

RFC: Load Core File Driver From Image

John Mainzer

While the HDF5 API currently allows the core file driver to be opened with data taken from an existing file, there have been user requests to modify the API to allow the user to both create and read in core HDF5 file images with the core file driver without requiring disk I/O.

This RFC proposes API extensions to allow these operations.

1 Background

On systems with sufficient main memory, the core file driver allows fast HDF5 file access, by creating the HDF5 file in RAM, supporting all normal HDF5 operations on the file, and if desired, storing the file to disk on flush or file close. The core file driver also allows an existing file on disk to be loaded into RAM for fast access via the core file driver.

2 Goal

In some cases, it would be useful to create an image of an HDF5 file on one process, transmit it to a second, and then read it without any mandatory file I/O on either end, and with minimal overhead. For large images, minimizing the overhead means sharing buffers between the HDF5 library and the application so as to avoid large memory copies. As sharing buffers like this has a number of potential problems, the option of copying buffers must also be maintained.

In this RFC, I propose HDF5 API extensions and modifications to support these operations.

3 Use Cases

The following list of use cases is an explicit statement of the use cases driving the requirements for this document. Only the first two use cases are of interest to LLNL, the remaining use cases are unrelated to LLNL's.

Use Case 1: Reading an HDF5 File Image

Given an image of an HDF5 file in memory, open it, and read it without any file system I/O.

This is mostly a matter of using a memory based file driver (just the core file driver for now), to open and read the image. LLNL requires that when the image is opened read-only, it is possible to avoid the necessity of transferring ownership of the buffer from the application to the HDF5 library if the application agrees not to discard the image until the file is closed.

Use Case 2: Construction of an HDF5 File Image

Construct an image of an HDF5 file in memory for transmission from one process to another. As the purpose of the operation is to avoid file system I/O, the solution must avoid file I/O unless requested.

At a minimum, this requires transferring the image from HDF5 ownership (i.e. responsibility for discarding) to application ownership.

The safe way to do this is with a copy into a buffer allocated by the application. While this must be supported, it will be slow for large files, and thus more efficient means must be implemented as well. At a minimum, this means that it must be possible for the HDF5 library to allocate a buffer, write an image of an HDF5 file, and then pass ownership of the buffer to the application.

It may be useful to support passing an HDF5 file image back and forth between the HDF5 Library and the application, possibly with modifications on either end. However, this is not required, and should not be done unless it can be done cheaply.

Use Case 3: Template File

Either to avoid numerous collective calls (PHDF5), or to standardize the structure of HDF5 files, when creating a new file, write the contents of a template file containing the desired group, data type, and possibly definitions, and then open the file. (It should be mentioned that we have no plans for implementing support for this use case at this time - it is offered solely as a point to consider in designing our API extensions/modifications)

4 Approach

As currently envisioned, the HDF5 file image creation and read operations will center on the core file driver as a way of avoiding undesired file I/O. Thus, at first blush it would make sense to use API additions/modifications specific to the core file driver.

However, supplying an initial image of a file at creation does have at least one other use case – that of allowing the use of a template file containing initial datatype, group, and possibly dataset definitions to either enforce uniformity of basic file structure or (in the parallel case) to avoid long sequences of collective operations on metadata.

Mechanisms to allow the user application to obtain access to an HDF5 file image from the core file driver without a `memcpy()` seem peculiar to the core file driver, as file drivers that actually store data in a file system will typically have little if any of the file in core. The only point militating against a core file driver specific API for this case is the possibility of future file drivers that also keep their data in core (say a distributed core file driver for the parallel case).

In contrast, mechanisms to obtain a buffer containing an image of a file are generally applicable to all file drivers, albeit unnecessary as (with the exception of the core file driver) the operation is easily done via standard C library calls, if desired.

A final point in choosing our API modifications is the desirability of extending the interface, rather than modifying it so as to avoid breaking existing applications.

In the remainder of this RFC, I introduce several proposed API extensions directed at supporting HDF5 file image creation and read operations.

