RFC: File Image Operations

John Mainzer

Quincey Koziol

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

This RFC proposes API extensions to support the requested operations.

# Background

On systems with sufficient main memory, the core file driver allows fast access to HDF5 files, by creating images of HDF5 files[[1]](#footnote-2) in RAM, supporting all normal HDF5 operations on the file images, and if desired, storing the HDF5 file images to disk on flush or file close. The core file driver also allows an existing file on disk to be loaded into memory as an HDF5 file image for fast access via the core file driver.

# Goal

In some cases, it would be useful to create an image of an HDF5 file on one process, transmit it to a second process, and then access 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 raises a number of potential problems, the option of copying buffers must also be maintained.

In this RFC, we propose HDF5 API extensions to support these operations on HDF5 file images.

# Use Cases

The following list is an explicit statement of the use cases driving the development of this document. Only the first two use cases are of interest to us at this time, the remaining use case is included only as a design consideration.

Use Case 1: Reading an HDF5 File Image

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

Discussion:

This is mostly a matter of using the core file driver[[2]](#footnote-3) to open and read the image. To minimize overhead, we further require 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.

Examples of the use of the new API calls to support this used case are offered in section 7.1 below.

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.

Discussion:

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 implemented now unless it can be done at little extra cost.

Examples of the use of the new API calls to support this used case are offered in section 7.2 below.

Use Case 3: Template File

Most likely using a file driver other than the core file driver, open a new HDF5 file, with pre-defined group, data type, and possibly data set definitions taken from a template file image.

Discussion:

The objective is to avoid the necessity of numerous HDF5 API calls required to create an initial set of HDF5 file objects. In parallel HDF5, this might be done to allow the application to avoid making a long sequence of collective calls. In the serial case, it might be done to create a standard HDF5 file structure.

There are no plans for implementing support for this use case at present. It is offered only as a design consideration.

An example of the use of the new API calls to support this use case is offered in section 7.4 below.

# 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 time 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 malloc() and 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 memory based 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 API calls so as to avoid breaking existing applications.

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

# 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 conveniently packaged for application developers’ use.

## New Low-Level API Routines

These routines support new functionality that extends the capabilities of the main HDF5 library, allowing an in memory image of an HDF5 file to be opened without requiring file system I/O. The 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.

The requirement that we allow the application to avoid large buffer copies is met by allowing the user to provide image buffer allocation, copy, resize, and free callbacks. The use of these callbacks is potentially complex – the particulars will be discussed in the semantics and examples sections below.

For now, it is sufficient to note that the property list facility in HDF5 was designed for passing data, not consumable resources into API calls. The peculiar ways in which the file image allocation callbacks may be used allows us to avoid either extending the property list structure to handle consumable resources cleanly, or constructing a new facility for the purpose. This puts extra cognitive load on the application programmer, as insight into how the HDF5 property lists functions, and what HDF5 does with file image buffers is required to avoid arcane errors.

The expectation is that this callback interface will not be heavily used, and then only by uncommonly able programmers. If this expectation is inaccurate, the HDF5 help desk will be busy, and we may wish we had decided on some other solution.

Proposed syntax for the new HDF5 Library calls is offered below.

### H5Pset\_file\_image

The H5Pset\_file\_image routine is designed to allow an application to provide an image for a VFD to use as the initial contents of the file. This call is designed initially for use with the core VFD, but it can be used with any VFD that supports using an initial file image when opening a file (see section 5.1.5 below). Calling this routine makes a copy of the file image buffer provided (using the file image callbacks if defined – see section 5.1.3 below) for allocating, copying, and freeing the file image.

Given the tight interaction between the file image callbacks and the file image, we will not allow the file image callbacks in a property list to be changed while a file image is defined.

As shall be discussed below, with properly constructed file image callbacks, it is possible to avoid actually copying the file image. The necessary code is potentially complex. Avoiding arcane errors requires some knowledge of the internals of HDF5. The particulars of this are discussed in greater detail in the semantics and examples sections below.

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 current file image callbacks set for the FAPL, if defined), setting the FAPL’s file image buf\_len to 0 and buf\_ptr to NULL.

### 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 a file. This routine uses the file image callbacks (if defined) when allocating and loading the buffer to return to the application, or malloc and memcpy if the callbacks are undefined.

As before, appropriately defined file image callbacks can allow this function to avoid buffer allocation and memory copy operations. Also as before, doing this without encountering arcane errors requires care and some knowledge of how the HDF5 library manages property lists.

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 a pointer to a copy of the initial image provided in the last call to H5Pset\_file\_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.

### H5Pset\_file\_image\_callbacks

In order to provide an application with control over how file image buffers are managed, callback routines can be set by an application to handle file image buffer allocation, copying, re-allocation and release. These routines are invoked whenever a new file image buffer is allocated, an existing file image buffer is copied or resized, or when a file image buffer is released from use. From the perspective of the HDF5 Library, the operation and return values of the image\_malloc, image\_memcpy, image\_realloc, and image\_free callbacks are identical to those of the corresponding C standard library calls (i.e. malloc, memcpy, realloc and free), although the parameter lists are expanded as described below.

The signature of H5Pset\_file\_image\_callbacks is defined as follows:

typedef enum

{

H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_SET,

H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_COPY,

H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_GET, H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_CLOSE,

H5\_FILE\_IMAGE\_OP\_FILE\_OPEN,

H5\_FILE\_IMAGE\_OP\_FILE\_RESIZE,

H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE

} H5\_file\_image\_op\_t;

herr\_t H5Pset\_file\_image\_callbacks(hid\_t fapl\_id,

void \*(\*image\_malloc)(size\_t size, H5\_file\_image\_op\_t file\_image\_op,

void \*udata),

void \*(\*image\_memcpy)(void \*dest, const void \*src, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata),

void \*(\*image\_realloc)(void \*ptr, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata),

void (\*image\_free)(void \*ptr, H5\_file\_image\_op\_t file\_image\_op,

void \*udata),

void \*(\*udata\_copy)(void \*udata),

void (\*udata\_free)(void \*udata),

void \*udata)

The parameters of H5Pset\_file\_image\_callbacks are defined as follows:

* fapl\_id shall contain the ID of the target file access property list.
* image\_malloc shall contain a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library malloc() call. The parameters of the image\_malloc callback are defined as follows:
  + size will contain the size of the image buffer to allocate, in bytes.
  + file\_image\_op will be set to one of the values of H5\_file\_image\_op\_t indicating the operation being performed on the file image when this callback is invoked. The semantics of the possible values are discussed below.
  + udata will be set to the value passed in for the udata parameter to H5Pset\_file\_image\_callbacks.

