HDF5 File Image Operations

Contents

1. Introduction to HDF5 File Image Operations

File image operations allow users to work with HDF5 files in memory in the same ways that users currently work with HDF5 files on disk. Disk I/O is not required when file images are opened, created, read from, or written to.

An HDF5 file image is an HDF5 file that is held in a buffer in main memory. Setting up a file image in memory involves using either a buffer in the file access property list or a buffer in the Core (aka Memory) file driver.

The advantage of working with a file in memory is faster access to the data.

The challenge of working with files in memory buffers is maximizing performance and minimizing memory footprint while working within the constraints of the property list mechanism. This should be a non-issue for small file images, but may be a major issue for large images.

If invoked with the appropriate flags, the `H5LTopen_file_image()` high level library call should deal with these challenges in most cases. However, some applications may require the programmer to address these issues directly.

1.1. File Image Operations Function Summary

Functions used in file image operations are listed below.

Function Listing 1. File image operations functions

C Function	Purpose

`H5Pset_file_image`	Allows an application to specify an initial file image. For more information, see section 2.1.1.
`H5Pget_file_image`	Allows an application to retrieve a copy of the file image designated for a VFD to use as the initial contents of a file. For more information, see section 2.1.2.
`H5Pset_file_image_callbacks`	Allows an application to manage file image buffer allocation, copying, reallocation, and release. For more information, see section 2.1.3.
`H5Pget_file_image_callbacks`	Allows an application to obtain the current file image callbacks from a file access property list. For more information, see section 2.1.4.
`H5Fget_file_image`	Provides a simple way to retrieve a copy of the image of an existing, open file. For more information, see section 2.1.6.
`H5LTopen_file_image`	Provides a convenient way to open an initial file image with the Core VFD. For more information, see section 2.2.1.

1.2. Abbreviations

The following abbreviations are used in this document:

Table 1. Abbreviations

Abbreviation	This abbreviation is short for:
FAPL or fapl	File Access Property List. In code samples, fapl is used.
VFD	Virtual File Driver
VFL	Virtual File Layer

1.3. Developer Prerequisites

Developers who use the file image operations described in this document should be proficient and experienced users of the HDF5 C Library APIs. More specifically, developers should have a working knowledge of property lists, callbacks, and virtual file drivers.

1.4. Resources

See the following for more information.

The “RFC: File Image Operations” is the primary source for the information in this document.

The “Alternate File Storage Layouts and Low-level File Drivers” section is in “The HDF5 File” chapter of the *HDF5 User’s Guide* .

The H5P_SET_FAPL_CORE function call can be used to modify the file access property list so that the Memory virtual file driver, H5FD_ CORE, is used. The Memory file driver is also known as the Core file driver.

Links to the Virtual File Layer and List of VFL Functions documents can be found in the HDF5 Technical Notes.

2. C API Call Syntax

The C API function calls described in this chapter fall into two categories: low-level routines that are part of the main HDF5 C Library and one high-level routine that is part of the “lite” API in the high-level wrapper library. The high-level routine uses the low-level routines and presents frequently requested functionality conveniently packaged for application developers’ use.

2.1. Low-level C API Routines

The purpose of this section is to describe the low-level C API routines that support file image operations. These routines allow an in-memory image of an HDF5 file to be opened without requiring file system I/O.

The basic approach to opening an in-memory image of an HDF5 file is to pass the image to the Core file driver, and then tell the Core file driver to open the file. We do this by using the `H5Pget/set_file_image` calls. These calls allow the user to specify an initial file image.

A potential problem with the `H5Pget/set_file_image` calls is the overhead of allocating and copying of large file image buffers. The callback

routines enable application programs to avoid this problem. However, the use of these callbacks is complex and potentially hazardous: the particulars are discussed in the semantics and examples chapters below (see section 3.1 and section 4.1 respectively). Fortunately, use of the file image callbacks should seldom be necessary: the `H5LTopen_file_image` call should address most use cases.

The property list facility in HDF5 is employed in file image operations. This facility 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 extending the property list structure to handle consumable resources cleanly and to avoid constructing a new facility for the purpose.

The sub-sections below describe the low-level C APIs that are used with file image operations.

2.1.1. `H5Pset_file_image`

The `H5Pset_file_image` routine allows an application to provide an image for a file driver to use as the initial contents of the file. This call was 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 the "Virtual File Driver Feature Flags" section for more information. Calling this routine makes a copy of the provided file image buffer. See the "`H5Pset_file_image_callbacks`" section for more information.

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.

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. If a buffer is released, the FAPL's file image `buf_len` will be set to 0 and `buf_ptr` will be set to NULL.

Given the tight interaction between the file image callbacks and the file image, the file image callbacks in a property list cannot be changed while a file image is defined.

With properly constructed file image callbacks, it is possible to avoid actually copying the file image. The particulars of this are discussed in greater detail in the "C API Call Semantics" chapter and in the "Examples" chapter.

2.1.2. `H5Pget_file_image`

The `H5Pget_file_image` routine allows 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 it uses `malloc` and `memcpy` if the callbacks are undefined. When `malloc` and `memcpy` are used, it will be the caller's responsibility to discard the returned buffer via a call to free.

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` contains the ID of the target file access property list.
- `buf_ptr_ptr` contains a NULL or a pointer to a void*. If `buf_ptr_ptr` is not NULL, on successful return, `*buf_ptr_ptr` will contain a pointer to a copy of the initial image provided in the last call to `H5Pset_file_image` for the supplied `fapl_id`. If no initial image has been set, `*buf_ptr_ptr` will be NULL.
- `buf_len_ptr` contains a NULL or a pointer to size_t. If `buf_len_ptr` is not NULL, on successful return, `*buf_len_ptr` will contain the value of the `buf_len` parameter for the initial image in the supplied `fapl_id`. If no initial image is set, the value of `*buf_len_ptr` will be 0.

As with `H5Pset_file_image`, appropriately defined file image callbacks can allow this function to avoid buffer allocation and memory copy operations.