5 New API Call Syntax

New API calls described in this document fall into two categories: low-level API routines that are added to the main HDF5 library, and high-level API routines added to the “lite” API in the high-level wrapper library. The high-level API routines use the new low-level API routines, but present frequently requested functionality in a convenient way for application developers’ use.

5.1 New Low-Level API Routines

These routines represent new functionality that extend the capabilities of the main HDF5 library, allowing a memory image of an HDF5 file to be used when opening that file. These low-level routines are designed to provide the core functionality required to support this feature, with popular and convenient options provided in the high-level API routines described in the next section.

As the core file driver already supports creation of in memory images of HDF5 files, the basic approach to opening an in memory image of an HDF5 file is pass the image to the core file driver, and tell it to open it. We will do this by adding the H5Pget/set_file_image calls, which allow the user to specify an initial file image for this and other purposes (see use cases in section 4, above).

Proposed syntax for the H5Pget/set_file_image calls is offered below.

5.1.1 H5Pset_file_image

The H5Pset_file_image routine is designed to allow an application to provide the image for a VFD to use as the initial contents of the file. This call is designed primarily for use with the core VFD, but can be used with any VFD that supports using an initial file image when opening a file (described in section 5.1.3, “The H5FD_FEAT_ALLOW_FILE_IMAGE flag”). Calling this routine makes a copy of the file image buffer provided, using the file image allocation callbacks (described in section 5.1.4) for allocating and releasing the memory used for copying the file image. Note that if the file image allocation alloc_func callback returns a pointer that is the same value as the pointer passed in to H5Pset_file_image, the copy operation is not performed by H5Pset_file_image.¹

The signature of H5Pset_file_image is defined as follows:

```
herr_t H5Pset_file_image(hid_t fapl_id, void *buf_ptr, size_t buf_len)
```

The parameters of H5Pset_file_image are defined as follows:

- fapl_id contains the ID of the target file access property list.
- buf_ptr supplies a pointer to the initial file image, or NULL if no initial file image is desired.
- buf_len contains the size of the supplied buffer, or 0 if no initial image is desired.

Note: if either the buf_len parameter is zero, or the buf_ptr parameter is NULL, no file image will be set in the FAPL, and any existing file image buffer in the FAPL will be released, using the

¹ Avoiding the copy operation by using a custom alloc_func callback can be used as a mechanism for transferring ownership of a buffer to the HDF5 library without making a copy, or for sharing a buffer between the application and the HDF5 library. Pre-defined ways to perform some of these buffer operations are described in section 5.2.1, describing the new H5LTopen_file_image routine.

current file image allocation callbacks set for the FAPL, setting the FAPL's file image `buf_len` to 0 and `buf_ptr` to NULL.

5.1.2 H5Pget_file_image

The `H5Pget_file_image` routine is designed to allow an application to retrieve a copy of the file image designated for a VFD to use as the initial contents of the file. This routine uses the file image allocation callbacks when allocating the buffer to return to the application. Note that if the file image allocation `alloc_func` callback returns a pointer that is the same value as the pointer to the file buffer currently set in the FAPL, the copy operation is not performed by `H5Pget_file_image`.

The signature of `H5Pget_file_image` is defined as follows:

```
herr_t H5Pget_file_image(hid_t fapl_id, void **buf_ptr_ptr, size_t *buf_len_ptr)
```

The parameters of `H5Pget_file_image` are defined as follows:

- `fapl_id` shall contain the ID of the target file access property list.
- `buf_ptr_ptr` shall be NULL, or shall contain a pointer to a void*. If `buf_ptr_ptr` is not NULL, on successful return, `*buf_ptr_ptr` shall contain the value of a pointer to a copy of the initial image provided in the last call to `H5Pset_init_image` for the supplied `fapl_id`, or NULL if there is no initial image set.
- `buf_len_ptr` shall be NULL, or shall contain a pointer to `size_t`. If `buf_len_ptr` is not NULL, on successful return, `*buf_len_ptr` shall contain the value of the `buf_len` parameter for the initial image in the supplied `fapl_id`, which shall be 0 if no initial image is set.