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

* image\_memcpy shall contain a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library memcpy() call[[3]](#footnote-4). The parameters of the image\_memcpy callback are defined as follows:
  + dest will contain the address of the buffer into which to copy.
  + src will contain the address of the buffer from which to copy.
  + size will contain the number of bytes to copy.
  + file\_image\_op will be set to one of the values of H5\_file\_image\_op\_t indicating the operation being performed on the file image when this callback is invoked. The semantics of the possible values are discussed below.
  + udata will be set to the value passed in for the udata parameter to H5Pset\_file\_image\_callbacks.

Setting image\_memcpy to NULL will indicate that the HDF5 library should invoke the standard C library memcpy() routine when copying buffers.

* 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 of the image\_realloc callback are defined as follows:
  + ptr will contain the pointer to the buffer being reallocated.
  + size will contain the desired size of the buffer after realloc, in bytes.
  + file\_image\_op will be set to one of the values of H5\_file\_image\_op\_t indicating the operation being performed on the file image when this callback is invoked. The semantics of the possible values are discussed below.
  + udata will be set to the value passed in for the udata parameter to H5Pset\_file\_image\_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.

* 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 of the image\_free callback are defined as follows:
  + ptr will contain the pointer to the buffer being released.
  + file\_image\_op will be set to one of the values of H5\_file\_image\_op\_t indicating the operation being performed on the file image when this callback is invoked. The semantics of the possible values are discussed below.
  + udata will be set to the value passed in for the udata parameter to H5Pset\_file\_image\_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.

* udata\_copy shall contain a pointer to a function that (from the perspective of HDF5) will allocate a buffer of suitable size, copy the contents supplied udata into the new buffer, and return the address of the new buffer. This function is necessary if a non-NULL udata parameter is supplied, so that property lists containing the image callbacks can be copied. If the udata parameter (below) is NULL, this parameter may be NULL as well. The parameter of the udata\_copy callback is defined as follows:
  + udata will contain the pointer to the user data block being copied.
* udata\_free shall contain a pointer to a function that (from the perspective of HDF5) will free a user data block. This function is necessary if a non-NULL udata parameter is supplied, so that property lists containing image callbacks can be discarded without a memory leak. If the udata parameter (below) is NULL, this parameter may be NULL as well. The parameter of the udata\_free callback is defined as follows:
  + udata will contain the pointer to the user data block to be freed.
* udata shall contain a pointer value, potentially to user-defined data, that will be passed to the image\_alloc, image\_memcpy, image\_realloc, and image\_free callbacks.

The semantics of the values that can be set for the file\_image\_op parameter to the above callbacks are defined as follows:

* H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_SET – This value is passed to the image\_malloc and image\_memcpy callbacks when an image buffer is being copied while being set in a FAPL.
* H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_COPY – This value is passed to the image\_malloc and image\_memcpy callbacks when an image buffer is being copied when a FAPL is copied.
* H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_GET – This value is passed to the image\_malloc and image\_memcpy callbacks when an image buffer is being copied while being retrieved from a FAPL.
* H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_CLOSE – This value is passed to the image\_free callback when an image buffer is being released during a FAPL close operation.
* H5\_FILE\_IMAGE\_OP\_FILE\_OPEN – This value is passed to the image\_malloc, and image\_memcpy callback when an image buffer is copied during a file open operation.
* H5\_FILE\_IMAGE\_OP\_FILE\_RESIZE – This value is passed to the image\_realloc callback when a file driver needs to resize an image buffer
* H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE – This value is passed to the image\_free callback when an image buffer is being released during a file close operation.

In closing our discussion of H5Pset\_file\_image\_callbacks(), we note the interaction between this call and the H5Pget/set\_file\_image() calls above: since the malloc, memcpy, and free callbacks are used by H5Pget/set\_file\_image(), H5Pset\_file\_image\_callbacks() will fail if a file image is already set in the target property list.

### H5Pget\_file\_image\_callbacks()

The H5Pget\_file\_image\_callbacks routine is designed to obtain the current file image callbacks from a file access property list.

The signature of H5Pget\_file\_image\_callbacks() is defined as follows

herr\_t H5Pget\_file\_image\_callbacks(hid\_t fapl\_id,

void \*(\*\*image\_malloc\_ptr)(size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata),

void \*(\*\*image\_memcpy\_ptr)(void \*dest, const void \*src, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata),

void \*(\*\*image\_realloc\_ptr)(void \*ptr, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata),

void (\*\*image\_free\_ptr)(void \*ptr,

H5\_file\_image\_op\_t file\_image\_op, void \*udata),

void \*(\*\*udata\_copy\_ptr)(void \*udata),

void (\*\*udata\_free\_ptr)(void \*udata),

void \*\*udata\_ptr)

The parameters of H5Pget\_file\_image\_callbacks() are defined as follows:

* fapl\_id shall contain the ID of the target file access property list.
* image\_malloc\_ptr shall contain a pointer to pointer to function. Upon successful return, \*image\_malloc\_ptr shall contain the pointer passed as the image\_malloc parameter in the last call to H5Pset\_file\_image\_callbacks() for the specified FAPL, or NULL if there has been no such call.
* image\_memcpy\_ptr shall contain a pointer to pointer to function. Upon successful return, \*image\_memcpy\_ptr shall contain the pointer passed as the image\_memcpy parameter in the last call to H5Pset\_file\_image\_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\_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\_callbacks() for the specified FAPL, or NULL if there has been no such call.
* udata\_copy\_ptr shall contain a pointer to pointer to function. Upon successful return, \*udata\_copy\_ptr shall contain the pointer passed as the udata\_copy parameter in the last call to H5Pset\_file\_image\_callbacks() for the specified FAPL, or NULL if there has been no such call.
* udata\_free\_ptr shall contain a pointer to pointer to function. Upon successful return, \*udata\_free\_ptr shall contain the pointer passed as the udata\_free parameter in the last call to H5Pset\_file\_image\_callbacks() for the specified FAPL, or NULL if there has been no such call.
* udata\_ptr shall contain a pointer to pointer to a void\*. Upon successful return, \*udata\_ptr shall contain the pointer passed as the udata parameter in the last call to H5Pset\_file\_image\_callbacks() for the specified FAPL, or NULL if there has been no such call.

