RFC: HDF5 File Space Allocation and Aggregation

John Mainzer

The current HDF5 file space allocation and aggregation code places small raw data and metadata in aggregation blocks place more or less randomly through out the file. Further, small raw data and metadata aggregation block size is not recorded in the super block. To support page caching (discussed in the companion page caching RFC), both of these deficits must be repaired.

This RFC explores the current state of the free list manager and metadata/raw data aggregator code, and proposes API, file format, and code changes to address the above issues.

# Introduction

Efficient access to small pieces of metadata and raw data within an HDF5 file requires that the actual I/O operations accessing that data be performed in much larger, and ideally, well-aligned, “pages”, which can be cached by the HDF5 library. Implementing paged access for small pieces of raw data and metadata requires that such pieces be allocated in a way that aggregates them into constant-sized pages within the HDF5 file. For such pages to be read and written efficiently, they must also be aligned to the block size of the underlying file system (and ideally have size some multiple of the file system block size).

Any such system for aggregating small raw data and metadata allocations can’t help but affect the more general issue of file space allocation, de-allocation and re-use.

At present, while the HDF5 library supports both small raw data and metadata aggregation, the aggregation blocks are not aligned, and may actually vary widely from the specified aggregation block size. Also, the aggregator block size is not stored in the file, and thus must be set every time the file is opened – allowing even greater variation.

The objective of this RFC is to propose changes providing the prerequisites for small raw data and metadata page caching (but not address page caching proper, which is addressed in the companion Page Caching RFC). Thus the following issues must be addressed:

1. Design of an API for specifying small raw data and metadata aggregator page alignment and page size at file creation time.
2. Design of new superblock message for storing aggregator page alignment and size. We must also address the issue of how to deal with a file that has been opened (and modified) by a version of the HDF5 library that doesn’t understand the new superblock message.
3. Outline modifications to the library’s aggregators and the free list manager needed to implement aligned small raw data and metadata aggregation. The RFC must also address the follow-on effects of such modifications on file space allocation, de-allocation, and re-use in general.
4. Discuss the incompatibilities between Single Writer/Multiple Reader (SWMR) metadata writes and page caching. Outline possible solutions to the problem, and discuss their advantages and drawbacks.
5. Discuss the incompatibilities between metadata journaling and page caching. Outline possible solutions to the problem, and discuss their advantages and drawbacks.

# File Space Allocation and Aggregation – the Current State of Play

To make sensible recommendations for implementing small raw data and metadata aggregation in HDF5, one must first have a good understanding of the current small raw data and metadata aggregators, the free list, and HDF5 file space allocation in general.

Unfortunately, I am not aware of any developer level documentation of these subsystems, and thus I have found it necessary to review the HDF5 library source code and document the sections relevant to this RFC.

This has proved a painful exercise. While I hope I have done it well enough for the current design purposes, the implementer will have to take it further still – and will likely be bitten once or twice in the process.

This section contains discussions of the small raw data and metadata aggregators in the absence of the free list managers, in the presence of the free list managers, and of the free list managers proper.

Each of these discussions starts with a conceptual overview of the module in questions, followed by a detailed examination of the existing code. The detailed discussions are included in the hope that the implementer will find them useful. Other readers may skip them with impunity.

The developer is advised that the “detailed discussions” skip over many issues though irrelevant to the design issues at hand. While I may hope that most of these judgment calls will prove correct, some errors should be expected.

At the level of interest to this RFC, space in an HDF5 file is allocated via calls to H5MF\_alloc()[[1]](#footnote-2) and de-allocated via calls to H5MF\_xfree()[[2]](#footnote-3). This may be a bit of an oversimplification, as other routines may be used for the purposes (H5MF\_try\_extend()[[3]](#footnote-4) and possibly H5MF\_alloc\_tmp()[[4]](#footnote-5) for instance). However, this selection of routines should be sufficient for design purposes.

As we are interested in small raw data and metadata aggregation and the interaction between the aggregators and the free space manager, we can also presume that the file space strategy[[5]](#footnote-6) must be one of:

H5F\_FILE\_SPACE\_ALL\_PERSIST 🡺 Persistent free space managers, aggregators,

virtual file driver.

H5F\_FILE\_SPACE\_ALL 🡺 Non-persistent free space managers, aggregators,

virtual file driver.

This is the library default

H5F\_FILE\_SPACE\_AGGR\_VFD 🡺 Aggregators, Virtual file driver.

In other words, we may assume that the aggregators will be enabled, that the free space manager may or may not be enabled, and if it is, it may or may not be persistent.[[6]](#footnote-7)

## Small Raw Data and Metadata Aggregation in the Absence of the Free List Manager

We start our outline with the simplest case – small raw data and metadata aggregation without a free list.

Implementing the changes proposed in this RFC will require an intimate understanding of the existing aggregator code. While this is discussed in considerable detail, we start with a conceptual overview that should make the subsequent detailed discussion easier to follow.

### Conceptual Overview of Raw Data and Metadata Aggregation in the Absence of the Free List Manager

At present, HDF5 maintains two aggregators for each file – one for metadata, and the other for small pieces of raw data. The two aggregators are identical in their operation and supporting data structures.

From reading the existing aggregator code, I gather that the basic objective is to cluster small raw data and metadata in the HDF5 file in chunks of roughly alloc\_size[[7]](#footnote-8) bytes or greater – with no mixing of raw data and metadata in any one chunk. Please note the use of the term “roughly”. As shall be seen, the lower bound on the size of a cluster of small raw data or metadata produced by the aggregator code is the minimum size (respectively) of a piece of raw data or metadata in HDF5, and the upper bound is limited only by the order of space allocations and (again respectively) the maximum size of a pieces raw data or metadata in HDF5.

The behavior of the aggregators is somewhat involved, and is discussed in detail later. However, the essence of the algorithm employed is outlined below[[8]](#footnote-9). As the algorithm is the same for both raw data and metadata, I speak only of space allocation requests in the following discussion, with the understanding that the request will be for either raw data or metadata, and that we will know which one it is whenever we need to.