5.1.3 The H5FD_FEAT_ALLOW_FILE_IMAGE flag

The `H5FD_FEAT_ALLOW_FILE_IMAGE` flag will be added to the list of existing public VFD flags (defined in `H5FDpublic.h`). A VFD that sets the `H5FD_FEAT_ALLOW_FILE_IMAGE` flag when its 'query' callback is called indicates that the file image set in the FAPL will be used as the initial contents of a file.² If the VFD supports the `H5FD_FEAT_ALLOW_FILE_IMAGE` flag, and an initial file image is defined by an application, the VFD should ensure that file creation operations (as opposed to file open operations) bypass use of the file image, and create a new, empty file.³

5.1.4 H5Pset_file_image_alloc_callbacks

In order to provide an application with flexibility over how file image buffers are managed, callback routines can be set by an application to control file image buffer allocation, re-allocation and release. These routines are invoked whenever a new file image buffer is allocated (generally in support of copying the buffer), an existing file image buffer is resized, or when a file image buffer is released

² Support for setting an initial file image is designed primarily for use with the core VFD. However, any VFD can indicate support for this feature by setting the `H5FD_FEAT_ALLOW_FILE_IMAGE` flag, and copying the image in an appropriate way for the VFD (possibly by writing the image to a file and then opening the file, etc).

³ Alternatively, when a file image is set and a new file is created with a VFD, the VFD behavior could be defined to return an error. However, this behavior seems somewhat more user-friendly.

from use. The operation and return values of the allocation callbacks are identical to those of the corresponding C standard library calls (i.e. malloc, realloc and free), although the parameters are expanded (described below).

The signature of `H5Pset_file_image_alloc_callbacks` is defined as follows:

```
herr_t H5Pset_file_image_alloc_callbacks(hid_t fapl_id,
    void *(*image_alloc)(size_t size, unsigned file_image_op_flags,
        void *alloc_udata),
    void *alloc_udata,
    void *(*image_realloc)(void *ptr, size_t size, file_image_op_flags,
        void *realloc_udata),
    void *realloc_udata,
    void (*image_free)(void *ptr, unsigned file_image_op_flags,
        void *free_udata),
    void *free_udata)
```

The parameters of `H5Pset_file_image_alloc_callbacks` are defined as follows:

- `fapl_id` shall contain the ID of the target file access property list.
- `image_alloc` shall contain a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library `malloc()` call. The parameters to the `image_alloc` callback are defined as follows:
 - `size` will contain the size of the image buffer to allocate, in bytes.
 - `file_image_op_flags` will be set with one or more flags (defined below) indicating the operation being performed on the file image when this callback is invoked.
 - `alloc_udata` will be set with the value passed in for the `alloc_udata` parameter to `H5Pset_file_image_alloc_callbacks`.

Setting `image_alloc` to NULL will indicate that the HDF5 library should invoke the standard C library `malloc()` routine when allocating file image buffers.

- `alloc_udata` shall contain a pointer value, potentially to user-defined data, that will be passed to the `image_alloc` callback.
- `image_realloc` shall contain a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library `realloc()` call. The parameters to the `image_realloc` callback are defined as follows:
 - `ptr` will contain the pointer to the buffer being resized.
 - `size` will contain the size of the image buffer to allocate, in bytes.
 - `file_image_op_flags` will be set with one or more flags (defined below) indicating the operation being performed on the file image when this callback is invoked.
 - `realloc_udata` will be set with the value passed in for the `realloc_udata` parameter to `H5Pset_file_image_alloc_callbacks`.

Setting `image_realloc` to NULL will indicate that the HDF5 library should invoke the standard C library `realloc()` routine when resizing file image buffers.

- `realloc_uda` shall contain a pointer value, potentially to user-defined data, that will be passed to the `image_realloc` callback.
- `image_free` shall contain a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library `free()` call. The parameters to the `image_free` callback are defined as follows:
 - `ptr` will contain the pointer to the buffer being released.
 - `file_image_op_flags` will be set with one or more flags (defined below) indicating the operation being performed on the file image when this callback is invoked.
 - `free_uda` will be set with the value passed in for the `free_uda` parameter to `H5Pset_file_image_alloc_callbacks`.