### New Virtual File Driver Feature Flags

Implementation of the above H5Pget/set\_file\_image\_callbacks() and H5Pget/set\_file\_image() API additions requires a pair of new virtual file driver feature flags. Both of these will be defined in H5FDpublic.h

The first of these, the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag, will allow a file driver to indicate whether or not it supports file images. A VFD that sets the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag when its ‘query’ callback is invoked indicates that the file image set in the FAPL will be used as the initial contents of a file.[[4]](#footnote-5)

However, such a VFD need not employ the file image after file open time – indeed any file driver that doesn’t maintain an image of the file in memory will not. In such cases, the VFD will not make an in memory copy of the file image, and will not employ the file image callbacks.

However, file drivers that maintain a copy of the file in memory (only the core file driver at present) can be constructed to use the initial image callbacks (if defined). Those that do must set the H5FD\_FEAT\_CAN\_USE\_FILE\_IMAGE\_CALLBACKS flag when their ‘query’ callbacks are invoked.

Thus file drivers that set the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag but not the H5FD\_FEAT\_CAN\_USE\_FILE\_IMAGE\_CALLBACKS flag may read the supplied image from the property list (if present) and use it to initialize the contents of the file. However, they will not discard the image when done, nor will they make any use of any file image callbacks (if defined).

If an initial file image appears in a file allocation property list that is used in a H5Fopen(), and the underlying file driver does not set the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag, the open will fail.

If a driver sets both the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag and the H5FD\_FEAT\_CAN\_USE\_FILE\_IMAGE\_CALLBACKS flag, that driver will allocate a buffer of the required size, copy the contents of the initial image buffer from the file access property list, and then open the copy as if it had just loaded it from file. If the file image allocation callbacks are defined, the driver shall use them for all memory management tasks. Otherwise it will use the standard malloc, memcpy, realloc, and free C library calls for this purpose.

Two points in closing this section:

If the VFD sets 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.

Finally, as the astute reader will already have observed, it is logically possible that a file driver would set the H5FD\_FEAT\_CAN\_USE\_FILE\_IMAGE\_CALLBACKS flag, but not the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag. While it is hard to think of a situation in which this would be desirable, it will not cause any problems if the situation arises, as the two capabilities are logically distinct.

## New High-Level API Routines

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

### H5LTopen\_file\_image

The H5LTopen\_file\_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)[[5]](#footnote-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 value of 0 is invalid and will cause H5LTopen\_file\_image to fail.
* flags shall contain a set of flags indicating whether the image is to be opened read/write, 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\_OPEN\_RW – Indicates that the HDF5 library should open the image read/write instead of the default read only.
  + 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. 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.[[6]](#footnote-7)

The HDF5 library will modify the buffer on write if the image is opened R/W and the H5LT\_FILE\_IMAGE\_DONT\_COPY flag is set.

The following H5LT\_FILE\_IMAGE\_DONT\_RELEASE flag is invalid unless the H5LT\_FILE\_IMAGE\_DONT\_COPY flag is set.

* + H5LT\_FILE\_IMAGE\_DONT\_RELEASE - This flag indicates that the HDF5 library should not attempt to release the buffer when the file is closed. This implies that the application will tend to this detail, and that the application will not discard the buffer until after the file image is closed.

Since there is no way to return a changed buffer base address to the application, and since realloc can change this value, calls to realloc() must be barred when this flag is set. As a result, any write that requires an increased buffer size will fail.

This flag is invalid unless the H5LT\_FILE\_IMAGE\_DONT\_COPY flag is set.

If the H5LT\_FILE\_IMAGE\_DONT\_COPY flag is set in the absence of this flag, the HDF5 library will release the buffer after the file is closed, using the standard C library free() routine.[[7]](#footnote-8)

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.

### 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 on 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 file[[8]](#footnote-9)), or a negative value if the file is too large to store in the supplied buffer or on failure.

# New API Call Semantics

## File Image Callback Semantics

The H5Fget/set\_file\_image\_callbacks() API calls allow the application to hook the memory management operations used when allocating, duplicating, and discarding file images in the property list, in the core file driver, and if desired, in any future in memory file driver.

From the perspective of the HDF5 library, the supplied image\_alloc(), image\_memcpy(), image\_realloc(), and image\_free() callback routines must function identically to the C standard library malloc(), memcpy(), 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. However, whatever the application does with these calls, it must maintain the illusion that the calls have had the expected effect. Maintaining this illusion requires some understanding of how the property list structure works, and what HDF5 will do with the initial images passed to it.

### Property List Considerations

The HDF5 property lists are a mechanism for passing values into HDF5 library calls. They were created to allow calls to be extended with new parameters without changing the actual API or breaking existing code. However, they were designed with the assumption that all new parameters would be “call by value” not “call by reference”. This assumption allows property lists to be copied, reused, and discarded with ease – and is also the source of the requirement that, from the perspective of the HDF5 library, file image buffers inserted into or extracted from a property list via the H5Pset\_file\_image()/H5Pget\_file\_image() calls be copied into a newly allocated buffer.

The file image callbacks allow the application programmer to subvert this design assumption. However, this must be done with care, lest unexpected results ensue. In particular, the application developer must ensure that the value passed by reference is not used more than once – unless that is the desired effect. Further, the application must ensure that the target of the reference remains in existence until there is no possibility of the HDF5 library referring to it. This requires some care, as the property list code will attempt to free its copy when the property list is discarded.

Turning from generalities to the first use case described in section 3, if the application wishes to share a buffer with the HDF5 library, allowing it to open the image, but not copy or free it, the file image callbacks might be constructed as follows:

1. Construct the image\_malloc() call so that it returns the address of the buffer, instead of allocating new space. Support this by including the address of the buffer in the user data. As a sanity check, include the buffer’s size in the user data as well, and require image\_malloc() to fail if the requested buffer size is unexpected. Finally, include a reference counter in the user data, and increment it on each call to image\_malloc().
2. Construct the image\_memcpy() call so that it does nothing. As a sanity check, make it fail if the source and destination pointers don’t match the buffer address in the user data, or if the size is unexpected.
3. Construct the image\_free() routine that does nothing. As a sanity check, make it compare the supplied pointer with the expected pointer in the user data. Also, make it decrement the reference counter, and notify the application that the HDF5 library is done with the buffer when the reference count drops to 0.