Definitions:

req\_size: The size of a small raw data or metadata file space allocation request in bytes.

aggregator: The data structures and code use to allocate small pieces of raw data or metadata. In the HDF5 library proper, the data structure used for this purpose is struct H5F\_blk\_aggr\_t[[9]](#footnote-10), the definition of which is show in the next section. However, for conceptual purposes, we can think of it as a structure containing the address of the current allocation block (if any), and the length of the current allocation block.

allocation block: A block of file space allocated to an aggregator, and then subdivided (and possibly extended) to satisfy small raw data or metadata space allocation requests.

alloc\_size: The nominal size of aggregator allocation blocks. This value is user selectable.

Algorithm:

1. On file creation or open, the small raw data and metadata aggregators are created without allocation blocks.
2. If a space allocation request is received when the associated aggregator has no allocation block, processing depends on the size of the request.
	1. If req\_size is greater than or equal to alloc\_size, simply allocate the desired piece of metadata at the end of the file, extending the file as necessary. No allocation block is created in this case.
	2. If req\_size is less than alloc\_size, test to see if the *other* aggregator (i.e. the small raw data aggregator if this is a metadata space request, or the metadata aggregator if this is a raw data space request) has an allocation block defined, and if so, if it is at the end of file. If both conditions hold, de-allocate the other aggregators allocation block and adjust the end of file accordingly. Mark the other aggregator as having no allocation block.

Allocate an allocation block at the end of the file, assign it to the aggregator, and fulfill the space request out of that block.

Note that the aggregator will attempt to satisfy future space allocation requests out of the remainder of the allocation block just allocated.

1. If a space allocation request is received when the associated aggregator has an allocation block and there is sufficient space in the allocation block to satisfy the current request, make the allocation out of the allocator block and update the allocation block address and length accordingly.
2. If a space allocation request is received when the associated aggregator has an allocation block, and there is insufficient space in the allocation block to satisfy the request, processing again depends on the size of the request.
	1. If req\_size is greater than or equal to alloc\_size, test to see if the allocation block ends on the end of file. If it does, extend the end of file and the allocation block by req\_size, and then perform the desired allocation out of the allocation block.

If the allocation block does not end on the end of file, simply allocate the desired piece of file space at the end of file, and leave the aggregator unchanged.

* 1. If req\_size is less than alloc\_size, test to see if the allocation block ends on the end of file. If it does, extend the end of file and the allocation block by alloc\_size, and then perform the desired allocation out of the allocation block.

If the allocation block does not end on the end of file, proceed as follows:

1. Discard the existing allocation block.
2. Test to see if the *other* aggregator has an allocation block defined, and if it does, if it is at the end of file. If both conditions hold, de-allocate the other aggregators allocation block, mark the other aggregator as having no allocation block, and adjust the end of file.
3. Regardless of the results of 2) above, allocate a new aggregator block (of size alloc\_size) for the aggregator associated with the space request at the end of file. Fulfill the file space request out of this new allocation block, saving the rest for future allocations.

The above outline leaves out the issue of alignment requirements on space allocations. These in turn can result in snippets of waste space (which, in the absence of a free list manager, are simply discarded), and also in slight adjustments in the size of allocation block extensions and/or allocations.

While the alignments issue must be dealt with, I have left it out of the above discussion, as it adds significant complexity to the algorithm without making significant changes to the end result.

The full details may be seen in the detailed discussion of the code below, or in the code itself.

### Conceptual Overview of File Space De-allocation in the Absence of the Free List Manager

Conceptually, de-allocation of raw data or metadata file space in the absence of a free list manager is very simple.

1. First, check to see if the piece of file space to be freed is at the end of file, and reduce the end of file accordingly if it is.
2. If this fails, check to see if the piece of space is adjacent to either of the allocation blocks maintained by the metadata and small raw data aggregators. If it is, and we are currently configured to allow it, and if the piece of space is the same type (metadata or raw data) as the aggregator it is adjacent to, add the freed file space to the appropriate allocation block for re-use.
3. If this fails, simply discard the freed file space.

As shall be seen, the file space de-allocation code contains a bug that allows creation of a free list even when free lists are specifically disabled. This bug is described in the detailed discussion below, but omitted here.

### A Detailed Look at Small Raw Data and Metadata Aggregation in the Absence of the Free List Manager

While the above conceptual overview should give a good understanding of what the small raw data and metadata aggregators do in the absence of the free list managers, the actual code is more convoluted. The reason for this is unknown, but at a guess it is the result of hand optimization and repeated modifications.

This presumption (presence of the small raw data and metadata aggregators and absence of any free list manager) implies that the file space strategy is H5F\_FILE\_SPACE\_AGGR\_VFD.

In this case, H5MF\_alloc() simply calls H5MF\_aggr\_vfd\_alloc()[[10]](#footnote-11), which, in the case of a metadata space allocation request, in turn calls H5MF\_aggr\_alloc()[[11]](#footnote-12) with the file’s metadata aggregator listed at the principle aggregator, and the small raw data aggregator listed as the alternate aggregator. The parameters are reversed in the event of a raw data space allocation request.

As we will refer to the details of aggregators in what follows, the declaration of the associated H5F\_blk\_aggr\_t structure is shown below[[12]](#footnote-13):

/\* Structure for metadata & "small [raw] data" block aggregation fields \*/

struct H5F\_blk\_aggr\_t {

 unsigned long feature\_flag; /\* Feature flag type \*/

 hsize\_t alloc\_size; /\* Size for allocating new blocks \*/

 hsize\_t tot\_size; /\* Total amount of bytes aggregated into block \*/

 hsize\_t size; /\* Current size of block left \*/

 haddr\_t addr; /\* Location of block left \*/

};

The fields of a files small raw data and metadata aggregators are initialized in a call to H5F\_new()[[13]](#footnote-14) during file open or create. The alloc\_size fields are initialized to values obtained from the property list. The feature flag fields are unconditionally set to H5FD\_FEAT\_AGGREGATE\_SMALLDATA (small raw data aggregator) and H5FD\_FEAT\_AGGREGATE\_METADATA (metadata aggregator) in the same call. All other fields are initialized to zero.