2.1.3. `H5Pset_file_image_callbacks`

The `H5Pset_file_image_callbacks` API call exists to allow an application to control the management of file image buffers through user defined callbacks. These callbacks will be used in the management of file image buffers in property lists and in select file drivers. These routines are invoked when a new file image buffer is allocated, when 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 operations of the `image_malloc`, `image_memcpy`, `image_realloc`, and `image_free` callbacks must be identical to those of the corresponding C standard library calls (`malloc`, `memcpy`, `realloc`, and `free`). While the operations must be identical, the file image callbacks have more parameters. The callbacks and their parameters are described below. The return

values of image_malloc and image_realloc are identical to the return values of malloc and realloc. However, the return values of image_memcpy and image_free are different than the return values of memcpy and free: the return values of image_memcpy and image_free can also indicate failure. See the "File Image Callback Semantics" section for more information.

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;

typedef struct
{
  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);
  herr_t (*image_free)(void *ptr, H5_file_image_op_t file_image_op,
        void *udata);
  void *(*udata_copy)(void *udata);
  herr_t (*udata_free)(void *udata);
  void *udata;
} H5_file_image_callbacks_t;

herr_t H5Pset_file_image_callbacks(hid_t fapl_id,
            H5_file_image_callbacks_t *callbacks_ptr)
```

The parameters of H5Pset_file_image_callbacks are defined as follows:

- fapl_id contains the ID of the target file access property list.
- callbacks_ptr contains a pointer to an instance of the H5_file_image_callbacks_t structure.

The fields of the H5_file_image_callbacks_t structure are defined as follows:

- image_malloc contains 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 contains the size in bytes of the image buffer to allocate.
  - file_image_op contains one of the values of H5_file_image_op_t. These values indicate the operation being performed on the file image when this callback is invoked. Possible values for file_image_op are discussed in Table 2.
  - udata holds the value passed in for the udata parameter to H5Pset_file_image_callbacks.

  Setting image_malloc to NULL indicates that the HDF5 Library should invoke the standard C library malloc() routine when allocating file image buffers.

- image_memcpy contains a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library memcpy() call except that it returns NULL on failure. Recall that the memcpy C Library routine is defined to return the dest parameter in all cases. The parameters of the image_memcpy callback are defined as follows:
  - dest contains the address of the destination buffer.
  - src contains the address of the source buffer.
  - size contains the number of bytes to copy.
  - file_image_op contains one of the values of H5_file_image_op_t. These values indicate the operation being performed on the file image when this callback is invoked. Possible values for file_image_op are discussed in Table 2.
  - udata holds the value passed in for the udata parameter to H5Pset_file_image_callbacks.

  Setting image_memcpy to NULL indicates that the HDF5 Library should invoke the standard C library memcpy() routine when copying buffers.

- image_realloc contains 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 contains the pointer to the buffer being reallocated.
  - size contains the desired size in bytes of the buffer after realloc.
  - file_image_op contains one of the values of H5_file_image_op_t. These values indicate the operation being performed on the file image when this callback is invoked. Possible values for file_image_op are discussed in Table

2.
- udata holds the value passed in for the udata parameter to H5Pset_file_image_callbacks.

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

- image_free contains a pointer to a function with (from the perspective of HDF5) functionality identical to the standard C library free() call except that it will return 0 (SUCCEED) on success and -1 (FAIL) on failure. The parameters of the image_free call back are defined as follows:
  - ptr contains the pointer to the buffer being released.
  - file_image_op contains one of the values of H5_file_image_op_t. These values indicate the operation being performed on the file image when this callback is invoked. Possible values for file_image_op are discussed in Table 2 .
  - udata holds the value passed in for the udata parameter to H5Pset_file_image_callbacks.

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

- udata_copy contains a pointer to a function that (from the perspective of HDF5) allocates a buffer of suitable size, copies the contents of the supplied udata into the new buffer, and returns the address of the new buffer. The function returns NULL on failure. 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, then this parameter should be NULL as well. The parameter of the udata_copy callback is defined as follows:
  - udata contains the pointer to the user data block being copied.

- udata_free contains a pointer to a function that (from the perspective of HDF5) frees 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 should be NULL as well. The parameter of the udata_free callback is defined as follows:
  - udata contains the pointer to the user data block to be freed.

  udata_free returns 0 (SUCCEED) on success and -1 (FAIL) on failure.

- udata contains a pointer value, potentially to user-defined data, that will be passed to the image_malloc, 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 described in the table below:

Table 2. Values for the file_image_op parameter

Value	Comments
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 callbacks when an image buffer is copied during a file open operation. While the image being opened will typically be copied from a FAPL, this need not always be the case. An example of an exception is when the Core file driver takes its initial image from a file.
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 defined in the instance of H5_file_image_callbacks_t 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.

For more information on writing the file image to disk, set the <code>backing_store</code> parameter. See the H5Pset_fapl_core entry

in the *HDF5 Reference Manual*.

2.1.4. 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,
                                   H5_file_image_callbacks_t *callbacks_ptr)
```

The parameters of H5Pget_file_image_callbacks are defined as follows:

- fapl_id contains the ID of the target file access property list.
- callbacks_ptr contains a pointer to an instance of the H5_file_image_callbacks_t structure. All fields should be initialized to NULL. See the "H5Pset_file_image_callbacks" section for more information on the H5_file_image_callbacks_t structure.

Upon successful return, the fields of *callbacks_ptr shall contain values as defined below:

- Upon successful return, callbacks_ptr->image_malloc will contain the pointer passed as the image_malloc field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.
- Upon successful return, callbacks_ptr->image_memcpy will contain the pointer passed as the image_memcpy field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.
- Upon successful return, callbacks_ptr->image_realloc will contain the pointer passed as the image_realloc field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.
- Upon successful return, callbacks_ptr->image_free_ptr will contain the pointer passed as the image_free field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.
- Upon successful return, callbacks_ptr->udata_copy will contain the pointer passed as the udata_copy field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.
- Upon successful return, callbacks_ptr-> udata_free will contain the pointer passed as the udata_free field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.
- Upon successful return, callbacks_ptr->udata will contain the pointer passed as the udata field of the instance of H5_file_image_callbacks_t pointed to by the callbacks_ptr parameter of the last call to H5Pset_file_image_callbacks() for the specified FAPL, or NULL if there has been no such call.