As the property list code will never resize a buffer, we do not discuss the image\_realloc() call here. Its behavior in this scenario depends on what the application wants to do with the file image after it has been opened – an issue we discuss in the next section. Note also that the op passed into the file image callbacks allow the callbacks to behave differently depending on the context in which they are used.

### File Driver Considerations

When an image is opened by a driver that sets both the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE and the H5FD\_FEAT\_CAN\_USE\_FILE\_IMAGE\_CALLBACKS flags, the driver will allocate a buffer large enough for the initial file image, and then copy the image from the property list into its buffer. As processing progresses, it will re-allocate the image as necessary to increase its size, and eventually discard the image at file close. If defined, it will use the file image callbacks for this purpose – otherwise it will use the standard C library calls (malloc, memcpy, realloc, and free).

As before, the file image callbacks can be constructed so as to avoid the overhead of buffer allocations and copies while allowing the HDF5 library to maintain its illusions on the subject. However, the problem is slightly more involved in this case, as the possibility of realloc calls from the driver complicates matters, as does the possibility of the continued existence of property lists containing references to the buffer.

Returning to our example in which the application wishes to share a buffer with the HDF5 library, allowing it to open the image, but not copy or free it, we must first decide whether the image is to be opened read only or read/write.

If the image is to be opened read only (or if we know that any writes will not change the size of the image), the image\_realloc() call should never be invoked. Thus the image\_realloc() routine can be constructed so as to always fail, and the image\_malloc(), image\_memcpy(), and image\_free() routines can be as described in the property list considerations section above.

In the more interesting case where the image is opened read/write, and may grow during the computation, we must adjust our sanity checks to allow for the case in which the base address of the buffer changes due to realloc calls, and employ the user data to communicate any change in the buffer base address and size to the application (which must free the buffer eventually). To this end, we might define a user data structure as follows:

typedef struct udata {

void \*init\_ptr;

size\_t init\_size;

int init\_ref\_count;

void \*mod\_ptr;

size\_t mod\_size;

int mod\_ref\_count;

}

We initialize an instance of the structure so that init\_ptr points to the buffer to be shared, init\_size contains the initial size of the buffer, and all other fields are initialized to either NULL or 0 as indicated by their type. We then pass a pointer to the instance of the user data structure to the HDF5 library along with allocation callback functions constructed as follows:

1. Construct the image\_malloc() call so that it returns the value in the init\_ptr field of the user data, and increments the init\_ref\_count. As a sanity check, the function should fail if the requested size does not match the init\_size field in the user data, or if any of the mod fields have values other than their initial values.
2. Construct the image\_memcpy() call so that it does nothing. As a sanity check it should be made to fail if the source, destination, and size parameters don’t match the init\_ptr and init\_size fields as appropriate.
3. Construct the image\_realloc() call so that it performs a standard realloc. Sanity checking, assuming that the realloc is successful, should be as follows:

If the mod\_ptr/mod\_size/mod\_ref\_count fields of the user data still have their initial values, verify that the supplied pointer matches the init\_ptr field, and the supplied size does not match the init\_size field. Decrement init\_ref\_count, set mod\_ptr equal to the address returned by the realloc, set mod\_size equal to the supplied size, and set mod\_ref\_count to 1.

If the mod\_ptr/mod\_size/mod\_ref\_count fields of the user data are defined, verify that the supplied pointer matches the mod\_ptr and that the supplied size does not match the mod\_size. Set mod\_ptr equal to the value returned by realloc, and set mod\_size equal to the supplied size.

In both cases, if all sanity checks pass, return the value returned by the realloc call. Otherwise, return NULL.

1. Construct the image\_free() routine so that it does nothing. Perform sanity checks as follows:

If the H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_CLOSE flag is set, decrement the init\_ref\_count field of the user data. Flag an error if this field drops below zero.

If the H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE flag is set, check to see if the mod\_ptr/mod\_size/mod\_ref\_count fields of the user data have been modified from their initial values. If they have, verify that mod\_ref\_count contains 1, and then set that field to zero. If they haven’t been modified, proceed as per the H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_CLOSE case.

In either case, if both the init\_ref\_count and mod\_ref\_count fields have dropped to zero, notify the application that the HDF5 library is done with the buffer. If the mod\_ptr/mod\_size fields have been modified, pass these values on to the application as well.

### Other Considerations

In the previous subsections, we have looked at only one of the many ways in which the file image callbacks could be used, and how this interacted with the property list facility and the core file driver. Despite our restricted exploration, we hope we have made the point that the file image callbacks must be used with care, and with understanding of how they will be employed in the HDF5 library.

In this final subsection, we consider some of the other implications of our decision to encourage the use of the image allocation callback functions to trick the HDF5 library into thinking copies of buffers have been made when they have not.

The most obvious of these is documentation – if the application developer is to use the file image callbacks to trick the HDF5 library, he must know what the library will do with them.

Another implication is that by providing the file image callbacks, we are locking ourselves into a particular pattern of use in the management of initial images in the property lists, and the buffer in the core file driver. If we change these patterns, we risk breaking applications that use the file image callbacks.

A final issue to consider is whether we should standardize these patterns of image allocation callback use across all memory based file drivers that support initial images? If we don’t, we may force the application developer to tailor his set of file image callbacks to the intended VFD. If we do, we may force file drivers to do things that aren’t appropriate to them.

## 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. However, the consensus seems to be that it is closer to a file open and thus we shall require that the initial image only be used 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.[[9]](#footnote-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 set the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE flag and a file image is specified in the FAPL.
2. If the target file driver supports the H5FD\_FEAT\_ALLOW\_FILE\_IMAGE 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.

As we indicated earlier, if an initial file image appears in the property list of a H5Fcreate() call, it is ignored.

While the above section on the semantics of the file image callbacks may have seemed rather gloomy, we get the payback here. The above says everything that needs to be said about initial file image semantics.

# Examples of Application of API Changes to the Primary Use Cases

Only the first two of the use cases listed in section 3 are of interest to us at present – thus only these two cases are addressed in detail. While we do address the third use case, we do so only briefly as it is tangential to our immediate objectives.