Considering only the case in which the file space strategy is H5F\_FILE\_SPACE\_AGGR\_VFD, H5MF\_aggr\_alloc() proceeds as follows:

1. If alignment is generally disabled for the file, or if the size of the request (call it req\_size) exceeds some threshold, disable alignment for the purposes for this allocation. Note that this applies only for code in H5MF\_alloc() proper – alignment requirements may still be applied in functions called by H5MF\_alloc().
2. If the principal aggregator (call it “aggr”) refers to an allocation block at present, and that block contains any unallocated file space, calculate the amount of space remaining in the allocation block that cannot be used due to alignment considerations. Needless to say, this value will be zero if either there is no current block from which to allocate space, or if alignment is disabled. Call this value aggr\_frag\_size.
3. If aggregation is enabled, and req\_size plus the aggr\_frag\_size exceeds the amount of space remaining in the current allocation block, H5MF\_aggr\_alloc() must either extend the current allocation block, or create a new one.

Here there are two subcases, depending on whether req\_size exceeds the normal size of an aggregation block (aggr.alloc\_size).

* If req\_size is greater than or equal to aggr.alloc\_size, the function attempts to extend the allocation block by req\_size plus aggr\_frag\_size. This is done via a call to H5FD\_try\_extend()[[14]](#footnote-15).

If this attempt is successful, the request is satisfied using the block of length req\_size starting at aggr.addr + aggr\_frag\_size. The total size of the allocation block (aggr.tot\_size) and the address of the beginning of free space in the aggregator block (aggr.addr) are incremented by req\_size plus the aggr\_frag\_size. Note that the free space remaining in the aggregation block (aggr.size) remains unchanged.

If this attempt is unsuccessful, test to see if the alternate aggregator (call it “other\_aggr”) has an allocation block defined that is located at the end of file. If it does, free (via H5FD\_free[[15]](#footnote-16)) all remaining unallocated space in the alternate aggregator’s allocation block, and mark the alternate aggregator as having no allocation block.

Then simply allocate the desired space at the end of the file via H5FD\_alloc()[[16]](#footnote-17).

* If req\_size is less than aggr.alloc\_size, the function also attempts to extend the allocation block – but by a different amount, call it ext\_size. Initially, ext\_size is set to aggr.alloc\_size. However, if aggr\_frag\_size is greater than ext\_size minus req\_size, ext\_size is incremented by the amount by which aggr\_frag\_size exceeds ext\_size minus req\_size.

Once ext\_size is determined, the function attempts to extend the allocation block by that value via a call to H5FD\_try\_extend().

If this attempt is successful, the base address of the free space in the allocation block (aggr.addr) is incremented by aggr\_frag\_size, and the amount of space available in the allocation block (aggr.size) is incremented by ext\_size – aggr\_frag\_size. Note that the requested block is not allocated at this time.

If the attempt fails, the function first tests to see if the alternate aggregator (other\_aggr) is located at the end of file. If it is, the function frees (again via H5FD\_free()) all remaining unallocated space in the alternate aggregator’s allocation block, and marks the alternate aggregator as having no allocation block.

The function then allocates a new allocation block via H5FD\_alloc(), and then frees the remainder of the old one using H5MF\_xfree()[[17]](#footnote-18). aggr.addr is set to the address of the new block, and aggr.size and aggr.tot\_size are set to aggr.alloc\_size.

Regardless of whether the attempt to extend the primary aggregator’s allocation block succeeded, at this point, the primary aggregator should have sufficient space to satisfy the space request.

Satisfy the space request from the (either new or extended) allocation block. Thus aggr.addr is returned as the base address of the newly allocated block, aggr.addr is incremented by req\_size, and aggr.size is decremented by req\_size.

Regardless of whether req\_size is less than aggr.alloc\_size or not, some tidying up may be required.

If either H5FD\_extend() or H5FD\_alloc() was used to allocate new space, it is possible that, as a result of the alignment requirements, a fragment may have been allocated between the original end of file, and the beginning of the newly allocated block. If this occurred, the function frees the fragment using H5MF\_xfree().

Similarly, if, due to alignment considerations, a fragment is left between the beginning of the free space in the primary aggregator’s allocation block and beginning of the newly allocated block, the function frees this fragment as well, again using H5MF\_xfree().

1. Now consider the case in which aggregation is enabled, an allocation block exists, and the req\_size is less than or equal to aggr.size (the amount of free space remaining in the aggregator).

In this case, the function simply allocates the desired block out of the allocation block, returning aggr.addr + aggr\_frag\_size as the base address of the new block, incrementing aggr.addr by req\_size + aggr\_frag\_size, and decrementing aggr.size by the same value.

As with the cleanup in 3 above, if, due to alignment considerations, a fragment is left between the beginning of the free space in the metadata aggregator and beginning of the newly allocated block, the function frees this fragment using H5MF\_xfree().

1. Finally, there is the case in which aggregation is disabled. Here, the desired block is simply allocated at the end of the file via H5FD\_alloc(). Any fragment allocated after the desired block is freed via H5FM\_xfree().

### A Detailed Look at File Space De-Allocation in the Absence of the Free List Manager

The first point that is striking when examining H5MF\_xfree() is the apparent absence of any tests for the current file strategy. Indeed, under the correct circumstances, H5MF\_xfree() appears to create free list even when the free list manager is specifically disabled via the file space strategy.

After consulting with Quincey on this matter, it appears that I have stumbled over a bug that should be repaired in passing when the revised metadata aggregator is implemented. However, for now I simply describe the current behavior as I understand it – with the understanding that it is incorrect in this point at least. This bug should be fixed in passing when we implement the new metadata and small raw data aggregation scheme, and tests should be added to verify the fix.

With the above caveat in mind, H5MF\_xfree() proceeds as follows when the file strategy is H5F\_FILE\_SPACE\_AGGR\_VFD.

1. Perform a variety of initial sanity checks. Specifically:
	1. Verify that the address of the piece of file space to be freed is defined, and that its length is greater than zero.
	2. Verify that the address of the piece of space to be freed is not zero – the address of the super block.
	3. Verify that the file space to be freed does not intersect with the metadata accumulator. Recall that the metadata accumulator attempts to buffer adjacent metadata writes so that they can be combined in a single write operation.
	4. Verify that the file space to be freed is not “temporary” file space. Recall that temporary file space is file space beyond the end of the file that is assigned to temporary pieces of metadata that should never be written to file.
2. Map the allocation type of the file space to be freed (passed in via the alloc\_type parameter) to the free space type. This is necessary as several different memory types may be handled in the same free list. Call the free space type fs\_type.
3. Check to see if a free space manager has been defined for fs\_type.