Setting `image_free` to NULL will indicate that the HDF5 library should invoke the standard C library `free()` routine when releasing file image buffers.

- `free_uda` shall contain a pointer value, potentially to user-defined data, that will be passed to the `image_free` callback.

The values that can be set for the `file_image_op_flags` parameter to all the callbacks above are defined as follows:⁴⁵

- `H5_FILE_IMAGE_PROPERTY_LIST_SET` – This flag is passed to the `image_alloc` callback when an image buffer is being copied while being set in a FAPL.
- `H5_FILE_IMAGE_PROPERTY_LIST_COPY` – This flag is passed to the `image_alloc` callback when an image buffer is being copied when a FAPL is copied.
- `H5_FILE_IMAGE_PROPERTY_LIST_GET` – This flag is passed to the `image_alloc` callback when an image buffer is being copied while being retrieved from a FAPL.
- `H5_FILE_IMAGE_FILE_OPEN` – This flag is passed to the `image_alloc` callback when an image buffer is copied during a file open operation.
- `H5_FILE_IMAGE_PROPERTY_LIST_CLOSE` – This flag is passed to the `image_free` callback when an image buffer is being released during a FAPL close operation.
- `H5_FILE_IMAGE_FILE_CLOSE` – This flag is passed to the `image_free` callback when an image buffer is being released during a file close operation.

5.1.5 `H5Pget_file_image_alloc_callbacks()`

The `H5Pget_file_image_alloc_callbacks` routine is designed to provide applications with the functionality to query the existing file image allocation routines and user data pointers.

The signature of `H5Pget_file_image_alloc_callbacks()` is defined as follows

⁴ We may need more/less of these flags.

⁵ Note that there aren't currently any flags defined to passing to the `realloc_image` callback, but those could be added in the future and the `file_image_op_flags` parameter is included in that callback for orthogonality, completeness and future expansion.

```

herr_t H5Pset_fapl_core_alloc_callbacks(hid_t fapl_id,
    void (**image_alloc_ptr)(size_t size, unsigned file_image_op_flags,
        void *alloc_udata),
    void **alloc_udata_ptr,
    void (**image_realloc_ptr)(void *ptr, size_t size,
        file_image_op_flags, void *realloc_udata),
    void **realloc_udata_ptr,
    void (**image_free_ptr)(void *ptr, unsigned file_image_op_flags,
        void *free_udata),
    void **free_udata_ptr)

```

The parameters of `H5Pset_file_image_alloc_callbacks()` are defined as follows:

- `fapl_id` shall contain the ID of the target file access property list.
- `image_alloc_ptr` shall contain a pointer to pointer to function. Upon successful return, `*image_alloc_ptr` shall contain the pointer passed as the `image_alloc` parameter in the last call to `H5Pset_file_image_alloc_callbacks()` for the specified FAPL, or NULL if there has been no such call.
- `alloc_udata_ptr` shall contain a pointer to pointer to a void*. Upon successful return, `*alloc_udata_ptr` shall contain the pointer passed as the `alloc_udata` parameter in the last call to `H5Pset_file_image_alloc_callbacks()` for the specified FAPL, or NULL if there has been no such call.
- `image_realloc_ptr` shall contain a pointer to pointer to function. Upon successful return, `*image_realloc_ptr` shall contain the pointer passed as the `image_realloc` parameter in the last call to `H5Pset_file_image_alloc_callbacks()` for the specified FAPL, or NULL if there has been no such call.
- `realloc_udata_ptr` shall contain a pointer to pointer to a void*. Upon successful return, `*realloc_udata_ptr` shall contain the pointer passed as the `realloc_udata` parameter in the last call to `H5Pset_file_image_alloc_callbacks()` for the specified FAPL, or NULL if there has been no such call.
- `image_free_ptr` shall contain a pointer to pointer to function. Upon successful return, `*image_free_ptr` shall contain the pointer passed as the `image_free` parameter in the last call to `H5Pset_file_image_alloc_callbacks()` for the specified FAPL, or NULL if there has been no such call.
- `free_udata_ptr` shall contain a pointer to pointer to a void*. Upon successful return, `*free_udata_ptr` shall contain the pointer passed as the `free_udata` parameter in the last call to `H5Pset_file_image_alloc_callbacks()` for the specified FAPL, or NULL if there has been no such call.