## Reading an in Memory HDF5 File Image

HDF5 already allows the core file driver to be initialized from a file. 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);

<discard buf any time after this point>

<open file>

<discard fapl any time after this point>

<read and/or write file as desired, close>

Observe that while this solution is easy to code, the supplied buffer is duplicated twice. Once in the call to H5Pset\_file\_image() when the image is duplicated, and the duplicate inserted into the property list, and again on file open, when the image is copied from the property list into the initial buffer allocated by the core file driver. This is a non-issue for small images, but could become a significant performance hit for large ones.

If we want to avoid the extra malloc and memcpy calls, we must decide whether the application should retain ownership of the buffer, or pass ownership to the HDF5 library.

The following pseudo code illustrates opening the image read only using the H5LTopen\_file\_image() routine, with the application retaining ownership of the buffer – thereby avoiding any extra buffer allocations and memcpy calls.

<allocate and initialize buf\_len and buf>

hid\_t file\_id;

unsigned flags = H5LT\_FILE\_IMAGE\_DONT\_COPY | H5LT\_FILE\_IMAGE\_DONT\_RELEASE;

file\_id = H5LTopen\_file\_image(buf, buf\_len, flags);

<read file as desired, and then close>

<discard buf any time after this point>

If the application want to transfer ownership of the buffer to the library, and the standard C library routine free is an acceptable way of discarding it, the above example can be modified as follows:

<allocate and initialize buf\_len and buf>

hid\_t file\_id;

unsigned flags = H5LT\_FILE\_IMAGE\_DONT\_COPY;

file\_id = H5LTopen\_file\_image(buf, buf\_len, flags);

<read file as desired, and then close>

Again, file access is read only. Read/Write access can be obtained via the H5LTopen\_file\_image() call, but we will explore that in the next use case.

## In Memory HDF5 File Image Construction

HDF5 already supports construction of an image of an HDF5 file in memory with the core file driver. 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, buffer\_ptr, size);

While the use of H5LTget\_file\_image() may be acceptable for small images, for large images, the cost of the malloc() and memcpy() may be excessive. To address this issue, the H5Pset\_file\_image\_callbacks() call allows the application to manage dynamic memory allocation for file images and memory based file drivers (only the core file driver at present). The following code fragment illustrates its use. Note that most error checking is omitted for simplicity and that H5Pset\_file\_image is *not* used to set the initial file image.

struct udata\_t {

void \* image\_ptr;

size\_t image\_size;

} udata = {NULL, 0};

void image\_malloc(size\_t size, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

((struct udata\_t \*)udata)->image\_size = size;

return(malloc(size));

}

void \*image\_memcpy)(void \*dest, const void \*src, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(FALSE); /\* Should never be invoked in this scenario. \*/

return(dest);

}

void image\_realloc(void \*ptr, size\_t size, H5\_file\_image\_op\_t file\_image\_op,

void \*udata)

{

((struct udata\_t \*)udata)->image\_size = size;

return(realloc(ptr, size));

}

void image\_free(void \*ptr, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(file\_image\_op == H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE);

((struct udata\_t \*)udata)->image\_ptr = ptr;

return;

}

void \*udata\_copy(void \*udata)

{

return(udata);

}

void udata\_free(void \*udata)

{

return;

}

<allocate fapl\_id>

H5Pset\_file\_image\_callbacks(fapl\_id, image\_malloc, image\_memcpy, image\_realloc,

image\_free, udata\_copy, udata\_free, &udata);

<open core file using fapl\_id, write file, close it>

assert(udata.image\_ptr!= NULL);

/\* udata now contains the base address and length of the final

version of the core file \*/

<use image of file, 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, and notifies the application that the HDF5 library is done with the buffer by setting udata.image\_ptr to something other than NULL . Recall that if read access to the buffer is sufficient, the H5Fget\_vfd\_handle() API call (to get access to the base address of the core file driver’s buffer) is an alternate solution.

The above solution avoids some unnecessary malloc and memcpy calls, and should be quite adequate if an image of an HDF5 file is constructed only occasionally. However, if an HDF5 file image must be constructed regularly, and we can put a strong and tight upper bound on the size of the necessary buffer, the following pseudo code demonstrates a method of avoiding memory allocation completely – albeit at the cost of allocating the buffer statically. Again, much error checking is omitted for clarity.

char buf[BIG\_ENOUGH];

struct udata\_t {

void \* image\_ptr;

size\_t image\_size;

size\_t max\_image\_size;

int ref\_count;

} udata = {(void \*)(&(buf[0])), 0, BIG\_ENOUGH, 0};

void \*image\_malloc(size\_t size, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(size <= ((struct udata\_t \*)udata)->max\_image\_size);

assert(((struct udata\_t \*)udata)->ref\_count == 0);

((struct udata\_t \*)udata)->image\_size = size;

(((struct udata\_t \*)udata)->ref\_count)++;

return((((struct udata\_t \*)udata)->image\_ptr);

}

void \*image\_memcpy)(void \*dest, const void \*src, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(FALSE); /\* Should never be invoked in this scenario. \*/

return(dest);

}

void \*image\_realloc(void \*ptr, size\_t size, H5\_file\_image\_op\_t file\_image\_op,

void \*udata)

{

assert(ptr == ((struct udata\_t \*)udata)->image\_ptr);

assert(size <= ((struct udata\_t \*)udata)->max\_image\_size);

assert(((struct udata\_t \*)udata)->ref\_count == 1);

((struct udata\_t \*)udata)->image\_size = size;

return((((struct udata\_t \*)udata)->image\_ptr);

}

void image\_free(void \*ptr, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(file\_image\_op == H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE);

assert(ptr == ((struct udata\_t \*)udata)->image\_ptr);

assert(((struct udata\_t \*)udata)->ref\_count == 1);

(((struct udata\_t \*)udata)->ref\_count)--;

return;

}

void \*udata\_copy(void \*udata)

{

return(udata);

}

void udata\_free(void \*udata)

{

return;

}

/\* end of initialization \*/

<allocate fapl\_id>

H5Pset\_file\_image\_callbacks(fapl\_id, image\_malloc, image\_memcpy,

image\_realloc, image\_free,

udata\_copy, udata\_free, (void \*)(&udata));

<open core file using fapl\_id>

<discard fapl any time after the open>

<write the file, flush it, and then close it>

assert(udata.ref\_count == 0);

/\* udata now contains the base address and length of the final

version of the core file \*/

<use the image of the file>

<reinitialize udata, and repeat the above from the end of initialization onwards to write a new file image>

If we can further arrange matters so that only the contents of the datasets in the HDF5 file image change, but not the structure of the file itself, we can optimize still further by re-using the image and changing only the contents of the datasets after the initial write to the buffer. The following pseudo code shows how this might be done – note that it assumes that buf already contains the image of the HDF5 file whose dataset contents are to be overwritten. Again, much error checking is omitted for clarity. Also, observe that the file image callbacks do not support the H5Pget\_file\_image() call.