If it hasn’t, first check to see if the file space to be freed is at the end of file. If it is, it can be freed by simply reducing the EOF. If this fails, check to see if the file space to be freed can be put into either the metadata or small raw data aggregator for re-use. Note that either of these options allows us to avoid starting up a free space manager.

Test these conditions by calling H5MF\_try\_shrink()[[18]](#footnote-19), which performs the tests and does the free by adjusting the EOF or incorporating the freed file space into an aggregator if possible. If H5MF\_try\_shrink() succeeds, exit indicating success.

If H5MF\_try\_shrink() fails, check to see if the size of the file space to be freed is less than the fs\_threshold field of the files shared instance H5F\_file\_t[[19]](#footnote-20). If it is, drop the file space to be freed on the floor, and exit indicating success.

Finally, if all the above attempts to avoid setting up a free space manager fail, call H5MF\_alloc\_open()[[20]](#footnote-21) to setup the free space manager for the desired free space type. As indicated above, this is a bug when free space managers are disabled.

Observe that the fs\_threshold field could be used to prevent allocation of a free space list when the files strategy is H5F\_FILE\_SPACE\_AGGR\_VFD. However, the value appears to be set by the user in H5Pset\_file\_space()[[21]](#footnote-22), and thus appears to be arbitrary.

1. If we get to this point, a free space manager exists to handle the piece of file space to be freed.

First, allocate an instance of H5MF\_free\_section\_t, and initialize it so that it describes the chunk of file space to be freed.

Then, if the size of the piece of file space to be freed is greater than or equal to the fs\_threshold discussed above, call H5FS\_sect\_add()[[22]](#footnote-23) to add the file space to be freed to the appropriate free list. Note that H5FS\_sec\_add() will attempt to merge the newly freed piece of file space with existing free space in the free list. Depending on whether we choose to allow pieces of metadata to span metadata aggregation blocks, this may have implications for this RFC, and may require further investigation.

Otherwise, if the size of the piece of file space to be freed is less than fs\_threshold, attempt to merge it with an existing piece of file space on the appropriate free list via a call to H5FS\_sect\_try\_merge()[[23]](#footnote-24). Again, depending on whether we choose to allow pieces of metadata to span metadata aggregation blocks, the particulars of this attempt may require further investigation. If the attempt to merge fails, just drop the free space on the floor.

The above discussion of space allocation and de-allocation in the metadata aggregator only case is very much a first cut. It should be simplified, and focused more narrowly as the proposed design for metadata data aggregation takes shape.

## Small Raw Data and Metadata Aggregation in the Presence of the Free List Manager

As we are not at present concerned with whether a free list is saved to the file or not, for purposes of this section we do not care whether the file space strategy is either H5F\_FILE\_SPACE\_ALL\_PERSIST or H5F\_FILE\_SPACE\_ALL.

### Conceptual Overview of Small Raw Data and Metadata Aggregation in the Presence of the Free List Manager

The algorithm for small raw data and metadata aggregation in the presence of the free list manager is an extension of the algorithm for allocating metadata in the absence of the free list manager.

Free list managers can be created for a variety of different metadata types. In particular, a free list manager can specialize in one type of metadata (say B-Tree nodes), several types of metadata, or all types of metadata. To map a specific kind of metadata to the appropriate free list manager, it is necessary to map the actual metadata type to the free space type – call this the fs\_type of the piece of metadata to be allocated.

Given this mapping, the revised algorithm for allocating file space can be described at the conceptual level as follows:

1. If a free list manager does not exist for the free space type of the file space request, create one.
2. Test to see if the free list manager for the fs\_type of the space request contains a block of adequate size.
	1. If it does, satisfy the metadata space request out of this block – trimming off the excess if necessary, and returning it to the free list manager.
	2. If it does not, allocate the desired file space as per the no free list manager algorithm (above).

Note that in this case, since free space managers are enabled, most if not all pieces of file space that are discarded by this algorithm in the no free space manager case, are instead added to the appropriate free list.

As before, I have neglected the complications associated with alignment requirements.

### Conceptual Overview of File Space De-Allocation in the Presence of the Free List Manager

Conceptually, de-allocation of metadata in the presence of the free list manager is almost the same as that used when free list managers are disabled.

1. First, check to see if the piece of file space to be freed is at the end of file, and reduce the end of file accordingly if it is.
2. If this fails, check to see if the piece of space is adjacent to either of the allocation blocks maintained by the metadata and small raw data aggregators. If it is, and if the piece of space is of the appropriate type, add the freed file space to the allocation block for re-use.
3. If this fails, test to see if a free list manager exists for the free space type of the freed file space. If it does, add the freed space to the free list manager.
4. If no such manager exists, and the size of the space to be freed exceeds some threshold, create a manager and add the space to it.
5. Otherwise, discard the space.

### A Detailed Look at Small Raw Data and Metadata Aggregation in the Presence of the Free List Manager

When free list managers are enabled, H5MF\_alloc() proceeds as follows:

1. Map the allocation type (passed in via the parameter list) to the free space type. This is necessary as several different memory types may be handled in the same free list. Call the free space type fs\_type.
2. Verify that the file space strategy is either H5F\_FILE\_SPACE\_ALL\_PERSIST or H5F\_FILE\_SPACE\_ALL. This is done via the H5F\_HAVE\_FREE\_SPACE\_MANAGER()[[24]](#footnote-25) macro.
3. Check to see if a free space manager has been defined for fs\_type. If it hasn’t, call H5MF\_alloc\_open()[[25]](#footnote-26) to set up the free space manager for the desired free space type. While we will go into the particulars of the free space managers later, for now suffice it to say that H5MF\_alloc\_open() calls H5FS\_open()[[26]](#footnote-27) for the specified free space type, and marks the free space manager as having been initialized.
4. Check to see if the free space manager contains a section (i.e. a block of unused file space) of size sufficient to satisfy the file space request. Do this via a call to H5FS\_sect\_find()[[27]](#footnote-28). Blocks are stored in instances of H5FS\_section\_info\_t – the declaration of which is given below (from H5FSprivate.h):

/\* Free space section info \*/

struct H5FS\_section\_info\_t {

 haddr\_t addr; /\* Offset of free space section in the address space \*/

 hsize\_t size; /\* Size of free space section \*/

 unsigned type; /\* Type of free space section (i.e. class) \*/

 H5FS\_section\_state\_t state; /\* Whether the section is in "serialized" or \*/

 /\* "live" form \*/

};

NOTE: At present, there is no difference between H5FS\_section\_info\_t and H5MF\_free\_section\_t (H5MFpkg.h) – although it is clear from comments that eventually the latter structure will become a superset of the former, containing the former as its first field. The exact plans should be investigated before we commit to any design for metadata aggregation.