2.1.5. Virtual File Driver Feature Flags

Implementation of the H5Pget/set_file_image_callbacks() and H5Pget/set_file_image() function calls requires a pair of virtual file driver feature flags. The flags are H5FD_FEAT_ALLOW_FILE_IMAGE and H5FD_FEAT_CAN_USE_FILE_IMAGE_CALLBACKS. Both of these are defined in H5FDpublic.h.

The first flag, H5FD_FEAT_ALLOW_FILE_IMAGE, allows a file driver to indicate whether or not it supports file images. A VFD that sets this 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. 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 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). However, such a VFD need not employ the file image after file open time. In such cases, the VFD will not make an in-memory copy of the file image and will not employ the file image callbacks.

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, the second 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 an H5Fopen() call, and if the underlying file driver does not set the H5FD_FEAT_ALLOW_FILE_IMAGE flag, then 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, then 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.

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, 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, setting the flags this way will not cause any problems: the two capabilities are logically distinct.

2.1.6. H5Fget_file_image

The purpose of the H5Fget_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 the SEC2 (aka POSIX), STDIO, and Core (aka Memory) VFDs.

The signature of H5Fget_file_image is defined as follows:

```
ssize_t H5Fget_file_image(hid_t file_id, void *buf_ptr, size_t buf_len)
```

The parameters of H5Fget_file_image are defined as follows:

- file_id contains the ID of the target file.
- buf_ptr contains 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 contains the size of the supplied buffer.

If the return value of H5Fget_file_image is a positive value, then the value will be the length of buffer required to store the file image (in other words, the length of the file). A negative value might be returned if the file is too large to store in the supplied buffer or on failure.

The current file size can be obtained via a call to H5Fget_filesize(). Note that this function returns the value of the end of file (EOF) and not the end of address space (EOA). While these values are frequently the same, it is possible for the EOF to be larger than the EOA. Since H5Fget_file_image() will only obtain a copy of the file from the beginning of the superblock to the EOA, it will be best to use H5Fget_file_image() to determine the size of the buffer required to contain the image.

Other Design Considerations

Here are some other notes regarding the design and implementation of H5Fget_file_image.

The H5Fget_file_image call should be part of the high-level library. However, a file driver agnostic implementation of the routine requires access to data structures that are hidden within the HDF5 Library. We chose to implement the call in the library proper rather than expose those data structures.

There is no reason why the H5Fget_file_image() API call could not work on files opened with any file driver. However, the Family, Multi, and Split file drivers have issues that make the call problematic. At present, files opened with the Family file driver are marked as being created with that file driver in the superblock, and the HDF5 Library refuses to open files so marked with any other file driver. This negates the purpose of the H5Fget_file_image() call. While this mark can be removed from the image, the necessary code is not trivial.

Thus we will not support the Family file driver in H5Fget_file_image() unless there is demand for it. Files created with the Multi and Split file drivers are also marked in the superblock. In addition, they typically use a very sparse address space. A sparse address space would require the use of an impractically large buffer for an image, and most of the buffer would be empty. So, we see no point in supporting the Multi and Split file drivers in H5Fget_file_image() under any foreseeable circumstances.

2.2. High-level C API Routine

The H5LTopen_file_image high-level routine encapsulates the capabilities of routines in the main HDF5 Library with conveniently accessible abstractions.

2.2.1. H5LTopen_file_image

The H5LTopen_file_image routine is designed to provide an easier 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. See the *HDF5 Reference Manual* for more information on high-level APIs.

The signature of H5LTopen_file_image is defined as follows:

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

The parameters of H5LTopen_file_image are defined as follows:

- buf_ptr contains a pointer to the supplied initial image. A NULL value is invalid and will cause H5LTopen_file_image to fail.
- buf_len contains the size of the supplied buffer. A value of 0 is invalid and will cause H5LTopen_file_image to fail.
- flags contains 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 described in the table below:

Table 3. Flags for H5LTopen_file_image

Flag	Comments
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 the file image when finished. The supplied buffer must have been allocated via a call to the standard C library malloc() or calloc() routines. The HDF5 Library will call free() to release the buffer. In the absence of this flag, the HDF5 Library will copy the buffer provided. 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. The HDF5 Library will modify the buffer on write if the image is opened read/write and the H5LT_FILE_IMAGE_DONT_COPY flag is set. The H5LT_FILE_IMAGE_DONT_RELEASE flag, see below, is invalid unless the H5LT_FILE_IMAGE_DONT_COPY flag is set
H5LT_FILE_IMAGE_DONT_RELEASE	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, see above, is set. If the H5LT_FILE_IMAGE_DONT_COPY flag is set and this flag is not set, the HDF5 Library will release the file image buffer after the file is closed using the standard C library free() routine. Using this flag and the 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.

The following table is intended to summarize the semantics of the H5LT_FILE_IMAGE_DONT_COPY and H5LT_FILE_IMAGE_DONT_RELEASE flags (shown as "Don't Copy Flag" and "Don't Release Flag" respectively in the table):

Table 4. Summary of Don't Copy and Don't Release Flag Actions

Don't Copy Flag	Don't Release Flag	Make Copy of User Supplied Buffer	Pass User Supplied Buffer to File Driver	Release User Supplied Buffer When Done	Permit realloc of Buffer Used by File Driver
False	Don't care	True	False	False	True
True	False	False	True	True	True
True	True	False	True	False	False

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.

Note that there is no way currently to specify a "backing store" file name in this definition of H5LTopen_image.

3. C API Call Semantics

The purpose of this chapter is to describe some issues that developers should consider when using file image buffers, property lists, and callback APIs.

3.1. File Image Callback Semantics

The H5Fget/set_file_image_callbacks() API calls allow an 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 potentially in any in-memory file driver developed in the future.

From the perspective of the HDF5 Library, the supplied image_malloc(), 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.

At the beginning of this document, we talked about the need to work within the constraints of the property list mechanism. When we said “from the perspective of the HDF5 Library…” in the paragraph above, we are making reference to this point.

The property list mechanism was developed as a way to add parameters to functions without changing the parameter list and breaking existing code. However, it was designed to use only “call by value” semantics, not “call by reference”. The decision to use “call by value” semantics requires that the values of supplied variables be copied into the property list. This has the advantage of simplifying the copying and deletion of property lists. However, if the value to be copied is large (say a 2 GB file image), the overhead can be unacceptable.

