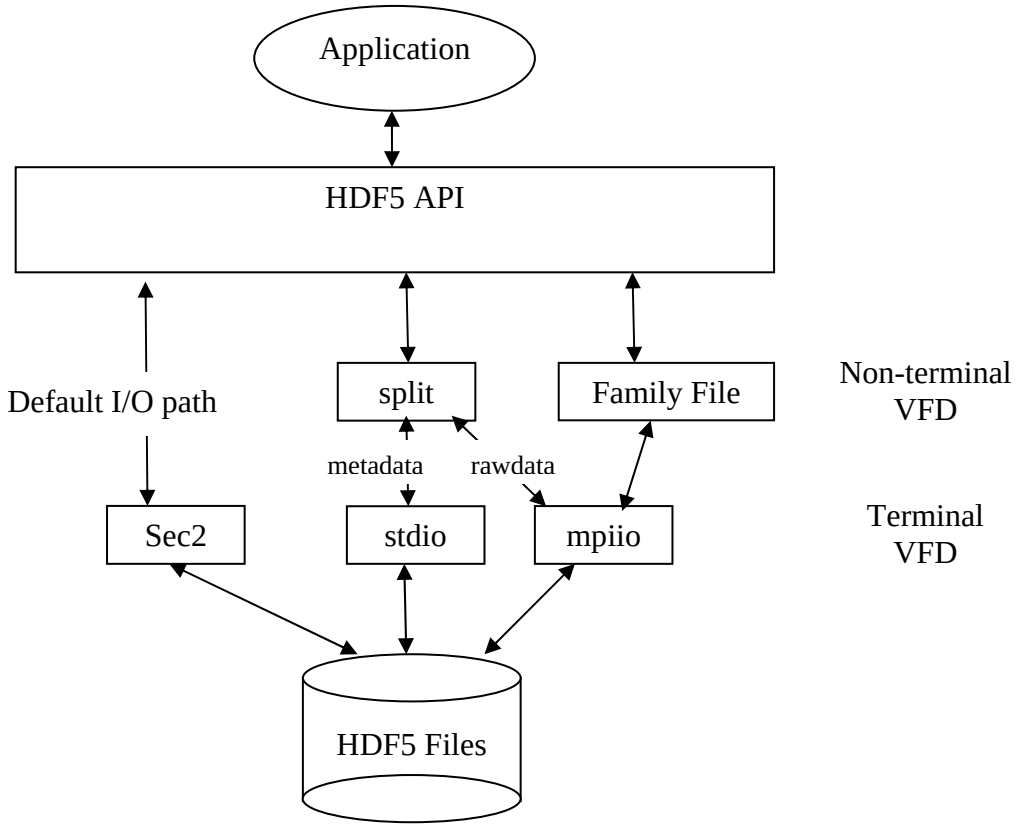
HDF5 Stackable Virtual File Driver

(Rev: June 6, 2005)

HDF5 currently has an internal "Virtual File Driver" (VFD) abstraction layer through which it performs I/O on a file. However, the VFDs implemented in the library currently perform two different functions, limiting their ability to perform certain operations.

One set of VFDs in the library don't actually perform I/O, they instead perform various "logical" partitioning of the data in the file (such as sending all the metadata to one file and all the raw data to another file) and then are hard-wired to perform I/O using a specific set of I/O calls. The other set of VFDs in the library perform physical I/O on a file (or proxy for a file, such as a memory image of a file or a network socket).

It would be very advantageous to allow "non-terminal" VFDs in the HDF5 library to stack on different "terminal" VFDs to perform their actual I/O. Additionally, this should allow new VFDs to be developed with less cost and code duplication, as they would be able to address specific logical or physical I/O issues without mixing the two domains together.



HDF5 Stackable Virtual File Driver

Examples of current implementation:

- The 'split file' VFD puts all metadata in one file and all raw data in another file, but is hard-coded to use the stdio routine to perform actual I/O on the file:

```
split
  +--stdio (meta)
  +--stdio (raw)
```

- The 'mpio' VFD enforces parallel HDF5 restrictions on metadata coherency, but is hard-coded to always use MPI-I/O for file I/O:

```
mpio = phdf5+mpiio (meta & raw)
```

Some examples of what would be possible under the revised design:

- A replacement for the current 'mpio' driver, which separated the parallel HDF5 metadata coherency constraints from the terminal I/O driver, using two VFDs:

```
phdf5
  +--mpiio
```

- A stack of drivers that enforced PHDF5 semantics on file accesses, while storing metadata and raw data in separate files. The metadata file, tend to have small sizes I/O, can be stored on a regular Unix file system while the raw data file, tend to have large size I/O, can be store on the parallel file system. This split arrangement can provide improved I/O according to I/O sizes characteristics.

```
phdf5
  +--split
     +--stdio (meta)
     +--mpiio (raw)
```

Pre-implemented Virtual File Drivers

Regarding the last changes in the HDF5 library (version 1.7.46), currently the following virtual file drivers have been implemented in the library:

- **H5FD_sec2**
  - This is the default driver which uses Posix file-system functions like `read` and `writes` to perform I/O to a single file. All I/O requests are unbuffered although the driver does optimize file seeking operations to some extent.