H5FS\_sect\_find() handles some metadata cache entry locking issues, and then calls H5FS\_sect\_find\_node()[[28]](#footnote-29) to handle the actual search for a suitable block of memory.

If there are no alignment issues, H5FS\_sect\_find\_node() finds the smallest section of free space that will satisfy the request (in case of ties, the lower address wins) – if such a section exists. It removes the section (and its associated instance of H5FS\_section\_info\_t – call it \*si\_ptr) from the free list, and returns si\_ptr to the caller in \*node, a parameter to the function which is an instance of H5FS\_section\_info\_t \*\*.

If alignment is an issue, things are a bit more complicated, as H5FS\_sect\_find\_node() must find a section of free memory (if it exists), that is large enough to satisfy the request even after some fragment of the beginning of the block is shaved off to satisfy the alignment requirement. While the particulars are probably not relevant for the purposes of this RFC, it is worth noting that if the search is successful, the end result is more or less the same as in the non-aligned case, save that the non-aligned fragment of the chosen section is snipped off and returned to the free list. The pointer to the instance of H5FS\_section\_info\_t (possibly clipped for purposes of alignment) is returned in \*node as outlined above.

Regardless of alignment issues, if H5FS\_sect\_find\_node() is successful, it passes a pointer to the unlinked instance of H5FS\_section\_info\_t associated with the chunk of memory back to H5FS\_sect\_find(), which after handling some book keeping and unlocking, passes the pointer back to H5MF\_alloc().

1. If H5FS\_sect\_find() is successful, it returns the address of the section of memory that has been removed from the free list, possibly after trimming it to the desired size, and adding the fragment back to the free list.
2. If H5FS\_sect\_find() is unsuccessful, the desired section of file space is allocated via a call to H5MF\_aggr\_vfd\_alloc() as discussed in section 2.1.3 above. Note however, that in this case, the free lists are defined, and thus the various discarded fragments will be inserted onto a free list instead of being discarded.

### A Detailed Look at File Space De-Allocation in the Presence of the Free List Manager

As I understand it, the buggy version of H5MF\_xfree() described in section 2.1.4 above behaves correctly in this case. Thus, until such time as the bug is repaired, this section need only point to section 2.1.4 above.

## Free Space Managers

At the most elementary level the function of the free space managers is trivial – each free space manager maintains a list of blocks of file space that have been allocated for use as one of the target file space types, and then released for whatever reason. Adjacent blocks within a free space manager are combined. Conceptually, the set of free list managers in any open file will cover all possible memory types without overlap.

When file space of a type managed by the free space managers is requested, the manager responsible for the target memory type is queried to determine if it contains a suitable block. If it does, the block is removed from the free list manager, trimmed to size if need be (fragments return to the free list manager), and the resulting block is returned to the caller. If no such block exists, the request is satisfied by either the raw data or metadata aggregator, or by extending the file.

While the above is a good thumbnail sketch of what the free list managers do, for purposes of this document we need a much more detailed understanding of the free list managers. This need is addressed below. As with the discussions of the raw data and metadata aggregators, the detailed discussions are included for the implementer, and may be safely skipped by all other readers.

The implementer is advised that the detailed discussions are focused mainly on the data structures used by the free list managers. The discussions of allocation and de-allocation are much more cursory, and in many ways repeat material discussed above.

### A Conceptual Overview of the Data Structures Used by the Free List Managers

The data structures used by the free list managers are quite involved, and as such, the following overview will be inadequate for implementation purposes. However, it should provide a good conceptual overview for other readers, and a good starting point for the implementer.

It is suggested that the implementer supplement this section with a careful review the detailed discussion in section 2.3.4 below. Note, however, that this section too is quite cursory in spots, being directed only at determining the particulars of the data structures needed for the purposes of this document. As such, it skips over many details. While I hope that none of these details will bite the implementer, I can’t swear to it.

At the broadest conceptual level, the function of the free list manager data structures is trivial – they maintain a list of all blocks of freed file space in such a format as to allow easy identification of candidate blocks for reuse, for merging newly freed blocks of file space with existing free blocks when these are adjacent, and for easy save and reload from file. While not addressed in this document, it should also be mentioned that the free list managers are also employed to manager free space in the fractal heap – a consideration that appears to have had considerable impact on their design.

Turning to concrete issues, the first point to observe is that HDF5 recognizes six different types of memory – raw data and five flavors of metadata. To make matters more interesting, the multi file driver allows each of these types of data to segregated into an individual file and the associated segment of HDF5 file memory space. Under such circumstances, it is necessary to have a separate free space manager for each memory type. In contrast, under the more common circumstance in which all types of memory share a single file and memory space, only one free space manager is needed for all memory types.

Thus some method of mapping the type of memory being freed or requested to the appropriate free space manager is the first point to be considered. For file space at least, this is done with a lookup table that maps the integer associated with a memory type to an integer called the free space type. This free space type then indexes into an array of base addresses of free space manager data structures to yield the base address of the data structures of the free space manager to which such requests should be directed.

The top level structure in a free list manager’s data structure is an instance of H5FS\_t[[29]](#footnote-30). This structure contains a large number of fields – however, for purposes of this conceptual overview, there are only three that are of particular import.

The first of these is the cache\_info field, which is an instance of H5AC\_info\_t. This field, which must be the first field in the structure, is a structure that contains all fields necessary for managing the instance of H5FS\_t as an entry in the metadata cache.

The second is the sect\_cls field, which when initialized, contains a pointer to an array of H5FS\_section\_class\_t[[30]](#footnote-31). Each instance of this array appears to contain configuration data and callbacks for managing a particular class of free space. In the case of free space in the HDF5 file, this array will always contain exactly one element, whose value is defined in H5MF\_FSPACE\_SECT\_CLS\_SIMPLE[[31]](#footnote-32).