<buf already defined and loaded with file image>

<udata already defined and initialized>

void \*image\_malloc(size\_t size, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(size <= ((struct udata\_t \*)udata)->max\_image\_size);

assert(size == ((struct udata\_t \*)udata)->image\_size);

assert(((struct udata\_t \*)udata)->ref\_count >= 0);

((struct udata\_t \*)udata)->image\_size = size;

(((struct udata\_t \*)udata)->ref\_count)++;

return((((struct udata\_t \*)udata)->image\_ptr);

}

void \*image\_memcpy)(void \*dest, const void \*src, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert(dest == ((struct udata\_t \*)udata)->image\_ptr);

assert(src == ((struct udata\_t \*)udata)->image\_ptr);

assert(size <= ((struct udata\_t \*)udata)->max\_image\_size);

assert(size == ((struct udata\_t \*)udata)->image\_size);

assert(((struct udata\_t \*)udata)->ref\_count >= 1);

return(dest);

}

void \*image\_realloc(void \*ptr, size\_t size, H5\_file\_image\_op\_t file\_image\_op,

void \*udata)

{

assert(FALSE); /\* should never be invoked in this scenario \*/

return(NULL);

}

void image\_free(void \*ptr, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

assert((file\_image\_op == H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_CLOSE) ||

(file\_image\_op == H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE));

assert(((struct udata\_t \*)udata)->ref\_count >= 1);

(((struct udata\_t \*)udata)->ref\_count)--;

return;

}

void \*udata\_copy(void \*udata)

{

return(udata);

}

void udata\_free(void \*udata)

{

return;

}

/\* end of initialization \*/

<allocate fapl\_id>

H5Pset\_file\_image\_callbacks(fapl\_id, image\_malloc, image\_memcpy,

image\_realloc, image\_free, udata\_copy, udata\_free, (void \*)(&udata));

H5Pset\_file\_image(fapl\_id, udata.image\_ptr, udata.image\_len);

<open core file using fapl\_id>

<discard fapl any time after the open>

<overwrite data in datasets in the file, and then close it>

assert(udata.ref\_count == 0);

/\* udata now contains the base address and length of the final

version of the core file \*/

<use the image of the file>

<repeat the above from the end of initialization onwards to write new data to datasets in file image>

Before we go on, we should note that the above pseudo code can be written more compactly, albeit with fewer sanity checks, using the H5LTopen\_file\_image() call:

<buf already defined and loaded with file image>

<udata already defined and initialized>

hid\_t file\_id;

unsigned flags = H5LT\_FILE\_IMAGE\_OPEN\_RW | H5LT\_FILE\_IMAGE\_DONT\_COPY |

H5LT\_FILE\_IMAGE\_DONT\_RELEASE;

/\* end initialization \*/

file\_id = H5LTopen\_file\_image(udata.image\_ptr, udata.image\_len, flags);

<overwrite data in datasets in the file, and then close it>

/\* udata now contains the base address and length of the final

version of the core file \*/

<use the image of the file>

<repeat the above from the end of initialization onwards to write new data to datasets in file image>

The above pseudo code allows updates of a file image about as cheaply as possible – assuming the application has the RAM to spare for the image, and assuming that HDF5 file structure is constant after the first write.

While this is a plausible scenario, we will finish this section with a more general scenario, in which we assume sufficient RAM to retain the HDF5 file image between uses, but do not assume that the HDF5 file structure remains constant or that we can place a hard upper bound on the image size.

Since we must use malloc/realloc/free in this example, and since realloc can change the base address of a buffer, we must maintain two of ptr/size/ref\_count triples in the user data – one for the property list (which will never change the buffer), and one for the file driver. As shall be seen, this complicates the file image callbacks considerably.

Also, while we don’t use H5Pget\_file\_image() in this example, we include support for it in the file image callbacks.

As usual, much error checking is omitted in favor of clarity.

struct udata\_t {

void \* fapl\_image\_ptr;

size\_t fapl\_image\_size;

int fapl\_ref\_count;

void \* vfd\_image\_ptr;

size\_t vfd\_image\_size;

int vfd\_ref\_count;

} udata = {NULL, 0, 0, NULL, 0, 0};

boolean initial\_file\_open = TRUE;

void \*image\_malloc(size\_t size, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

void \* return\_value = NULL;

switch ( file\_image\_op ) {

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_SET:

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_COPY:

assert(((struct udata\_t \*)udata)->fapl\_image\_ptr != NULL);

assert(((struct udata\_t \*)udata)->fapl\_image\_size == size);

assert(((struct udata\_t \*)udata)->fapl\_ref\_count >= 0);

return\_value = ((struct udata\_t \*)udata)->fapl\_image\_ptr;

(((struct udata\_t \*)udata)->fapl\_ref\_count)++;

break;

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_GET:

assert(((struct udata\_t \*)udata)->fapl\_image\_ptr != NULL);

assert(((struct udata\_t \*)udata)->vfd\_image\_size == size);

assert(((struct udata\_t \*)udata)->fapl\_ref\_count >= 1);

return\_value = ((struct udata\_t \*)udata)->fapl\_image\_ptr;

/\* don’t increment ref count \*/

break;

case H5\_FILE\_IMAGE\_OP\_FILE\_OPEN:

assert(((struct udata\_t \*)udata)->vfd\_image\_ptr == NULL);

assert(((struct udata\_t \*)udata)->vfd\_image\_size == 0);

assert(((struct udata\_t \*)udata)->vfd\_ref\_count == 0);

if (((struct udata\_t \*)udata)->fapl\_image\_ptr == NULL ) {

((struct udata\_t \*)udata)->vfd\_image\_ptr =

malloc(size);

((struct udata\_t \*)udata)->vfd\_image\_size = size;