- **H5FD_log**
  - This VFD implements a logging mechanism along with another file. The log file is accessed through ANSI C I/O library like and stores the logging information about the operations on other file. The main file is accessed through the low level I/O functions in C. Therefore, the stacking capabilities have not been implemented yet.

- **H5FD_stdio**
  - This driver uses functions from `` `stdio.h' `` to perform buffered I/O to a single file.

- **H5FD_core**
  - This driver performs I/O directly to memory and can be used to create small temporary files that never exist on permanent storage. This type of storage is generally very fast since the I/O consists only of memory-to-memory copy operations.

- **H5FD_mpiio**
  - This is the driver of choice for accessing files in parallel using MPI and MPI-IO. It is only predefined if the library is compiled with parallel I/O support. The current implementation involves both *terminal* and *non-terminal* functionalities, which could be substituted as it will be mentioned later in this document with a stack of a non-terminal parallel and a terminal virtual file driver.

- **H5FD_mpiposix**
  - The functionalities are like H5FD_mpiio; the only difference is that it is using the file system provided consistency mechanism instead of MPI-IO library. The current implementation involves both *terminal* and *non-terminal* functionalities, which could be substituted as it will be mentioned later in this document with a stack of a sec-2 virtual file driver as a non-terminal and a terminal parallel virtual file drive.

- **H5FD_fphdf5**
  - It is a flexible parallel virtual file driver based on MPI-IO which reserves one processor called server to do all the meta-data operation while other processes are doing computation on raw data. This technique increases performance and flexibility.

- **H5FD_stream**

- It provides the notion of file over any stream based resource (i.e. over the network)

- **H5FD_family**
  - Large format address spaces are partitioned into more manageable pieces and sent to separate storage locations using an underlying driver of the user's choice. The `h5repart` tool can be used to change the sizes of the family members when stored as files or to convert a family of files to a single file or vice versa.

- **H5FD_multi**
  - Using this driver the file information can be scattered through one or more files based on specified map, which determines which logical part of the standard HDF5 file including different parts of meta-data and raw data must be assigned to which file. It supports the one level stack functionality, but only for meta-data files. It has also implemented all the split virtual driver functions for the sake of backward compatibility.

The following virtual file drivers are no longer exist or has been merged to the new ones for backward compatibility.

- **H5FD_Split**
  - Merged into H5FD_multi
- **H5FD_Gass**
  - No longer supported
- **H5FD_srb**
  - It implements the Storage Resource Broker standard to access the resources over any network of storages. No longer supported

As it is said above, after the implementation of new non-terminal file drivers and stackability capability, the H5FD_MPIPOSIX will be replaced by a combination of parallel non-terminal file driver and H5FD_Sec2 file driver.

Also, user can design and implement his/her own virtual file drivers regarding the VFD specification at *http://hdf.ncsa.uiuc.edu/HDF5/doc/TechNotes/VFL.html* using the public functions and capabilities of the HDF library. H5FD_multi is an example for such a virtual file driver. Therefore, the goal of this document is to specify not only the new rules and capabilities, which should be implemented for the pre-implemented virtual file drivers, but also the required functions for the used defined virtual file drivers in order to be stackable.

The use cases and rules for stackability

The first important issue to define the stackability of virtual file drivers is to distinguish to sets of terminal and non-terminal virtual file drivers. The meaning of terminal might be misunderstood here. **By terminal virtual file driver, we mean a driver which is doing HDF file storage and retrieval on an external storage system using a predefined protocol.**

Due to the explanations in the previous part, these two sets are:

- **Terminal virtual file drivers**
  - H5FD_sec2, H5FD_stdio, H5FD_core, H5FD_stream, H5FD_mpio, H5FD_fphdf5

- **Non-terminal virtual file drivers**
  - H5FD_multi, H5FD_family, H5FD_log

Obviously a terminal file driver cannot be pushed over any other virtual file driver either terminal or non-terminal in the stack. Now, we have the first rule of stackbility:

Rule 1: The last element in the virtual file driver stack *must* be a terminal file driver. This is the only place in the stack for a terminal file driver.

