
      info :: LIST
          KEY_VALUE_PAIR* '(' INFO_KEY QUOTED_LITERAL ')'
          Zero or more key-value pairs mapping each
          value (in QUOTED_LITERAL) to its key;
          INFO_KEY is either a KEY_NAME or
          QUOTED_LITERAL, both are valid.

(mpio
```

```
  (comm "MPI_COMM_WORLD")
  (info (
     ("hdf_info_name" "XYZ")
  ))
)

MULTI
    KEY_VALUE_PAIR*
        KEY_NAME 'btree' | 'draw' | 'gheap' | 'lheap' | 'ohdr' | 'super'
        LIST
             driver  :: CONFIGURATION (default '(sec2)')
             name    :: QUOTED_LITERAL (required)
                        Filename for storage of contents
                        relevant to key name category.
             maxaddr :: EVAL (default TBD)
                        See definition below.
             memory  :: KEY_NAME (default TBD)
                        'BTREE' | 'DRAW' | 'GHEAP' |
                        'LHEAP' | 'OHDR' | 'SUPER'

The memory keynames map to the enum index in the source file:
1: SUPER, 2: BTREE, 3: DRAW, 4: GHEAP, 5: LHEAP, 6: OHDR.

The MAXADDR value is system-dependent. The driver supports
an embedded LISP-like arithmetic evaluation syntax for the option
keyname 'maxaddr' and the MAXADDR keyword, enabling a static
string to intelligently operate on this value.
    EVAL ::= INTEGER_CONSTANT
           | KEYWORD
           | '(' OP ')'
        EVAL-OP is an S-expression with contents separated by whitespace.
    IN_EVAL ::= INTEGER_CONSTANT
              | KEYWORD
              | OP
```

```
    KEYWORD ::= 'MAXADDR'
    OP ::= '*' IN_EVAL IN_EVAL
         | '/' IN_EVAL IN_EVAL
         | '+' IN_EVAL IN_EVAL
         | '-' IN_EVAL IN_EVAL
      Whitespace must separate elements in OP.
Example EVAL, MAXADDR*3/4: (/ * MAXADDR 3 4) or (* 3 / MAXADDR 4).

No specified terminal VFDs for any constituent file -- uses default:
(multi
  (super
      ((name multi_file-s.h5) (maxaddr 0))
  )
  (btree
      ((name multi_file-b.h5) (maxaddr (/ MAXADDR 4))     (memory BTREE))
  )
  (gheap
      ((name multi_file-g.h5) (maxaddr (/ * MAXADDR 3 4)) (memory GHEAP))
  )
  (draw
      ((name multi_file-r.h5) (maxaddr (/ MAXADDR 2))     (memory DRAW))
  )
)

(sec2)

(stdio)

(windows)
```

```
Examples with in-progress VFD plugins
=====================================

ROS3 (Read-Only AWS S3 VFD)
    access_key_id     :: QUOTED_LITERAL (default none)
                         Verbatim AWS-given of access key ID.
    aws_profile       :: QUOTED_LITERAL (default none)
                         Use credentials from profile on local machine.
                         Ignored any other option is present.
    aws_region        :: QUOTED_LITERAL (default none)
                         AWS-given name of region.
    secret_access_key :: QUOTED_LITERAL (default none)
                         Verbatim AWS-given secret access key.

Anonymous access:
(ros3)

Credentials stored locally in a "profile":
(ros3 (aws_profile "test-hdf5-aws"))

Explicit credentials:
(ros3
 (aws_region "us-east-1") (access_key_id "TBD") (secret_access_key "TBD")
)

MIRROR_SOCK (Socket-based mirror VFD)
    port :: INTEGER_CONSTANT (default TBD)
    ip   :: QUOTED_LITERAL (default localhost)

(mirror_sock (port 8080) (ip "127.0.0.10"))