} else {

assert(((struct udata\_t \*)udata)->fapl\_image\_size ==

size);

assert(((struct udata\_t \*)udata)->fapl\_ref\_count >=

1);

((struct udata\_t \*)udata)->vfd\_image\_ptr =

((struct udata\_t \*)udata)->fapl\_image\_ptr;

((struct udata\_t \*)udata)->vfd\_image\_size = size;

}

return\_value = ((struct udata\_t \*)udata)->vfd\_image\_ptr;

(((struct udata\_t \*)udata)->vfd\_ref\_count)++;

break;

default:

assert(FALSE);

}

return(return\_value);

}

void \*image\_memcpy)(void \*dest, const void \*src, size\_t size,

H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

switch(file\_image\_op) {

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_SET:

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_COPY:

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_GET:

assert(dest == ((struct udata\_t \*)udata)->fapl\_image\_ptr);

assert(src == ((struct udata\_t \*)udata)->fapl\_image\_ptr);

assert(size == ((struct udata\_t \*)udata)->fapl\_image\_size);

assert(((struct udata\_t \*)udata)->fapl\_ref\_count >= 1);

break;

case H5\_FILE\_IMAGE\_OP\_FILE\_OPEN:

assert(dest == ((struct udata\_t \*)udata)->vfd\_image\_ptr);

assert(src == ((struct udata\_t \*)udata)->fapl\_image\_ptr);

assert(size == ((struct udata\_t \*)udata)->fapl\_image\_size);

assert(size == ((struct udata\_t \*)udata)->vfd\_image\_size);

assert(((struct udata\_t \*)udata)->fapl\_ref\_count >= 1);

assert(((struct udata\_t \*)udata)->vfd\_ref\_count == 1);

break;

default:

assert(FALSE);

break;

}

return(dest);

}

void \*image\_realloc(void \*ptr, size\_t size, H5\_file\_image\_op\_t file\_image\_op,

void \*udata)

{

assert(ptr == ((struct udata\_t \*)udata)->vfd\_image\_ptr);

assert(((struct udata\_t \*)udata)->vfd\_ref\_count == 1);

((struct udata\_t \*)udata)->vfd\_image\_ptr = realloc(ptr, size);

((struct udata\_t \*)udata)->vfd\_image\_size = size;

return((((struct udata\_t \*)udata)->vfd\_image\_ptr);

}

void image\_free(void \*ptr, H5\_file\_image\_op\_t file\_image\_op, void \*udata)

{

switch(file\_image\_op) {

case H5\_FILE\_IMAGE\_OP\_PROPERTY\_LIST\_CLOSE:

assert(ptr == ((struct udata\_t \*)udata)->fapl\_image\_ptr);

assert(((struct udata\_t \*)udata)->fapl\_ref\_count >= 1);

(((struct udata\_t \*)udata)->fapl\_ref\_count)--;

break;

case H5\_FILE\_IMAGE\_OP\_FILE\_CLOSE:

assert(ptr == ((struct udata\_t \*)udata)->vfd\_image\_ptr);

assert(((struct udata\_t \*)udata)->vfd\_ref\_count == 1);

(((struct udata\_t \*)udata)->vfd\_ref\_count)--;

break;

default:

assert(FALSE);

break;

}

return;

}

void \*udata\_copy(void \*udata)

{

return(udata);

}

void udata\_free(void \*udata)

{

return;

}

/\* end of initialization \*/

<allocate fapl\_id>

H5Pset\_file\_image\_callbacks(fapl\_id, image\_malloc, image\_memcpy,

image\_realloc, image\_free,

udata\_copy, udata\_free, (void \*)(&udata));

if ( initial\_file\_open ) {

initial\_file\_open = FALSE;

} else {

assert(udata.vfd\_image\_ptr != NULL);

assert(udata.vfd\_image\_size > 0);

assert(udata.vfd\_ref\_count == 0);

assert(udata.fapl\_ref\_count == 0);

udata.fapl\_image\_ptr = udata.vfd\_image\_ptr;

udata.fapl\_image\_size = udata.vfd\_image\_size;

udata.vfd\_image\_ptr = NULL;

udata.vfd\_image\_size = 0;

H5Pset\_file\_image(fapl\_id, udata.fapl\_image\_ptr, udata.fapl\_image\_size);

}

<open core file using fapl\_id>

<discard fapl any time after the open>

<write/update the file, and then close it>

assert(udata.fapl\_ref\_count == 0);

assert(udata.vfd\_ref\_count == 0);

/\* udata.vfd\_image\_ptr and udata.vfd\_image\_size now contain the base address

and length of the final version of the core file \*/

<use the image of the file>

<repeat the above from the end of initialization to modify the file image as needed>

<free the image when done>

The above pseudo code shows how a buffer can be passed back and forth between the application and the HDF5 library, with the application having control of the actual allocation, reallocation, and freeing of the buffer. While numerous other scenarios are possible, this example demonstrates the limits of the capabilities of the proposed API extensions.

## Using HDF5 to Construct and Read a Data Packet

Use cases 1 and 2 are really just two halves of the main use case, which is to bundle up data in an image of an HDF5 file on one process, transmit the image to a second process, then open and read the image on the second process without any mandatory file system I/O.

We have already demonstrated the construction and reading of such buffers above, but it may be useful to offer an example of the full operation. We do so below, using as simple a set of calls as possible, as we have already demonstrated how we can avoid unnecessary buffer allocations and copies.

In the following example, we construct a HDF5 file image on process A, transmit the image to process B, where we then open the image and extract the desired data. Note that no file system I/O is performed – everything is done in memory with the core file driver.

\*\*\* Process A \*\*\* \*\*\* Process B \*\*\*

<Open and construct the desired file hid\_t file\_id;

with the core file driver>

H5Fflush(fid);

H5Fget\_filesize(fid, &size);

buffer\_ptr = malloc(size);

H5LTget\_file\_image(fid, buffer\_ptr,

size);

<transmit size> <receive size>

buffer\_ptr = malloc(size)

<transmit \*buffer\_ptr> <receive image in \*buffer\_ptr>

free(buffer\_ptr);

<close core file> file\_id = H5LTopen\_file\_image(buf,

buf\_len,

H5LT\_FILE\_IMAGE\_DONT\_COPY);

<read data from file, then close.

note that the core file driver

will discard the buffer on close>

## Use of a Template File

After the above examples, an example of the use of a template file either to enforce consistency on file structure between files, or (in the parallel HDF5 case) to avoid long sequences of collective operations to create the desired groups, datatypes, and possibly datasets may seem anti-climactic. The following pseudo code outlines a possible use:

<allocate and initialize buf and buflen, with buf containing the desired initial image (which in turn contains the desired group, datatype, and dataset definitions), and buf\_len containing the size of buf>

<allocate fapl\_id>

<set fapl to use desired file driver that supports initial images>

H5Pset\_file\_image(fapl\_id, buf, buf\_len);

<discard buf any time after this point>

<open file>

<discard fapl any time after this point>

<read and/or write file as desired, close>

Observe that the above pseudo code includes an un-necessary buffer allocation and copy in the call to H5Pset\_file\_image(). As we have already discussed ways of avoiding this, we will not address that issue here.