The usual solution to this problem is to use “call by reference” where only a pointer to an object is placed in a parameter list rather than a copy of the object itself. However, use of “call by reference” semantics would greatly complicate the property list mechanism: at a minimum, it would be necessary to maintain reference counts to dynamically allocated objects so that the owner of the object would know when it was safe to free the object.

After much discussion, we decided that the file image operations calls were sufficiently specialized that it made no sense to rework the property list mechanism to support “call by reference.” Instead we provided the file image callback mechanism to allow the user to implement some version of “call by reference” when needed. It should be noted that we expect this mechanism to be used rarely if at all. For small file images, the copying overhead should be negligible, and for large images, most use cases should be addressed by the `H5LTopen_file_image` call.

In the (hopefully) rare event that use of the file image callbacks is necessary, the fundamental point to remember is that the callbacks must be constructed and used in such a way as to maintain the library’s illusion that it is using “call by value” semantics.

Thus the property list mechanism must think that it is allocating a new buffer and copying the supplied buffer into it when the file image property is set. Similarly, it must think that it is allocating a new buffer and copying the contents of the existing buffer into it when it copies a property list that contains a file image. Likewise, it must think it is de-allocating a buffer when it discards a property list that contains a file image.

Similar illusions must be maintained when a file image buffer is copied into the Core file driver (or any future driver that uses the file image callbacks) when the file driver re-sizes the buffer containing the image and finally when the driver discards the buffer.

3.1.1. Buffer Ownership

The owner of a file image in a buffer is the party that has the responsibility to discard the file image buffer when it is no longer needed. In this context, the owner is either the HDF5 Library or the application program.

We implemented the `image_*` callback facility to allow efficient management of large file images. These facilities can be used to allow sharing of file image buffers between the application and the HDF5 library, and also transfer of ownership in either direction. In such operations, care must be taken to ensure that ownership is clear and that file image buffers are not discarded before all references to them are discarded by the non-owning party.

Ownership of a file image buffer will only be passed to the application program if the file image callbacks are designed to do this. In such cases, the application program must refrain from freeing the buffer until the library has deleted all references to it. This in turn will happen after all property lists (if any) that refer to the buffer have been discarded, and the file driver (if any) that used the buffer has closed the file and thinks it has discarded the buffer.

3.1.2. Sharing a File image Buffer with the HDF5 Library

As mentioned above, 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. They were designed based on the assumption that all new parameters would be “call by value” and not “call by reference.” Having “call by value” parameters means property lists can be copied, reused, and discarded with ease.

Suppose an application wished to share a file image buffer with the HDF5 Library. This means the library would be allowed to read the file image, but not free it. The file image callbacks might be constructed as follows to share a buffer:

- Construct the `image_malloc()` call so that it returns the address of the buffer instead of allocating new space. This will keep the library thinking that the buffers are distinct even when they are not. 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 the reference counter on each call to `image_malloc()`.
- Construct the `image_memcpy()` call so that it does nothing. As a sanity check, make it fail if the source and destination pointers do not match the buffer address in the user data or if the size is unexpected.
- Construct the `image_free()` routine so that it 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. The behavior of `image_realloc()` in this scenario depends on what the application wants to do with the file image after it has been opened. We discuss this issue in the next section. Note also that the operation passed into the file image callbacks allow the callbacks to behave differently depending on the context in which they are used.

For more information on user defined data, see the "H5Pset_file_image_callbacks" section.

3.1.3. File Driver Considerations

When a file 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 this buffer. As processing progresses, the driver will reallocate the image as necessary to increase its size and will eventually discard the image at file close. If defined, the driver will use the file image callbacks for these operations; otherwise, the driver will use the standard C library calls. See the "H5Pset_file_image_callbacks" section for more information.

As described above, 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. There are two possible complications involving the file driver. The complications are the possibility of reallocation calls from the driver and the possibility of the continued existence of property lists containing references to the buffer.

Suppose an application wishes to share a file image buffer with the HDF5 Library. The application allows the library to read (and possibly write) the image, but not free it. We must first decide whether the image is to be opened read-only or read/write.

If the image will 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 constructed as described in the section above.

Suppose, however, that the file image will be opened read/write and may grow during the computation. We must now allow for the base address of the buffer to change due to reallocation calls, and we must employ the user data structure to communicate any change in the buffer base address and size to the application. We pass buffer changes to the application so that the application will be able to eventually free the buffer. To this end, we might define a user data structure as shown in the example below:

```
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;
}
```

Example 1. Using a user data structure to communicate with an application

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:

- Construct the image_malloc() call so that it returns the value in the init_ptr field of the user data structure 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 structure or if any of the modified fields have values other than their initial values.
- 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 do not match the init_ptr and init_size fields as appropriate.
- 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, or mod_ref_count fields of the user data structure still have their initial values, verify that the supplied pointer matches the init_ptr field and that the supplied size does not match the init_size field. Decrement init_ref_count, set mod_ptr equal to the address returned by realloc, set mod_size equal to the supplied size, and set mod_ref_count to 1.
    - If the mod_ptr, mod_size, or mod_ref_count fields of the user data structure are defined, verify that the supplied pointer matches the value of mod_ptr and that the supplied size does not match 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.

- 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 structure. Flag an error if init_ref_count drops below zero.
    - If the H5_FILE_IMAGE_OP_FILE_CLOSE flag is set, check to see if the mod_ptr, mod_size, or mod_ref_count fields of the user data structure 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 have not 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 or mod_size fields have been modified, pass these values on to the application as well.

3.2. Initial File Image Semantics

One can argue whether creating a file with an initial file image is closer to creating a file or opening a file. 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, opening a file with an initial file image is a bit of both creating a file and opening a file. Conceptually, we will create a file on disk, write the supplied image to the file, close the file, open the file as an HDF5 file, and then proceed as usual (of course, the Core VFD will not write to the file system unless it is configured to do so). This process is similar to a file create: we are creating a file that did not exist on disk to begin with and writing data to it. Also, we must verify that no file of the supplied name is open. However, this process is also similar to a file open: 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 when an initial file image is supplied:

- `H5Fopen()` must fail if the target file driver does not set the `H5FD_FEAT_ALLOW_FILE_IMAGE` flag and a file image is specified in the FAPL.
- If the target file driver supports the `H5FD_FEAT_ALLOW_FILE_IMAGE` flag, then `H5Fopen()` must fail if the file is already open or if a file of the specified name exists.
- Even if the above constraints are satisfied, `H5Fopen()` must still fail if the image does not 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.