5.2 New High-Level API Routines

These high-level routines encapsulate the capabilities of routines in the main HDF5 library with conveniently accessible abstractions.

5.2.1 H5LTopen_file_image

The H5LTopen_image routine is designed to provide a convenient way to open an initial file image with the core VFD. Flags to H5LTopen_file_image allow for various file image buffer ownership policies to be requested conveniently.

The signature of H5LTopen_file_image is defined as follows:

```
hid_t H5LTopen_file_image(void *buf_ptr, size_t buf_len, unsigned flags)6
```

The parameters of H5LTopen_file_image() shall be defined as follows:

- buf_ptr shall contain a pointer to the supplied initial image. A NULL value is invalid and will cause H5LTopen_file_image to fail.
- buf_len shall contain the size of the supplied buffer. A 0 value is invalid and will cause H5LTopen_file_image to fail.
- flags shall contain a set of flags indicating restrictions on the use of the buffer, whether HDF5 is to take control of the buffer, and how long the application promises to maintain the buffer. Possible flags are as follows:
 - H5LT_FILE_IMAGE_DONT_COPY - Indicates that the HDF5 library should not copy the file image buffer provided, but should use it directly. The HDF5 library will release it when done, however. The supplied buffer must have been allocated via a call to the standard C library malloc() or calloc() routines, as the HDF5 library will call free() to release the buffer. In the absence of this flag, the HDF5 library will copy the buffer provided.⁷
 - H5LT_FILE_IMAGE_DONT_RELEASE - Only valid when the H5LT_FILE_IMAGE_DONT_COPY flag is also specified, this flag indicates that the HDF5 library should not attempt to release the buffer when the file is closed. In the absence of this flag, the HDF5 library will release the buffer after the file is closed.⁸
 - H5LT_FILE_IMAGE_ALLOW_WRITE - Indicates that the buffer may be modified by HDF5. Absence of this flag indicates that the file image will only be read from.
 - H5LT_FILE_IMAGE_DONT_RESIZE – Only valid when the H5LT_FILE_IMAGE_ALLOW_WRITE flag is also specified, this flag indicates that any write that requires a change in the file image buffer size should fail. In the absence of

⁶ Note that there's no way to specify a "backing store" file name in this definition of H5LTopen_image, but that is a possible addition, if desired.

⁷ The H5LT_FILE_IMAGE_DONT_COPY flag provides an application with the ability to "give ownership" of a file image buffer to the HDF5 library.

⁸ Using H5LT_FILE_IMAGE_DONT_RELEASE (with the required H5LT_FILE_IMAGE_DONT_COPY flag) provides a way for the application to specify a buffer that the HDF5 library can use for opening and accessing as a file image, but letting the application retain ownership of the buffer. If the H5LT_FILE_IMAGE_ALLOW_WRITE is given also, the H5LT_FILE_IMAGE_DONT_RESIZE is recommended as well, so that the HDF5 library doesn't attempt to re-allocate the file image buffer.

this flag, the file image may be re-allocated into a larger or smaller buffer, and the write performed as usual.

The return value of `H5LTopen_file_image` will be a file ID on success, or a negative value on failure. The file ID returned should be closed with `H5Fclose`.

5.2.2 `H5LTget_file_image`

The purpose of the `H5LTget_file_image` routine is to provide a simple way to retrieve a copy of the image of an existing, open file. This routine can be used with files opened using any VFD.