What is interesting in this case is to consider why the application would find this use case attractive.

In the serial case, at first glance there seems little reason to use the initial image facility at all – it is easy enough to use standard C calls to duplicate a template file, rename it as desired, and then open it as an HDF5 file.

However, this assumes that the template file will always be available and in the expected place – a questionable assumption for an application that will be widely distributed. Thus we can at least make an argument for either keeping an image of the template file in the executable or for including code for writing the desired standard definitions to new HDF5 files.

Assuming the image is relatively small, we can further make an argument for the image in place of the code, as, quite simply, the image should be easier to maintain and modify with a HDF5 file editor.

However, there remains the question of why one should pass the image to the HDF5 library instead of writing it directly with standard C calls and then using HDF5 to open it. Other than convenience and a slight reduction in code size, we are hard pressed to offer a reason. If offered, the feature will probably be used now and then, but we doubt that anyone will complain if it is unavailable in the serial case.

In contrast, the argument is stronger in the parallel case, as group, datatype, and dataset creations are all collective operations -- which are expensive. It is also weaker, as simply copying an existing template file and opening it should lose many of its disadvantages in the HPC context – although we would imagine that it is always useful to reduce the number of files in a deployment.

In closing, we would like to consider one last point. In the parallel case, we would expect template files to be quite large, as parallel HDF5 requires eager space allocation for chunked datasets, and for similar reasons, we would expect template files in this context to contain long sequences of zeros, with a scattering of metadata here and there. Such files would compress well, and the compressed images would be cheap to distribute across the available processes if necessary. Once distributed, each process could uncompress the image, and write to file those sections containing actual data that lay within the section of the file assigned to the process. This approach might be significantly faster than a simple copy as it would allow sparse writes, and thus it might provide a compelling use case for template files. However, it does require extending our current API to allow compressed images – possibly through the addition of H5Pget/set\_image\_decompression\_callback() API calls.

We see no problem in doing this. However, it is beyond the scope of the current effort, and thus we will not pursue the matter further unless there is interest in our doing so.

# Recommendations

While the point has already been largely addressed in email exchanges, it would be useful to confirm that the proposed API extensions meet the users’ needs. If there are any tweaks that would make life easier for developers, now is the time to mention them.

Considerable thought should be given to the use of the file image callbacks to avoid memory allocations and buffer copies.

As has been discussed above, this approach to the problem requires tricking the HDF5 library into thinking that these operations have been performed, and thus requires some knowledge of HDF5 internals to avoid unexpected results. It therefore places increased load on the developer, and thus on the HDF5 help desk as well. Finally, it places constraints on future modification of the relevant internals, and on the design of any future memory based file drivers.

On the plus side, it has major advantages in terms of ease of implementation, and allows support of everything we can think of with the exception of a “copy on write” policy for file image buffers opened by memory based file drivers.

If everyone is happy with this trade off, well and good. If not, we should explore other solutions to the problem.[[10]](#footnote-11)

Assuming we are happy with the general approach and functionality, we should specify the associated Fortran and Java interfaces. There doesn’t seem to be any interest in a C++ interface, so we will not address that issue unless we hear requests.

Finally, we need to come to a decision on how much of the above we will be implementing at this time.

# 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 derived 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. |
| *June 22, 2011:* | Version 5 – Major re-write (by John) after discussions with Quincey. |
| *June 30, 2011:* | Version 6 – Tweaks (by John) in response to comments from Quincey. |
| *July 6, 2011:* | Version 7 – Tweaks (by John) in response to comments from Quincey. |
| *July 7, 2011:* | Version 8 – Tweaks (by John) after discussions with Quincey. |
| *July 13, 2011:* | Version 9 – Minor tweaks (by John & Quincey). |
| *July 19, 2011:* | Version 10 – Copy edits (by John & Quincey). First version for wide distribution. |

1. For the purposes of this RFC, a HDF5 file image is simply a buffer in main memory that contains an HDF5 file. The easiest way of creating such an image is to allocate a large enough buffer, and then load an HDF5 file from disk into the buffer. Internally, the core file driver also creates and maintains HDF5 file images. [↑](#footnote-ref-2)
2. Or more generally, any memory based file driver – by which we mean a file driver that maintains its image of the file in memory, without any mandatory file system I/O. At present, the core file driver is the only member of this class, but we should allow for the possibility of other memory based file drivers in the future. [↑](#footnote-ref-3)
3. The phrase “from the perspective of HDF5” is very important here – as in absolute terms, this statement isn’t true. Recall that the memcpy() has undefined behavior when the source and destination buffers overlap. From the HDF5 library’s point of view, this is irrelevant, as the buffers will never overlap. However, the library will not verify that this is in fact the case, and, as shall be seen, it will sometimes be useful to arrange matters so that the source and destination buffers are identical so as to avoid large buffer copies. Needless to say, the behavior of the image\_memcpy() callback cannot be undefined in such cases – it must maintain the illusion that the desired buffer copy has taken place. [↑](#footnote-ref-4)
4. 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). [↑](#footnote-ref-5)
5. 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. [↑](#footnote-ref-6)
6. 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. [↑](#footnote-ref-7)
7. 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, while letting the application retain ownership of the buffer. [↑](#footnote-ref-8)
8. The current file size can also be obtained via a call to H5Fget\_filesize(). [↑](#footnote-ref-9)
9. Of course, the core VFD will not write to the file system unless it is configured to do so. [↑](#footnote-ref-10)
10. The obvious candidates are expanding the property list concept to handle call by reference cleanly, or adding a H5Fopen\_init\_image() call. [↑](#footnote-ref-11)