See the "Virtual File Driver Feature Flags" section for more information.

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

While the above section on the semantics of the file image callbacks may seem rather gloomy, we get the payback here. The above says everything that needs to be said about initial file image semantics in general. The sub-section below has a few more observations on the Core file driver.

3.2.1. Applying Initial File Image Semantics to the Core File Driver

At present, the Core file driver uses the <code>open()</code> and <code>read()</code> system calls to load an HDF5 file image from the file system into RAM. Further, if the <code>backing_store</code> flag is set in the FAPL entry specifying the use of the Core file driver, the Core file driver's internal image will be used to overwrite the source file on either flush or close. See the `H5Pset_fapl_core` entry in the *HDF5 Reference Manual* for more information.

This results in the following observations. In all cases assume that use of the Core file driver has been specified in the FAPL.

- If the file specified in the `H5Fopen()` call does not exist, and no initial image is specified in the FAPL, the open must fail because there is no source for the initial image needed by the Core file driver.
- If the file specified in the `H5Fopen()` call does exist, and an initial image is specified in the FAPL, the open must fail because the source of the needed initial image is ambiguous: the file image could be taken either from file or from the FAPL.
- If the file specified in the `H5Fopen()` call does not exist, and an initial image is specified in the FAPL, the open will succeed. This assumes that the supplied image is valid. Further, if the backing store flag is set, the file specified in the `H5Fopen()` call will be created, and the contents of the Core file driver's internal buffer will be written to the new file on flush or close.

Thus a call to `H5Fopen()` can result in the creation of a new HDF5 file in the file system.

4. Examples

The purpose of this chapter is to provide examples of how to read or build an in-memory HDF5 file image.

4.1. Reading an In-memory HDF5 File Image

The `H5Pset_file_image()` function call allows 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>
```

Example 2. Using H5Pset_file_image to initialize the Core file driver

This solution is easy to code, but the supplied buffer is duplicated twice. The first time is in the call to H5Pset_file_image() when the image is duplicated and the duplicate inserted into the property list. The second time is when the file is opened: 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 this could become a significant performance hit for large images.

If we want to avoid the extra malloc and memcpycalls, 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. In this example, the application retains ownership of the buffer and avoids 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>
```

Example 3. Using H5LTopen_file_image to open a read-only file image where the application retains ownership of the buffer

If the application wants to transfer ownership of the buffer to the HDF5 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>
```

Example 4. Using H5LTopen_file_image to open a read-only file image where the application transfers ownership of the buffer

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 section below.

4.2. In-memory HDF5 File Image Construction

Before the implementation of file image operations, HDF5 supported construction of an image of an HDF5 file in memory with the Core file driver. The H5Fget_file_image() function call allows an application access to the file image without first writing it to disk. See the following code fragment:

```
<Open and construct the desired file with the Core file driver>

H5Fflush(fid);
size = H5Fget_file_image(fid, NULL, 0);
buffer_ptr = malloc(size);
H5Fget_file_image(fid, buffer_ptr, size);
```

Example 5. Accessing the image of a file in memory

The use of H5Fget_file_image() may be acceptable for small images. For large images, the cost of the malloc() and memcpy() operation s may be excessive. To address this issue, the H5Pset_file_image_callbacks() call allows an 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(NULL); /* always fails */
}

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));
}

herr_t 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(0); /* if we get here, we must have been successful */
}

void *udata_copy(void *udata)
{
return(udata);
}

herr_t udata_free(void *udata)

{
return(0);
}

H5_file_image_callbacks_t callbacks = {image_malloc, image_memcpy,
image_realloc, image_free,
udata_copy, udata_free,
(void *)(&udata)};

<allocate fapl_id>

H5Pset_file_image_callbacks(fapl_id, &callbacks);

<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()>
```

Example 6. Using `H5Pset_file_image_callbacks` to improve memory allocation

The above code fragment gives the application full ownership of the buffer used by the Core file driver after the file is closed, and it notifies the application that the HDF5 Library is done with the buffer by setting `udata.image_ptr` to something other than NULL. If read access to the buffer is sufficient, the `H5Fget_vfd_handle()` call can be used as an alternate solution to get access to the base address of the Core file driver's buffer.

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 if we can put a strong and tight upper bound on the size of the necessary buffer, then the following pseudo code demonstrates a method of avoiding memory allocation completely. The downside, however, is that buffer is allocated 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(NULL); /* always fails */
}

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);
}

herr_t 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(0); /* if we get here, we must have been successful */
}

void *udata_copy(void *udata)
{
return(udata);
}

herr_t udata_free(void *udata)
{
return(0);
}

H5_file_image_callbacks_t callbacks = {image_malloc, image_memcpy,
image_realloc, image_free,
udata_copy, udata_free,
(void *)(&udata)};

/* end of initialization */

<allocate fapl_id>

H5Pset_file_image_callbacks(fapl_id, &callbacks);

<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>
```

Example 7. Using `H5Pset_file_image_callbacks` with a static buffer

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 the code 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 `H5P get_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); /* if we get here, we must have been successful */
}

void *image_realloc(void *ptr, size_t size, H5_file_image_op_t file_image_op, void *udata)
{
/* One would think that this function is not needed in this scenario, as
*  only the contents of the HDF5 file is being changed, not its size or
*  structure. However, the Core file driver calls realloc() just before
*  close to clip the buffer to the size indicated by the end of the
*  address space.
*
*  While this call must be supported in this case, the size of
*  the image should never change. Hence the function can limit itself
*  to performing sanity checks, and returning the base address of the
*  statically allocated buffer.
*/
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);
assert(((struct udata_t *)udata)->image_size == size);
return((((struct udata_t *)udata)->image_ptr);
}

herr_t 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(0); /* if we get here, we must have been successful */
}

void *udata_copy(void *udata)
{

return(udata);
}

herr_t udata_free(void *udata)
{
return(0);
}
```

```
H5_file_image_callbacks_t callbacks = {image_malloc, image_memcpy,
image_realloc, image_free,
udata_copy, udata_free,
(void *)(&udata)};
/* end of initialization */

<allocate fapl_id>

H5Pset_file_image_callbacks(fapl_id, &callbacks);

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>
```

Example 8. Using `H5Pset_file_image_callbacks` where only the datasets change

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. See the example below:

```
<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>
```

Example 9. Using `H5LTopen_file_image` where only the datasets change

The above pseudo code allows updates of a file image about as cheaply as possible. We assume the application has enough RAM for the image and that the HDF5 file structure is constant after the first write.