At this point, a brief digression into terminology is in order. While I may be corrected on this, the free list managers use the term “section” to describe a piece of free space in the HDF5 file. A section of free space is specified by its base address and length. Descriptions of sections of free space are stored in instances of H5FS\_section\_info\_t[[32]](#footnote-33). This structure also contains fields for the section type[[33]](#footnote-34) and state[[34]](#footnote-35).

With the notion of the section in hand, we can now discuss how the free list managers store lists of sections so as to facilitate merging of adjacent sections, and selection of sections for re-use.

The list of sections of free file space currently managed by the free space manager in maintained in a data structure whose root is an instance of H5FS\_sinfo\_t[[35]](#footnote-36). The sinfo field of the top level instance of H5FS\_t points to the free list manager’s instance of H5FS\_sinfo\_t. Note that H5FS\_sinfo\_t like H5FS\_t contains an instance of H5AC\_info\_t named cache\_info as its first field. As before, this allows instance of H5FS\_info\_t to be managed by the metadata cache. Note that of the structures discussed in this overview, only H5FS\_t and H5FS\_sinfo\_t contain this field.

Each instance of H5FS\_section\_info\_t managed by the free list manager is indexed by the two data structures rooted in the instance of H5FS\_sinfo\_t.

The simplest of these is used to identify sections that can be merged. A skip list serves for this purpose. Skip lists are a reasonably well known data structure, so I shall not describe them here, but the effect is to construct a list of all sections under management sorted in increasing address order. As this list can be searched, inserted into, and deleted from in O(log n) time, this allows easy identification of candidates for merging. The merge\_list field of H5FS\_sinfo\_t points to this skip list.

The structure used to support selecting a section to be used in satisfying a space request is considerably more complex – it can be loosely described as a vector of skip lists of skip lists.

 The “vector“ in this structure is an array of instances of H5FS\_bin\_t[[36]](#footnote-37) of length equal to the log base 2 of the maximum size of the HDF5 file. The base address of this array is stored in the bins field of H5FS\_sinfo\_t. Sections are mapped to entries in bins array by taking the log base 2 of the size of the section, and mapping the section to the bin with index equal to the resulting value rounded down.

Each non-empty instance of H5FS\_bin\_t has an associated skip list pointed to by the bin\_list field. However, the entries in the skip list are not instances of H5FS\_section\_info\_t, nor or they sorted by address. Instead the entries are instances of H5FS\_node\_t[[37]](#footnote-38), each of which contains a list of sections of a given size. The instances of H5FS\_node\_t in a skip list are sorted by the size of sections they index.

Each instance of H5FS\_node\_t has an associated skip list pointed to by its section\_list field. This skip list serves as a container for all sections for which the instance of H5FS\_node\_t is responsible. This skip list contains instances of H5FS\_section\_info\_t. All of these sections are of the same size – and thus this time the skip list is sorted by section base address.

The above should give the reader a good conceptual overview of the data structures used by the free list managers, and (I hope), also convey a good intuition of how the free list managers operate on these data structures. If the above is confusing, Figure 1 may be of value. While it doesn’t show many of the fields of the constituent structures, is does show how they are linked together.

Two points in closing:

First, it is worth repeating that each instance of H5FS\_section\_info\_t that represents a section of free memory managed by a given free list manager appears in two skip lists – that pointed to by the merge\_list field of H5FS\_sinfo\_t, and that pointed to by the sect\_list field of the instance of H5FS\_node\_t that manages sections of the indicated size.

Second, it seems that all the constituent structures that appear in the in memory representations of free list manager data structures are allocated and freed using HDF5 private free lists. The objective here seems to be to minimize the overhead of repeated calls to malloc()/calloc() and free. The implementer should take particular note of this, as it will affect debugging strategies.



Figure -- An Overview of the Free Space Manager Data Structures

### A Conceptual Overview of File Space Allocation in the Free List Managers

Given a good understanding of the free list manager data structures, the conceptual view of file space allocation in the Free List Managers becomes almost trivial.

After mapping the desired space type to the appropriate free manager (and initializing it if necessary), search for the smallest section of free space that will satisfy the space request. Note that alignment issues may complicate the search. Ties are broken by selecting the section of free space with the lowest base address. If a candidate is selected, remove it from the free list manager data structures, clip off any excess space, and return the fragments to the free space manager. Use the resulting piece of file free space to satisfy the request.

If the free space manager does not contain a suitable section of free space, satisfy the request from the metadata / raw data aggregator if possible, or by extending the EOF.

### A Conceptual Overview of File Space De-Allocation in the Free List Managers

Again, a good understanding of the free list manager data structures makes this section almost unnecessary.

To de-allocate a piece of file space, first see if it is at the end of the file, or if it can be added to one of the aggregators. If this fails, try to merge any adjacent entries in the free list into the newly freed section, and then insert the section into the free list.

###  A Detailed Look at the Free List Manager Data Structures and their Initialization

As with our examination of the raw data and metadata aggregators, for purposes of this discussion, file space allocation in the presence of the free list managers begins with a call to H5MF\_alloc(). As we assume that the free list managers are active, the file space strategy must be either H5F\_FILE\_SPACE\_ALL\_PERSIST or H5F\_FILE\_SPACE\_ALL — which implies that both the free list managers and the raw data and metadata aggregators are active. As we shall be interested in how free lists are stored and retrieved from file, we assume that the file space strategy is H5F\_FILE\_SPACE\_ALL\_PERSIST.

#### Mapping Memory Types to Free Space Types

As the reader likely recalls, at present, there are six file space memory types in HDF5, which are defined in the H5F\_mem\_t enumerated type[[38]](#footnote-39). This type declaration is reproduced below:

typedef enum H5F\_mem\_t {

 H5FD\_MEM\_NOLIST = -1, /\* Data should not appear in the free list.

 \* Must be negative.

 \*/

 H5FD\_MEM\_DEFAULT = 0, /\* Value not yet set. Can also be the

 \* datatype set in a larger allocation

 \* that will be suballocated by the library.

 \* Must be zero.