The signature of `H5LTget_file_image` shall be defined as follows:

```
ssize_t H5LTget_file_image(hid_t file_id, void *buf_ptr, size_t buf_len)
```

The parameters of `H5LTget_file_image` shall be defined as follows:

- `file_id` shall contain the ID of the target file
- `buf_ptr` shall contain a pointer to the buffer into which the image of the HDF5 file is to be copied. If `buf_ptr` is `NULL`, no data will be copied, but the return value will still indicate the buffer size required (or a negative value for an error).
- `buf_len` shall contain the size of the supplied buffer.

The return value of `H5LTget_file_image` will be a positive value indicating the length of buffer required to store the file image (i.e. the length of the file⁹), or a negative value if the file is too large to store in memory or on failure.

6 New API Call Semantics

6.1 Core Allocation Callback Semantics

The `H5Fget/set_file_image_alloc_callbacks()` API calls allow the application to hook the memory management calls in the core file driver (and if desired, in any future driver that keeps the entire file in core).

From the perspective of the HDF5 library, the supplied `image_alloc()`, `image_realloc()`, and `image_free()` drivers must function identically to the C standard library `malloc()`, `realloc()`, and `free()` calls.

What happens on the application side can be much more nuanced, particularly with the ability to pass user data to the callbacks.

For example, the application can use the `file_image_op_flags` parameter of the `image_free()` call to allow it to grab the buffer containing the final in memory image of the file created by the core file driver just before the HDF5 library closes the file. At this point the application can do whatever it wants with this buffer, as HDF5 thinks it has been freed.

⁹ The current file size can also be obtained via a call to `H5Fget_filesize`.

Further, if the application doesn't want to keep track of all the different core file drivers, it can increment all malloc requests by `sizeof(size_t)`, and use the first `sizeof(size_t)` bytes of the buffer to store the buffer length – offsetting the pointer returned to the core file driver by `sizeof(size_t)` bytes.

As should be obvious from the above, the core allocation callbacks offer clean way to obtain an image of an HDF5 file from the core file driver without the cost of a `memcpy()`. Further, in combination with the initial file image facility discussed below, it provides us with a clean mechanism for passing a buffer that contains an HDF5 file back and forth between the application and the HDF5 library.

6.2 Initial File Image Semantics

One can argue whether creating a file with an initial file image is closer to creating a file or opening one. While I tend to view it as being closer to a file create, the consensus seems to be that it is closer to a file open. Bowing to the consensus, we shall require that the initial image only be set for calls to `H5Fopen()`.

Whatever our convention, from an internal perspective, it is a bit of both. Conceptually, we will create a file on disk, write the supplied image to it, close it, open it as an HDF5 file, and then proceed as usual.¹⁰ This process is similar to a file create, as we are creating a file that didn't exist on disk to begin with and writing a bunch of data to it. Also, we must verify that no file of the supplied name is open. However, it is also similar to a file open, as we must read the superblock and handle the usual file open tasks.

Implementing the above sequence of actions has a number of implications on the behavior of the `H5Fopen()` call:

1. `H5Fopen()` must fail if the target file driver doesn't support the "allow file image" VFD flag and a file image is specified in the FAPL.¹¹
2. If the target file driver supports the "allow file image" VFD flag, `H5Fopen()` must fail if the file is already open, or if a file of the specified name exists.
3. Even if the above constraints are satisfied, `H5Fopen()` must still fail if the image doesn't contain a valid (or perhaps just plausibly valid) image of an HDF5 file. In particular, the superblock must be processed, and the file structure be set up accordingly.

In addition to the effects on `H5Fopen()`, there is also the matter of management of the supplied buffer when `H5LTopen_file_image()` is used instead of `H5Fopen()`. As management of the buffer will be driven in large part by the flags specified in the call to `H5LTopen_file_image()` call, I reproduce the list of flags here:

- `H5LT_FILE_IMAGE_DONT_COPY` - Indicates that the HDF5 library should not copy the file image buffer provided, but should use it directly. The HDF5 library will release it when done, however. The supplied buffer must have been allocated via a call to the standard C library `malloc()` or `calloc()` routines, as the HDF5 library will call `free()` to release the buffer. In the absence of this flag, the HDF5 library will copy the buffer provided.