Default port:
(mirror_sock (ip "127.0.0.12"))
```

```
SPLITTER (Passthrough VFD with read-write and read-only channels)
    logfile_path       :: QUOTED_LITERAL (default none)
                           Path/name of log file. If given,
                           keeps log of operations in the
                           filename path.
    read_write_driver  :: CONFIGURATION (default none - a default driver)
    write_only_channel :: LIST
        base_filename      :: QUOTED_LITERAL (default name received by
                              Splitter)
        driver             :: CONFIGURATION (default none -
                              a default driver )
        ignore_errors      :: KEY_NAME (default 'FALSE')
                              'TRUE' | 'FALSE'
                              If not ignored, errors on
                              write-only channel will
                              raise an error.

(splitter
  (write_only_channel (
      (driver (mirror_sock (ip localhost) (port 3000)))
      (ignore_errors TRUE)
  ) )
  (read_write_driver (sec2))
  (logfile "splitter.log")
)

Default r/w channel, no logging (default),
acknowledge errors (default):
(splitter
 (write_only_channel
     ((driver (mirror_sock (ip "localhost") (port 3000))))
 )
)
```

C. Appendix: Configuration Examples

The following are examples demonstrating how the dynamic loading of VFD plugins might be accomplished in practice. While these examples use the new grammar proposed in Appendix A, note that this is a minor detail and that the examples would be just as serviceable if the configuration strings were in some other format (JSON, YAML, etc.).

C.1. Accessing a File: Local vs AWS S3 (Amazon Simple Storage Service)

Accessing a file locally is very straightforward. In the most common case where a driver is built into the library, the programmer can use the dedicated FAPL driver-set routine associated with that driver – a fictional driver "mydriver", below – which can receive arguments for configuration information in binary form. This is the current approach for all VFDs as of HDF5 version 1.10.

```
hid_t file;
hid_t fapl;
const char filename[] = "/path/to/myfile.h5";

fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mydriver(fapl, ...);
file = H5Fopen(filename, H5F_ACC_RDONLY, fapl);
H5Pclose(fapl); /* done with FAPL */

/* do stuff using file */

H5Fclose(file); /* done with file */
```

Often, users will have no need for configuring a driver, and will bypass the FAPL creation, passing in `H5P_DEFAULT` for a default driver as determined by the library. If instead we want to access a file hosted on S3, we can make the modification to the application nearly trivial. Note that the read-only S3 VFD,

"ros3", is a plugin, so it can be selected with the proposed generic, string-based FAPL driver-set routine.

```
hid_t fapl;
hid_t file;
const char filename[] = "url://to.aws.s3/myfile.h5";
const char config[] = "(ros3 (aws_profile my-profile))";
                      /* use profile credentials */

fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_driver_by_name(fapl, "ros3", config);
file = H5Fopen(filename, H5F_ACC_RDONLY, fapl);
H5Pclose(fapl); /* done with FAPL */

/* do stuff using file */

H5Fclose(file); /* done with file */
```

C.2. VFD Stacking: A Case Study with Family and Direct

At present, the user/program must be responsible for all property lists, and building the hierarchy of drivers:

```
hid_t file;
hid_t fapl_fam;
hid_t fapl_dir;
size_t family_size = 1024;
size_t block_size = 4096;
size_t boundary = 512;
size_t cbuf_size = 8192;
const char filename[] = "/path/to/myfile.h5";

fapl_dir = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_direct(fapl_dir, boundary, block_size, cbuf_size);
```

```
fapl_fam = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_family(fapl_fam, (hsize_t)family_size, fapl_dir);

file = H5Fopen(filename, H5F_ACC_RDWR, fapl_fam);
H5Pclose(fapl_fam);
H5Pclose(fapl_dir);

/* do stuff with file */

H5Fclose(file);
```

This API for THG-supported drivers will be retained for the foreseeable future. The internals will be modified to deal with strings internally by HDF5 version 1.14. If we apply the string configuration approach, only the top level VFD is relevant to the user – this represents a paradigm shift in how the user constructs the ”stack” of VFDs.