 \*/

 H5FD\_MEM\_SUPER = 1, /\* Superblock data \*/

 H5FD\_MEM\_BTREE = 2, /\* B-tree data \*/

 H5FD\_MEM\_DRAW = 3, /\* Raw data (content of datasets, etc.) \*/

 H5FD\_MEM\_GHEAP = 4, /\* Global heap data \*/

 H5FD\_MEM\_LHEAP = 5, /\* Local heap data \*/

 H5FD\_MEM\_OHDR = 6, /\* Object header data \*/

 H5FD\_MEM\_NTYPES /\* Sentinel value - must be last \*/

} H5F\_mem\_t;

In theory, we can have one free list manager for each memory type, although this is seldom the case. More commonly, we will have one free list manager, with that manager handling both raw data, and all the flavors of metadata.

This of course requires some method of mapping memory type to free list type. In the free list manager code, this is done via the H5MF\_ALLOC\_TO\_FS\_TYPE() macro[[39]](#footnote-40), whose definition is reproduced below:

#define H5MF\_ALLOC\_TO\_FS\_TYPE(F, T) ((H5FD\_MEM\_DEFAULT == (F)->shared->fs\_type\_map[T]) \

 ? (T) : (F)->shared->fs\_type\_map[T])

To follow what is going on here, we must start digging into the data structures used to support the free list managers for each file.

HDF5 maintains a single instance of struct H5F\_file\_t[[40]](#footnote-41) for each open HDF5 file. Note that there should be exactly one such structure for each open file, regardless of how many times that file has been opened.[[41]](#footnote-42)

The declaration of H5F\_file\_t is reproduced below. While I have reproduced the entire structure declaration, only the section entitled “file space allocation information” is of interest to us here. Note also that we will typically access this structure via the “shared” pointer in the instance of the H5F\_t[[42]](#footnote-43) structure associated with the open file. Unlike the instance of H5F\_file\_t, which is created once per open file regardless of how many times it is opened, a new instance of H5F\_t is created each time a file is opened. The idea is that all instances of H5F\_t for a given file point to the same instance of H5F\_file\_t.

/\*

 \* Define the structure to store the file information for HDF5 files. One of

 \* these structures is allocated per file, not per H5Fopen(). That is, set of

 \* H5F\_t structs can all point to the same H5F\_file\_t struct. The `nrefs'

 \* count in this struct indicates the number of H5F\_t structs which are

 \* pointing to this struct.

 \*/

typedef struct H5F\_file\_t {

 H5FD\_t \*lf; /\* Lower level file handle for I/O \*/

 H5F\_super\_t \*sblock; /\* Pointer to (pinned) superblock for file \*/

 unsigned nrefs; /\* Ref count for times file is opened \*/

 unsigned flags; /\* Access Permissions for file \*/

 H5F\_mtab\_t mtab; /\* File mount table \*/

 H5F\_efc\_t \*efc; /\* External file cache \*/

 /\* Cached values from FCPL/superblock \*/

 uint8\_t sizeof\_addr; /\* Size of addresses in file \*/

 uint8\_t sizeof\_size; /\* Size of offsets in file \*/

 haddr\_t sohm\_addr; /\* Relative address of shared object \*/

 /\* header message table \*/

 unsigned sohm\_vers; /\* Version of shared message table on disk \*/

 unsigned sohm\_nindexes; /\* Number of shared messages indexes in the table \*/

 unsigned long feature\_flags; /\* VFL Driver feature Flags \*/

 haddr\_t maxaddr; /\* Maximum address for file \*/

 H5AC\_t \*cache; /\* The object cache \*/

 H5AC\_cache\_config\_t

 mdc\_initCacheCfg; /\* initial configuration for the \*/

 /\* metadata cache. This structure is \*/

 /\* fixed at creation time and should \*/

 /\* not change thereafter. \*/

 hid\_t fcpl\_id; /\* File creation property list ID \*/

 H5F\_close\_degree\_t fc\_degree; /\* File close behavior degree \*/

 size\_t rdcc\_nslots; /\* Size of raw data chunk cache (slots) \*/

 size\_t rdcc\_nbytes; /\* Size of raw data chunk cache (bytes) \*/

 double rdcc\_w0; /\* Preempt read chunks first? [0.0..1.0]\*/

 size\_t sieve\_buf\_size; /\* Size of the data sieve buffer allocated (in bytes) \*/

 hsize\_t threshold; /\* Threshold for alignment \*/

 hsize\_t alignment; /\* Alignment \*/

 unsigned gc\_ref; /\* Garbage-collect references? \*/

 hbool\_t latest\_format; /\* Always use the latest format? \*/

 hbool\_t store\_msg\_crt\_idx; /\* Store creation index for object header messages? \*/

 unsigned ncwfs; /\* Num entries on cwfs list \*/

 struct H5HG\_heap\_t \*\*cwfs; /\* Global heap cache \*/

 struct H5G\_t \*root\_grp; /\* Open root group \*/

 H5FO\_t \*open\_objs; /\* Open objects in file \*/

 H5RC\_t \*grp\_btree\_shared; /\* Ref-counted group B-tree node info \*/

 /\* File space allocation information \*/

 H5F\_file\_space\_type\_t fs\_strategy; /\* File space handling strategy \*/

 hsize\_t fs\_threshold; /\* Free space section threshold \*/

 hbool\_t use\_tmp\_space; /\* Whether temp. file space allocation is allowed \*/

 haddr\_t tmp\_addr; /\* Next address to use for temp. space in the file \*/

 unsigned fs\_aggr\_merge[H5FD\_MEM\_NTYPES]; /\* Flags for whether free space can \*/

 /\* merge with aggregator(s) \*/

 H5F\_fs\_state\_t fs\_state[H5FD\_MEM\_NTYPES]; /\* State of free space manager for \*/

 /\* each type \*/

 haddr\_t fs\_addr[H5FD\_MEM\_NTYPES]; /\* Address of free space manager info for \*/

 /\* each type \*/

 H5FS\_t \*fs\_man[H5FD\_MEM\_NTYPES]; /\* Free space manager for each file space type \*/

 H5FD\_mem\_t fs\_type\_map[H5FD\_MEM\_NTYPES]; /\* Mapping of "real" file space type \*/