¹⁰ Although this is not true when the core VFD is used.

¹¹ Note that to support stackable VFDs, internal VFDs will have to be able to construct this flag at run time by querying their descendants.

- H5LT_FILE_IMAGE_DONT_RELEASE - Only valid when the H5LT_FILE_IMAGE_DONT_COPY flag is also specified, this flag indicates that the HDF5 library should not attempt to release the buffer when the file is closed. In the absence of this flag, the HDF5 library will release the buffer after the file is closed.
- H5LT_FILE_IMAGE_ALLOW_WRITE - Indicates that the buffer may be modified by HDF5. Absence of this flag indicates that the file image will only be read from.
- H5LT_FILE_IMAGE_DONT_RESIZE – Only valid when the H5LT_FILE_IMAGE_ALLOW_WRITE flag is also specified, this flag indicates that any write that requires a change in the file image buffer size should fail. In the absence of this flag, the file image may be re-allocated into a larger or smaller buffer, and the write performed as usual.

The application is responsible for discarding the buffer unless the H5LT_FILE_IMAGE_DONT_COPY and H5LT_FILE_IMAGE_DONT_RELEASE flags are set. In this case the driver must discard the buffer when done. <<More discussions here? – QAK>>

7 Application of API Changes to the Primary Use Cases

Only the first two of the use cases listed above are of interest to LLNL – thus only these two cases are addressed below.

7.1 Reading an in Memory HDF5 File Image

HDF5 already allows the core file driver to be initialized from a file. When implemented, the new H5Pset_file_image() API call will allow the core file driver to be initialized from an application provided buffer. The following pseudo code illustrates its use:

```
<allocate and initialize buf_len and buf>
<allocate fapl_id>
<set fapl to use core file driver>

H5Pset_file_image(fapl_id, &buf, buf_len);

<open file, read as desired, close>
<discard buf>
```

This pseudo code shows how a buffer can be passed back and forth between application and HDF5 library.

```
/* initial declarations */
void * image_ptr;
size_t image_len;

void image_free(void *ptr, unsigned file_image_op_flags, void *udata)
{
    if(file_image_op_flags == H5_FILE_IMAGE_FILE_CLOSE)
        *(void **)udata = ptr;
    else
        free(ptr);
}
```

```

/* allocate image in application and initialize as desired */
<allocate and initialize image and image_len>

/* pass image off to core file driver */
<allocate fapl_id and set to use core file driver>

H5Pset_file_image_alloc_callbacks(fapl_id, NULL, NULL, NULL, NULL,
    image_free, &image_ptr);
H5Pset_file_image(fapl_id, image_ptr, image_len);

<open core file using fapl_id, modify it, flush it>

/* pass image back to application */
H5Fget_filesize(fid, &image_len);

<close core file>
/* image_ptr now contains a pointer to the final version of the core file.
*/

<modify *image_ptr as desired>

/* pass image back to core file driver */
H5Pset_file_image(fapl_id, image_ptr, buf_len);

<open core file using fapl_id, modify it, close it>
/*image_ptr again contains a pointer to the final version of the core file
*/

<use it, and then discard it via free()>

```

The pseudo code below shows the same operations, using H5LTopen_file_image instead of coding file image allocation callbacks in the application:

```

/* initial declarations */
void * image_ptr;
size_t image_len;

/* allocate image in application and initialize as desired */
<allocate and initialize image and image_len>

/* open image with core file driver, indicating that the application will
still "own" the image buffer */
fid = H5LTopen_file_image(image_ptr, image_len,
    H5LT_FILE_IMAGE_DONT_COPY | H5LT_FILE_IMAGE_DONT_RELEASE |
    H5LT_FILE_IMAGE_ALLOW_WRITE | H5LT_FILE_IMAGE_DONT_RESIZE);

<modify file, close it>
/* image_ptr now contains a pointer to the final version of the core file.
*/

<modify *image_ptr as desired>

```

```

/* pass image back to core file driver */
fid = H5LTopen_file_image(image_ptr,image_len,
    H5LT_FILE_IMAGE_DONT_COPY | H5LT_FILE_IMAGE_DONT_RELEASE |
    H5LT_FILE_IMAGE_ALLOW_WRITE | H5LT_FILE_IMAGE_DONT_RESIZE);

<modify file, close it>
/* image_ptr again contains a pointer to the final version of the core file
*/

<use it, and then discard it via free(>

```

Note that although the code above allows the file image to be modified when the image is opened with `H5LTopen_file_image`, it doesn't allow re-sizing the image buffer.