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

Since we must use `malloc`, `realloc`, and `free` in this example, and since `realloc` can change the base address of a buffer, we must maintain two of `ptr`, `size`, and `ref_count` triples in the `udata` structure. The first triple is for the property list (which will never change the buffer), and the second triple is for the file driver. As shall be seen, this complicates the file image callbacks considerably. Note also that while we do not use `H5Pget_file_image()` in this example, we do 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;
nt 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); /* if we get here, we must have been successful */
}
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);
}

herr_t 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(0); /* if we get here, we must have been successful */
}

void *udata_copy(void *udata)
{
return(udata);
}

herr_t udata_free(void *udata)
{
return(0);
}

H5_file_image_callbacks_t callbacks = {image_malloc, image_memcpy,
image_realloc, image_free,
udata_copy, udata_free,
(void *)(&udata)};

/* end of initialization */

<allocate fapl_id>

H5Pset_file_image_callbacks(fapl_id, &callbacks);

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>
```

Example 10. Using `H5LTopen_file_image` where only the datasets change and where the file structure and image size might not be constant

The above pseudo code shows how a buffer can be passed back and forth between the application and the HDF5 Library. The code also shows the application having control of the actual allocation, reallocation, and freeing of the buffer.

4.3. Using HDF5 to Construct and Read a Data Packet

Using the file image operations described in this document, we can bundle up data in an image of an HDF5 file on one process, transmit the image to a second process, and 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 in the example below using as simple a set of calls as possible. The set of calls in the example has extra buffer allocations. To reduce extra buffer allocations, see the sections above.

In the following example, we construct an HDF5 file image on process A and then 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: all the processing is done in memory with the Core file driver.

```
*** Process A ***

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

H5Fflush(fid);
size = H5Fget_file_image(fid, NULL, 0);
buffer_ptr = malloc(size);
H5Fget_file_image(fid, buffer_ptr, size);

<transmit size>

<transmit *buffer_ptr>
free(buffer_ptr);
<close core file>
```

```
*** Process B ***

hid_t file_id;




<receive size>
buffer_ptr = malloc(size)
<receive image in *buffer_ptr>

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>
```

Example 11. Building and passing a file image from one process to another

4.4. Using a Template File

After the above examples, an example of the use of a template file might seem anti-climactic. A template file might be used to enforce consistency on file structure between files or in parallel HDF5 to avoid long sequences of collective operations to create the desired groups, datatypes, and possibly datasets. The following pseudo code outlines a potential 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>
```

Example 12. Using a template file

Observe that the above pseudo code includes an unnecessary 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. This is 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 an 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.

In contrast, the argument is stronger in the parallel case since group, datatype, and dataset creations are all expensive collective operations. The argument is also weaker: 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. Parallel HDF5 requires eager space allocation for chunked datasets. 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, this approach would require extending our current API to allow compressed images. We would also have to add the 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.

5. Java Signatures for File Image Operations API Calls

Potential Java function call signatures for the file image operation APIs are described in this section. These have not yet been implemented, and there are no immediate plans for implementation.

Note that the `H5LTopen_file_image()` call is omitted. Our practice has been to not support high-level library calls in Java.

H5Pset_file_image

```
int H5Pset_file_image(int fapl_id, const byte[] buf_ptr);
```

H5Pget_file_image

```
herr_t H5Pget_file_image(hid_t fapl_id, byte[] buf_ptr_ptr);
```

H5_file_image_op_t

```
public static H5_file_image_op_t
{
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_malloc_cb

```
public interface H5_file_image_malloc_cb extends Callbacks {
buf[] callback(H5_file_image_op_t file_image_op, CBuserdata udata);
}
```

H5_file_image_memcpy_cb

```
public interface H5_file_image_memcpy_cb extends Callbacks {
buf[] callback(buf[] dest, const buf[] src, H5_file_image_op_t file_image_op, CBuserdata
udata);
}
```

H5_file_image_realloc_cb

```
public interface H5_file_image_realloc_cb extends Callbacks {
buf[] callback(buf[] ptr, H5_file_image_op_t file_image_op, CBuserdata udata);
}
```

H5_file_image_free_cb

```
public interface H5_file_image_free_cb extends Callbacks {
void callback(buf[] ptr, H5_file_image_op_t file_image_op, CBuserdata udata);
}
```

H5_file_udata_copy_cb

```
public interface H5_file_udata_copy_cb extends Callbacks {
buf[] callback(CBuserdata udata);
}
```

H5_file_udata_free_cb

```
public interface H5_file_udata_free_cb extends Callbacks {
void callback(CBuserdata udata);
}
```

H5_file_image_callbacks_t

```
public abstract class H5_file_image_callbacks_t
{
H5_file_image_malloc_cb image_malloc;
H5_file_image_memcpy_cb image_memcpy;
H5_file_image_realloc_cb image_realloc;
H5_file_image_free_cb image_free;
H5_file_udata_copy_cb udata_copy;
H5_file_udata_free_cb udata_free;
CBuserdata udata;
```

```
public H5_file_image_callbacks_t(
H5_file_image_malloc_cb image_malloc,
H5_file_image_memcpy_cb image_memcpy,
H5_file_image_realloc_cb image_realloc,
H5_file_image_free_cb image_free,
H5_file_udata_copy_cb udata_copy,
H5_file_udata_free_cb udata_free,
CBuserdata udata) {
this.image_malloc = image_malloc;
this.image_memcpy = image_memcpy;
this.image_realloc = image_realloc;
this.image_free = image_free;
this.udata_copy = udata_copy;
this.udata_free = udata_free;
this.udata = udata;
}
}
```

H5Pset_file_image_callbacks