```
hid_t file;
hid_t fapl;
const char filename[] = "/path/to/myfile.h5";

#if STATIC
  const char conf[] = "(family (member_size 1024) (member_driver " \
        "(direct (cbuf_size 8192) (boundary 512) (block_size 4096))))";
#else
  #define CONF_SIZE 1024 /* arbitrary space */
  size_t family_size = 1024;
  size_t block_size = 4096;
  size_t boundary = 512;
  size_t cbuf_size = 8192;
  const char conf[CONF_SIZE];
  snprintf(conf, CONF_SIZE,
        "(family (member_size %llu)" \
        " (member_driver" \
```

```
        " (direct (cbuf_size %llu) (boundary %llu) (block_size %llu))" \
        "))",
        family_size, cbuf_size, boundary, block_size);
  #undef CONF_SIZE
#endif /* dynamic or static configuration setting */

fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_driver_by_name(fapl, "family", conf);
file = H5Fopen(filename, H5F_ACC_RDWR, fapl);
H5Pclose(fapl);

/* do stuff with file */

H5Fclose(file);
```

It should be clear that, while the user is responsible for knowing the details of the VFD stack, the user is no longer necessarily responsible for managing that stack – the Virtual File Layer manages the stack instead.

C.3. Deeper Nesting with Splitter and Mirror

Here we look at a rather insane use case, where a user wants to mirror a file in a cycle across four machines, A to B to C to D back to A (A will have two copies of the file: the original and the fourth mirrored copy), using variants of splitter and mirror VFDs. The configuration properties for those VFDs is described in Appendix B. Assuming each machine has a mirror server listening on the default port, and the machines have IP addresses as `127.0.0.10`, `127.0.0.11`, `127.0.0.12`, `127.0.0.13` (A through D, respectfully), the resulting configuration string could look something like the following, formatted with indentation for clarity:

```
(splitter
 (write_only_channel (
   (base_filename "mirror_1")
```

```
(driver
 (mirror
  (ip "127.0.0.11")
  (remote_vfd
   (splitter
    (write_only_channel (
       (base_filename "mirror_2")
       (driver
        (mirror
         (ip "127.0.0.12")
         (remote_vfd
          (splitter
           (write_only_channel (
             (base_filename "mirror_3")
             (driver
               (mirror
                (ip "127.0.0.13")
                (remote_vfd
                 (splitter
                  (write_only_channel (
                    (base_filename "mirror_4")
                    (driver
                     (mirror
                      (ip "127.0.0.10")
                     )
                    )
                  ))
                 )
                )
               )
             )
           ))
          )
         )
        )
    ))
   )
  )
 )
```

```
              )
            )
          ))
        )
      )
    )
  )
))
)
```

Note that default values are being automatically used for the read-write driver, write channel errors (any write-channel error will result in failure), and log file path (none given, no logging). The string could be generated and used as follows:

```
# Required python 3.6+ for "new string format" features.
# A small python script to generate the string in the shell, e.g.
# The output would be copied into the application requiring the
# configuration string.
# Application code using the string is not shown.
# Included for clarity versus the C implementation below.

def recursive_create(info):
    s = ""
    if len(info) > 0 :
        remote = recursive_create(info[1:])
        if remote != "" :
            remote = f'(remote_vfd {remote})'
        fname = info[0]["filename"]
        ip = info[0]["ip"]
        wo_driver = f'(driver (mirror (ip "{ip}") {remote}))'
        s = f'(splitter (write_only_channel ((base_filename "{fname}") \
            {wo_driver})))'
    return s
```

```
machine_info = [
    {"ip": "127.0.0.11", "filename": "mirror_1"},
    {"ip": "127.0.0.12", "filename": "mirror_2"},
    {"ip": "127.0.0.13", "filename": "mirror_3"},
    {"ip": "127.0.0.10", "filename": "mirror_4"},
]

print(recursive_create(machine_info))
```

The same, but in C:

```
/* Application code contains the routine to create the
 * configuration string and use it via
 * H5Pset_driver_by_name(). The string generation in
 * the python example above would be superfluous.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct machine_info {
    const char mirror_ip[32];
    const char wo_basename[32];
};

static size_t
recursive_create(
        struct machine_info machine_list[],
        int                  machine_i,
        char                *buffer,
        size_t               buffer_len)
{
    struct machine_info info        = machine_list[machine_i];
```

```
    size_t              config_len = 0;

    /* If machine info is NULL, return 0 and write nothing to buffer
     */
    if (info.mirror_ip[0] != 0) {
        char    own_buf[256];

        /* Assemble the config for this machine.
         * Space after remote_vfd is important.
         */
        snprintf(own_buf, 256,
                "(splitter (write_only_channel ((base_filename \"%s\")" \
                "(driver (mirror (ip \"%s\") (remote_vfd ))))))",
                info.wo_basename, info.mirror_ip);
        config_len = strlen(own_buf);

        /* Only continue if there is enough space for
         * at least this config string
         */
        if (config_len <= buffer_len) {
            char   *child_buf = NULL;
            size_t ret;
            size_t max_child_len;

             /* Get config for the next machine
              * Write child config to a temporary buffer
              */
             max_child_len = buffer_len - config_len;
             child_buf = (char *)calloc(1, max_child_len * sizeof(char));
             ret = recursive_create(
                     machine_list,
                     machine_i+1,
                     child_buf,
                     max_child_len);
```

```
if (ret == 0) { /* There is no child config to insert;
                   nix remote_vfd */
     char *ptr = own_buf;

     /* find end of string */
     while (*(ptr++)) ;

     /* delete "))))))" */
     while (*(--ptr) != ' ') *ptr = '\0';

     /* delete "(remote_vfd " */
     while (*(--ptr) != ' ') *ptr = '\0';

     /* append parens */
     snprintf(buffer, buffer_len, "%s)))))", own_buf);
}
else
if (ret <= max_child_len) {
     /* Have valid child config
        and enough space */

     char *ptr = own_buf;

     /* find end of string */
     while (*(ptr++)) ;

     /* delete parentheses */
     while(*(--ptr) != ' ') *ptr = '\0';
     ptr++;

     /* "insert" child configs, write to received buffer */
     snprintf(buffer, buffer_len, "%s%s))))))",
              own_buf, child_buf);
```

```
            }
            config_len += ret; /* can be greater than buffer_len,
                                  in error */
            free(child_buf);
        } /* end if enough space for this config */
    } /* end if info exists (recursion stops) */

    return config_len; /* 0 :: stop, >buffer_len :: error,
                          else OK size written */
} /* end recursive_create() */

/* On machine with the given IP, the file with the given
 * basename will be mirrored on it.
 * Machine A (127.0.0.10) will have two files: mirror_4
 * and the original.
 */
struct machine_info info[5] = {
    {"127.0.0.11", "mirror_1"},
    {"127.0.0.12", "mirror_2"},
    {"127.0.0.13", "mirror_3"},
    {"127.0.0.10", "mirror_4"},
    {"", ""},               /* sentinel */
};

int
main(void)
{
    hid_t file;
    hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);
    /* 2048 + 1 for null-term, should be large enough */
    char config_buf[2049];
    /* info above */
    size_t ret = recursive_create(info, 0, config_buf, (size_t)2048);
```

```
    if (ret < 2049) { /* continue iff entire config fit in buffer */
        H5Pset_driver_by_name(fapl, "splitter", config_buf);

        /* or H5Fopen() */
        file = H5Fcreate("original.h5", ..., fapl, ...);

        /* close fapl, use file, close file */

        return 0;
    }
    else {
         printf("ERROR: string is too long for static buffer\n");
         return 1;
    }
} /* end main() */
```