By its definition, a non-terminal virtual file driver can be placed upon any terminal or other non-terminal virtual file drivers. Although there are some exceptions because of the especial conditions of some non-terminal file drivers, which prohibits putting some terminal or non-terminal virtual file drivers on top or below them in the stack, but the above argument is generally true, so:

Rule 2: Each non-terminal virtual file driver must claim the set of non-terminal/terminal virtual file drivers, which could not be placed on them in a valid stack. Otherwise, each non-terminal virtual file driver can be placed on top of any non-terminal or terminal virtual file driver.

Apparently, as it was mentioned, not all non-terminal virtual file drivers can be pushed upon any terminal one because some conceptual and/or implementation deficiency reasons. In this section, the concern is the conceptual limitation which makes the rules. It will also give a list of implementation deficiencies in current version which have to be removed and it will be talked about at the next section.

Being more specific, about the pre-implemented virtual file drivers, the current non-terminal virtual file drivers like H5FD_Family and H5FD_Multi can be placed upon all terminal or non-terminal virtual file drivers and vice versa.

Rule 3: H5FD_family, H5FD_multi, and H5FD_log virtual file driver can be pushed upon any virtual file driver in the stack and any non-terminal file driver can be placed above them.

The current implementation of family virtual file driver does not support the functionality of putting multiple family file drivers on top of each other. It would be a good idea to change the current implementation to support such a capability. However, this issue is out of the scope of this project.

As it can be seen, two basic non-terminal operations are supported by the core virtual file derivers: partitioning (H5FD_Multi, H5FD_Family), debug and statistics (H5FD_Log). However there is not any non-terminal file driver implementing the non-terminal notion of parallelism. All of the current implemented parallel virtual file drivers are terminal. This new virtual file driver can be stacked upon any terminal parallel file driver one to increase the flexibility. We call this new virtual file driver **H5FD_phdf5.** In addition to be a wrapper for all the other parallel terminal virtual file drivers, the new virtual file driver will encapsulate the logic of meta-data/raw-data locking and consistency checking for the parallel environments. Due to the definition of **H5FD_phdf5** all the lower level terminal file drivers in the stack with **H5FD_phdf5** at the top (or a sub-stack) must be parallel and use the same media to store the data if there are more than one (for example if some partitions are done using a non-terminal file driver like family).

Rule 4: All the lower level terminal file drivers in the stack or the sub-stack with H5FD_phdf5 at the top; must be parallel and use the same media to store the data.

The Read/Write Compatibility

In the case that stack of just one virtual file driver, the virtual file which read data will be the same virtual file driver that wrote it before and its specification get persisted in the file at the *driver information block*. Apparently in the case of a stack with multiple members, the read will be possible with the same stack arrangement setting and members. The stack structure information can be put in the file like the previous case to help user with the reading data afterwards. Also there might be some possibilities that the program can write the data with some stack arrangement and read it with other one that might be different and have more performance in reading data than the writing stack.

In this version user is responsible for choosing the right driver to read the file and the compatibility issues.

The current implementation deficiencies

Regarding the above conceptual requirements, there are following deficiencies in the current versions of virtual file drivers, which should be addressed to implement the stackable virtual file driver.

1- Fortunately, the current implementation of H5FD_multi supports different kinds of virtual file drivers as its underlying virtual file driver. (We have not tested it for all cases, just relay to the current documentation).

2- The current API for H5FD_Family supports stacking other virtual file driver as the type for its members; however, some tests with H5FD_log virtual file driver shows that the implementation has some bugs. Also, the current implementation does not provide transfer mode for the member which is necessary for parallel drivers.

3- For the H5FD_log, there are some shortcomings in the current implementation:

- The current API does not support the notion of stacking above some other file.
- The current implementation uses hard-coded low/level I/O functions of C library for the main file instead of accepting another virtual file driver for the main file.
- It does not support logging multiple files in one log file in case of being above of some virtual file driver like H5FD_Family.

4- The H5FD_phdf should be implemented as a new virtual file driver.