```
int H5Pset_file_image_callbacks(int fapl_id,
H5_file_image_callbacks_t callbacks_ptr);
```

H5Pget_file_image_callbacks

```
int H5Pget_file_image_callbacks(int fapl_id,
H5_file_image_callbacks_t[] callbacks_ptr);
```

H5Fget_file_image

```
long H5Fget_file_image(int file_id, byte[] buf_ptr);
```

6. Fortran Signatures for File Image Operations API Calls

Potential Fortran function call signatures for the file image operation APIs are described in this section. These have not yet been implemented, and there are no immediate plans for implementation.

6.1. Low-level Fortran API Routines

The Fortran low-level APIs make use of Fortran 2003's ISO_C_BINDING module in order to achieve portable and standard conforming interoperability with the C APIs. The C pointer (`C_PTR`) and function pointer (`C_FUN_PTR`) types are returned from the intrinsic procedures `C_LOC(X)` and `C_FUNLOC(X)`, respectively, defined in the ISO_C_BINDING module. The argument X is the data or function to which the C pointers point to and must have the TARGET attribute in the calling program. Note that the variable name lengths of the Fortran equivalent of the predefined C constants were shortened to less than 31 characters in order to be Fortran standard compliant.

6.1.1. H5Pset_file_image_f

The signature of `H5Pset_file_image_f` is defined as follows:

```
SUBROUTINE H5Pset_file_image_f(fapl_id, buf_ptr, buf_len, hdferr)
```

The parameters of `H5Pset_file_image` are defined as follows:

`INTEGER(hid_t), INTENT(IN):: fapl_id`	Will contain the ID of the target file access property list.
`TYPE(C_PTR), INTENT(IN):: buf_ptr`	Will supply the C pointer to the initial file image or `C_NULL_PTR` if no initial file image is desired.
`INTEGER(size_t), INTENT(IN):: buf_len`	Will contain the size of the supplied buffer or 0 if no initial image is desired.
`INTEGER, INTENT(OUT) :: hdferr`	Will return the error status: 0 for success and -1 for failure.

6.1.2. H5Pget_file_image_f

The signature of H5Pget_file_image_f is defined as follows:

```
SUBROUTINE H5Pget_file_image_f(fapl_id, buf_ptr, buf_len, hdferr)
```

The parameters of H5Pget_file_image_f are defined as follows:

INTEGER(hid_t), INTENT(IN) :: fapl_id	Will contain the ID of the target file access property list
TYPE(C_PTR), INTENT(INOUT), VALUE :: buf_ptr	Will hold either a C_NULL_PTR or a scalar of type c_ptr. If buf_ptr is not C_NULL_PTR, on successful return, buf_ptr shall contain a C pointer to a copy of the initial image provided in the last call to H5Pset_file_image_f for the supplied fapl_id, or buf_ptr shall contain a C_NULL_PTR if there is no initial image set. The Fortran pointer can be obtained using the intrinsic C_F_POINTER.
INTEGER(size_t), INTENT(OUT) :: buf_len	Will contain the value of the buffer parameter for the initial image in the supplied fapl_id. The value will be 0 if no initial image is set.
INTEGER, INTENT(OUT) :: hdferr	Will return the error status: 0 for success and -1 for failure.

6.1.3. H5Pset_file_image_callbacks_f

The signature of H5Pset_file_image_callbacks_f is defined as follows:

```
INTEGER :: H5_IMAGE_OP_PROPERTY_LIST_SET_F=0,
H5_IMAGE_OP_PROPERTY_LIST_COPY_F=1,
H5_IMAGE_OP_PROPERTY_LIST_GET_F=2,
H5_IMAGE_OP_PROPERTY_LIST_CLOSE_F=3,
H5_IMAGE_OP_FILE_OPEN_F=4,
H5_IMAGE_OP_FILE_RESIZE_F=5,
H5_IMAGE_OP_FILE_CLOSE_F=6

TYPE, BIND(C) :: H5_file_image_callbacks_t
TYPE(C_FUN_PTR), VALUE :: image_malloc
TYPE(C_FUN_PTR), VALUE :: image_memcpy
TYPE(C_FUN_PTR), VALUE :: image_realloc
TYPE(C_FUN_PTR), VALUE :: image_free
TYPE(C_FUN_PTR), VALUE :: udata
TYPE(C_FUN_PTR), VALUE :: udata_copy
TYPE(C_FUN_PTR), VALUE :: udata_free
TYPE(C_PTR), VALUE :: udata
END TYPE H5_file_image_callbacks_t
```

The semantics of the above values will be the same as those defined in the C enum. See Section 2.1.3 for more information.

Fortran Callback APIs
The Fortran callback APIs are shown below.