Finally, the pseudo code below illustrates using `H5LTopen_file_image` to open a R/W file with a R/O buffer that must be copied on file open.

```

<allocate and initialize buf_len and buf_ptr>

fid = H5LTopen_file_image(buf_ptr, buf_len, H5LT_FILE_IMAGE_ALLOW_WRITE);

<discard buf_ptr>
<use core file as desired, and then close>

```

In the above example, the core file driver allocates its own buffer, initializes it from the initial image, and then proceeds as usual, reallocating the private buffer as needed as the image grows.

7.2 In Memory HDF5 File Image Construction

HDF5 already supports construction of an image of an HDF5 file in central memory. Thus the only issue is how to allow the application access to the image without first writing it to disk.

The new `H5LTget_file_image()` call will allow the application to obtain a copy of the file's image. The following code fragment illustrates its use:

```

<Open and construct the desired file with the core file driver>

H5Fflush(fid);
H5Fget_filesize(fid, &size);
buffer_ptr = malloc(size);
H5LTget_file_image(fid, size, buffer_ptr);

```

While the use of `H5LTget_file_image()` may be acceptable for small images, for large images, the cost of the `malloc()` and `memcpy()` may become excessive. To address this issue, the `H5Pset_file_image_alloc_callbacks()` call allows the application to manage dynamic memory allocation for the library. The following code fragment illustrates its use.

```

void * image_ptr;
void image_free(void *ptr, unsigned file_image_op_flags, void *udata)
{
    if(file_image_op_flags == H5_FILE_IMAGE_FILE_CLOSE)
        *(void **)udata = ptr;
}

```

```
        else
            free(ptr);
    }

    <allocate fapl_id>

    H5Pset_file_image_alloc_callbacks(fapl_id, NULL, NULL, NULL, NULL,
        image_free, &image_ptr);

    <open core file using fapl_id, write file, flush it>

    H5Fget_filesize(fid, &size);

    <close file>
    /* image_ptr now contains a pointer to the final version of the core file */

    <use it, and then discard it via free()>
```

The above code fragment gives the application full ownership of the buffer used by the core file driver after the file is closed. Recall that if read access to the buffer is sufficient, the `H5Fget_vfd_handle()` API call to get access to the core file driver's buffer pointer is an alternate solution.

8 Recommendation

Unless I have missed something major, the API extensions proposed in this RFC should meet LLNL's needs. The main question remaining is how well, and what can be made better. For this I will need another round of reviews.

The reader will note that the set of flags supported in the `H5LOpen_file_image()` call offer more features than are strictly required for LLNL's purposes. However, the overhead for implementing the extra capability should be small and benefit the overall HDF5 community.

Acknowledgements

This work was supported by Lawrence Livermore National Laboratory (LLNL). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author[s] and do not necessarily reflect the views of LLNL.

Revision History

- May 12, 2011:* Version 0 -- sent to Quincey for comment and general direction check.
- May 23, 2011:* Version 1 -- Incorporates comments and course corrections from email exchanges with Mark Miller and other LLNL staff. Added use cases, core allocation callbacks. General rewrite.

- May 25, 2011:* Version 2 -- Incorporates more comments and course corrections from email exchanges with Mark Miller and discussions with Quincey.
- May 26, 2011:* Version 3 – Major overhaul (by Quincey) after discussions with Quincey.
- May 26, 2011:* Version 4 – Tweaks (by Quincey) after discussions w/Mark Miller.