The New API

The first group of API which each non-terminal virtual file driver must implement is stacking API. Considering compatibility issue the API will be the same in the current version with adding a new parameter to the function representing the file access property list of the top of the virtual file driver stack that is optional and can be `NULL`:

```
herr_t H5Pset_fapl_x(x driver specific properties+, hid_t stack_top)
```

The setting also can be performed by direct call to the `H5Pset_driver` using the driver specific properties struct, like the current version.

```
static H5FD_x_fapl_t xa = {x driver specific properties+, stack_top};
herr_t H5Pset_driver(fapl, H5FD_X, &xa);
```

For example, consider H5FD_family virtual file driver on top of H5FD_Sec2:

```
hid_t sapl = H5create(H5P_FILE_ACCESS);
H5set_fapl_sec2(sapl);
hid_t fapl = H5create(H5P_FILE_ACCESS);
size_t member_size = 100 * 1024 * 1024;
H5Pset_fapl_family(fapl,member_size,sapl2);
Hid_t file= H5create("foo%0d.h5",H5F_ACC_TRUNC,H5P_DEFAULT,fapl);
H5close(fapl);
```

The stack members' information can be retrieved using a similar API to the current API:

```
herr_t H5Pget_next_in_stack_size(fapl, H5FD_mem_t *branch_size)
```

to find how many member is in the stack with fapl on top of it (horizontally) and then:

```
herr_t H5Pget_next_in_stack(fapl, hid_t * stack_top);
```

will provide user with an array of virtual file driver members on top of the stack or sub-stack with fapl on the top. User can apply the call to the retrieved virtual file drivers to extract the remaining virtual file drivers in the stack.

For example, for family file driver, we will have:

```
…
hid_t satck_top;
H5FD_mem_t bsize,mt;
….
H5Pget_next_in_stack_size(fapl, &bsize);
H5Pget_next_in_stack(fapl,&stack_top);
for (mt=H5FD_MEM_DEFAULT; mt< bsize; mt=(H5FD_mem_t)(mt+1))
{
      … stack_top[mt]
      …
}
…
```

For the file level operation, we still can use the current API. The virtual file driver which is on the top of the stack will take the responsibility of doing all those file operations. For I/O operations like read and write we might want to change the data transfer properties for especial virtual file driver in the stack. One solution might be stacking those properties like the file access properties; however, since each data transfer mode is meaningful for just one virtual file driver and also using the current API helps us with the compatibility issues, for this group the API will remain the same as current version. Other API's (like address space, optimization …) remain unchanged.

Implementation road

The changes that should be maid on the existing library code are:

1- Change the file access properties data structure of all non-terminal virtual file drivers to:

```
typedef struct H5FD_x_fapl_t
{
     /* driver specific */
     hid_t stack_top
}
```

This makes us to change the mode functions of these drivers and also changes the file functions to use the new structure to create and update new file description and copy the access property list stack in the proper file description in the stack.

2- Change the file description data structure of all non-terminal virtual file drivers to:

```
typedef struct H5FD_x_t
{
     H5FD_t pub;
     H5FD_t *stack;
     /* the driver specific properties */
}
```

3- Add data transfer properties/ information to all non-terminal virtual file drivers because they might be in the stack on top of any other virtual file driver. It changes the transfer query functions

4- Change data functions to support the chain call with transfer mode through the stack.

5- The stack rules which mentioned in the previous sections will be implemented in the mode functions (setter functions) of each virtual file driver.

6- Change query driver information functions to support the stack information.

Test cases

1- Basic functionalities after the code change for each driver alone:
  1-1- Mode functions
  1-2- File functions
  1-3- Data Functions

2- Building Stack of following drivers (non-terminal and terminal):

- [(Family, log),(Sec-2,Stdio,MPIO,Mpi-posix,fphdf,Core,Stream)]
- [Multi, (meta data): (Sec-2,Stdio,Core,stream,srb)
  (raw data): (Sec-2, Stdio, MPIO, Mpi-posix, fphdf, Core, Stream)]

  2-1- Mode functions
  2-2- File functions
  2-3- Data Functions

3- Stack of multiple non-terminal virtual file driver

- [Family,log,(Sec-2,Stdio,MPIO,Mpi-posix,fphdf,Core,stream,srb)]
- [Multi, (Meta data) Family, log,(Sec-2,Stdio,Core,stream,srb)
  (raw data):Family,log,(Sec-2,Stdio,MPIO,Mpi-posix,fphdf,Core,stream)]

1-1- Mode functions
1-2- File functions
1-3- Data Functions

4- Data transfer modes for stacks with especial terminal driver supporting different data transfer mode:

- Sec-2,Stdio,MPIO,Mpi-posix,fphdf,Core,stream

5- Test the new non-terminal parallel virtual file driver, on top of the (MPIO,Mpi-posix,fphdf)
5-1- Mode functions
5-2- File functions
5-3- Data Functions with different transfer modes

6- Test the new non-terminal parallel virtual file driver on top of other non-terminal file drivers
[famliy,log,VPFD,(MPIO,MPIO-POSIX,fphdf5)]