```
FUNCTION op_func(size, file_image_op, udata,) RESULT(image_malloc)
```

INTEGER(size_t) :: size	Will contain the size of the image buffer to allocate in bytes.
INTEGER :: file_image_op	Will be set to one of the values of H5_IMAGE_OP_* indicating the operation being performed on the file image when this callback is invoked.
TYPE(C_PTR), VALUE :: udata	Will be set to the value passed in for the udata parameter to H5Pset_file_image_callbacks_f.

TYPE(C_FUN_PTR), VALUE :: image_malloc	Shall contain a pointer to a function with functionality identical to the standard C library memcpy() call.

FUNCTION op_func(dest, src, size, & file_image_op, udata) RESULT(image_memcpy)

TYPE(C_PTR), VALUE :: dest	Will contain the address of the buffer into which to copy.
TYPE(C_PTR), VALUE :: src	Will contain the address of the buffer from which to copy
INTEGER(size_t) :: size	Will contain the number of bytes to copy.
INTEGER :: file_image_op	Will be set to one of the values of H5_IMAGE_OP_* indicating the operation being performed on the file image when this callback is invoked.
TYPE(C_PTR), VALUE :: udata	Will be set to the value passed in for the udata parameter to H5Pset_file_image_callbacks_f.
TYPE(C_FUN_PTR), VALUE :: image_memcpy	Shall contain a pointer to a function with functionality identical to the standard C library memcpy() call.

FUNCTION op_func(ptr, size, & file_image_op, udata) RESULT(image_realloc)

TYPE(C_PTR), VALUE :: ptr	Will contain the pointer to the buffer being reallocated
INTEGER(size_t) :: size	Will contain the desired size of the buffer after realloc in bytes.
INTEGER :: file_image_op	Will be set to one of the values of H5_IMAGE_OP_* indicating the operation being performed on the file image when this callback is invoked.
TYPE(C_PTR), VALUE :: udata	Will be set to the value passed in for the udata parameter to H5Pset_file_image_callbacks_f.
TYPE(C_FUN_PTR), VALUE :: image_realloc	Shall contain a pointer to a unction functionality identical to the standard C library realloc() call.

FUNCTION op_func(ptr, file_image_op, udata) RESULT(image_free)

TYPE(C_PTR), VALUE :: ptr	Will contain the pointer to the buffer being released.
INTEGER :: file_image_op	Will be set to one of the values of H5_IMAGE_OP_* indicating the operation being performed on the file image when this callback is invoked.
TYPE(C_PTR), VALUE :: udata	Will be set to the value passed in for the udata parameter to H5Pset_file_image_callbacks_f.
TYPE(C_PTR), VALUE :: image_free	Shall contain a pointer to a function with functionality identical to the standard C library free() call

FUNCTION op_func(udata) RESULT(udata_copy)

TYPE(C_PTR), VALUE :: udata	Will be set to the value passed in for the udata parameter to H5Pset_file_image_callbacks_f.
TYPE(C_FUN_PTR), VALUE :: udata_copy	Shall contain a pointer to a function that will allocate a buffer of suitable size, copy the contents of the supplied udata into the new buffer, and return the address of the new buffer. The function will return C_NULL_PTR on failure.

FUNCTION op_func(udata) RESULT(udata_free)

TYPE(C_PTR), VALUE :: udata	Shall contain a pointer value, potentially to user-defined data, that will be passed to the image_malloc, image_memcpy, image_realloc, and image_free callbacks.

The signature of H5Pset_file_image_callbacks_f is defined as follows:

SUBROUTINE H5Pset_file_image_callbacks_f(fapl_id, &callbacks_ptr, hdferr)

The parameters are defined as follows:

INTEGER(hid_t), INTENT(IN) :: fapl_id	Will contain the ID of the target file access property list.
TYPE(H5_file_image_callbacks_t), INTENT(IN) :: callbacks_ptr	Will contain the callback derived type. callbacks_ptr shall contain a pointer to the Fortran function via the intrinsic functions C_LOC(X) and C_FUNLOC(X).
INTEGER, INTENT(OUT) :: hdferr	Will return the error status: 0 for success and -1 for failure.

6.1.4. H5Pget_file_image_callbacks_f

The H5Pget_file_image_callbacks_f routine is designed to obtain the current file image callbacks from a file access property list.

The signature is defined as follows

SUBROUTINE H5Pget_file_image_callbacks_f(fapl_id, callbacks_ptr, hdferr)

The parameters are defined as follows:

INTEGER(hid_t), INTENT(IN) :: fapl_id	Will contain the ID of the target file access property list.
TYPE(H5_file_image_callbacks_t), INTENT(OUT) :: callbacks_ptr	Will contain the callback derived type. Each member of the derived type shall have the same meaning as its C counterpart. See section 2.1.4 for more information.
INTEGER, INTENT(OUT) :: hdferr	Will return the error status: 0 for success and -1 for failure.

6.1.5. Fortran Virtual File Driver Feature Flags

Implementation of the H5Pget/set_file_image_callbacks_f() and H5Pget/set_file_image_f() APIs requires a pair of new virtual file driver feature flags:

H5FD_FEAT_LET_IMAGE_F
H5FD_FEAT_LET_IMAGE_CALLBACK_F

See the "Virtual File Driver Feature Flags" section for more information.

6.1.6. H5Fget_file_image_f

The signature of H5Fget_file_image_f shall be defined as follows:

```
SUBROUTINE H5Fget_file_image_f(file_id, buf_ptr, buf_len, hdferr, buf_size)
```

The parameters of H5Fget_file_image_f are defined as follows:

INTEGER(hid_t), INTENT(IN) :: file_id	Will contain the ID of the target file.
TYPE(C_PTR), INTENT(IN) :: buf_ptr	Will contain a C pointer to the buffer into which the image of the HDF5 file is to be copied. If buf_ptr is C_NULL_PTR, no data will be copied.
INTEGER(size_t), INTENT(IN) :: buf_len	Will contain the size in bytes of the supplied buffer.
INTEGER(ssizet_t), INTENT(OUT), OPTIONAL :: buf_size	Will indicate the buffer size required to store the file image (in other words, the length of the file). If only the buf_size is needed, then buf_ptr should be also be set to C_NULL_PTR
INTEGER, INTENT(OUT) :: hdferr	Returns the error status: 0 for success and -1 for failure.

See the "H5Fget_file_image" section for more information.

6.2. High-level Fortran API Routine

The new Fortran high-level routine H5LTopen_file_image_f will provide a wrapper for the high-level H5LTopen_file_image function. Consequently, the high-level Fortran API will not be implemented using low-level HDF5 Fortran APIs.

6.2.1. H5LTopen_file_image_f

The signature of H5LTopen_file_image_f is defined as follows:

```
SUBROUTINE H5LTopen_file_image_f(buf_ptr, buf_len, flags, file_id, hdferr)
```

The parameters of H5LTopen_file_image_f are defined as follows:

TYPE(C_PTR), INTENT(IN), VALUE :: buf_ptr	Will contain a pointer to the supplied initial image. A C_NULL_PTR value is invalid and will cause H5LTopen_file_image_f to fail.
INTEGER(size_t), INTENT(IN) :: buf_len	Will contain the size of the supplied buffer. A value of 0 is invalid and will cause H5LTopen_file_image_f to fail.
INTEGER, INTENT(IN) :: flags	Will 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_IMAGE_OPEN_RW_F, H5LT_IMAGE_DONT_COPY_F, and H5LT_IMAGE_DONT_RELEASE_F. The C equivalent flags are defined in the "H5LTopen_file_image" section.

`INTEGER(hid_t), INTENT(IN) :: file_id`	Will be a file ID on success.
`INTEGER, INTENT(OUT) :: hdferr`	Returns the error status: 0 for success and -1 for failure.