 /\* into tracked type \*/

 H5F\_blk\_aggr\_t meta\_aggr; /\* Metadata aggregation info \*/

 /\* (if aggregating metadata allocations) \*/

 H5F\_blk\_aggr\_t sdata\_aggr; /\* "Small data" aggregation info \*/

 /\* (if aggregating "small data" allocations) \*/

 /\* Metadata accumulator information \*/

 H5F\_meta\_accum\_t accum; /\* Metadata accumulator info \*/

} H5F\_file\_t;

Returning to the issue of mapping HDF5 file space types to free list types, we can see that the fs\_type\_map[] array in H5F\_File\_t supports this function in combination with the H5MF\_ALLOC\_TO\_FS\_TYPE() macro. The fs\_type\_map[] array is initialized via a call to H5FD\_get\_fs\_type\_map()[[43]](#footnote-44) shortly after the instance of H5F\_file\_t is allocated in H5F\_new()[[44]](#footnote-45). The H5FD\_get\_fs\_type\_map() call is passed through to the underlying Virtual File Driver (VFD), which initializes the fs\_type\_map[] array as indicated by the following table:

|  |  |
| --- | --- |
| VFD: | fs\_type\_map initialization: |
| sec2 | H5FD\_FLMAP\_SINGLE[[45]](#footnote-46) |
| stdio | H5FD\_FLMAP\_SINGLE |
| core | H5FD\_FLMAP\_SINGLE |
| family | H5FD\_FLMAP\_SINGLE |
| mpi | H5FD\_FLMAP\_SINGLE |
| mpiposix | H5FD\_FLMAP\_SINGLE |
| split/multi | H5FD\_FLMAP\_DEFAULT[[46]](#footnote-47) |

The entry for the split/multi file driver is deceptive. While H5FD\_FLMAP\_DEFAULT is the value specified in the fl\_map field of the instance of H5FD\_class\_t associated with the multi file driver, H5FD\_multi\_get\_type\_map() does not use this value, and instead returns the member map specified in the FAPL. The effect of this is to create one free list type for each file, with said free list managing all memory types assigned to the file.

#### Free List Manager Data Structures Proper

Once the memory type is mapped to the free list type, the base address of the data structure for the target free list manager can be looked up by using the free list type as an index into the fs\_man field in H5F\_file\_t (see above) at the index indicated by the free space type. Entries in the fs\_man field are NULL if either no free list manager is defined for the corresponding free list type, or if the free list manager has not yet been created.

While we shall trace through the code that inserts freed file space into the free list managers, and that attempts to service file space requests using the free list managers shortly, for now the objective is to survey the data structures used by the free list managers so that their use will be easy to follow when we get there.

While the data structures maintained by a free list manager can be saved to and reloaded from disk, they are initially created when a piece of file space is freed, and its memory type maps to a free space type, which indexes a NULL entry in the fs\_man array in the files instance of H5F\_file\_t[[47]](#footnote-48). This triggers a call to H5MF\_alloc\_start()[[48]](#footnote-49) – which performs some sanity checking and then calls H5MF\_alloc\_create()[[49]](#footnote-50). That function in turn loads an instance of H5FS\_create\_t[[50]](#footnote-51) with parameters needed to initialize the free list manager, calls H5FS\_create()[[51]](#footnote-52), and, on success, sets the entry in the fs\_man and fs\_state fields of H5F\_file\_t indexed by the free space to the address of the newly created free list manager data structures and to H5F\_FS\_STATE\_OPEN respectively.

The definition of H5FS\_create\_t is reproduced below, followed by the code initializing it in H5MF\_alloc\_create().

/\* Free space creation parameters \*/

typedef struct H5FS\_create\_t {

 H5FS\_client\_t client; /\* Client's ID \*/

 unsigned shrink\_percent; /\* Percent of "normal" serialized size to shrink \*/

 /\* serialized space at \*/

 unsigned expand\_percent; /\* Percent of "normal" serialized size to expand \*/

 /\* serialized space at \*/

 unsigned max\_sect\_addr; /\* Size of address space free sections are within \*/

 /\* (log2 of actual value) \*/

 hsize\_t max\_sect\_size; /\* Maximum size of section to track \*/

} H5FS\_create\_t;

/\* initialization of instance of H5FS\_create\_t passed to H5FS\_create() \*/

/\* by H5MF\_alloc\_create() \*/

fs\_create.client = H5FS\_CLIENT\_FILE\_ID;

fs\_create.shrink\_percent = H5MF\_FSPACE\_SHRINK;

fs\_create.expand\_percent = H5MF\_FSPACE\_EXPAND;

fs\_create.max\_sect\_addr = 1 + H5V\_log2\_gen((uint64\_t)f->shared->maxaddr);

fs\_create.max\_sect\_size = f->shared->maxaddr;

The declaration of H5FS\_create() is given below:

 H5FS\_t \* H5FS\_create(H5F\_t \*f,

 hid\_t dxpl\_id,

 haddr\_t \*fs\_addr,

 const H5FS\_create\_t \*fs\_create,

 size\_t nclasses,

 const H5FS\_section\_class\_t \*classes[],

 void \*cls\_init\_udata,

 hsize\_t alignment,

 hsize\_t threshold);

When calling H5FD\_create(), H5MF\_alloc\_create() sets the parameters of H5FS\_create() as follows:

f 🡪 f (pointer to H5F\_t passed into H5MF\_alloc\_create())

dxpl\_id 🡪 dxpl\_id (the DXPL ID passed into H5MF\_alloc\_create())

fs\_addr 🡪 NULL

fs\_create 🡪 pointer to an instance of H5FS\_create\_t initialized as shown above.

nclasses 🡪 1

classes 🡪 pointer to an array of pointer to H5FS\_section\_class\_t.

This array contains one element, which points to H5MF\_FSPACE\_SECT\_CLS\_SIMPLE[[52]](#footnote-53), whose definition is reproduced below.

cls\_init\_udata 🡪 f (pointer to H5F\_t passed into H5MF\_alloc\_create())

alignment 🡪 f->shared->alignment

threshold 🡪 f->shared->threshold

As we will have occasion to refer to them later, the definitions of H5FS\_section\_class\_t and H5MF\_FSPACE\_SECT\_CLS\_SIMPLE are reproduced below:

/\* Free space section class info \*/

typedef struct H5FS\_section\_class\_t {

 /\* Class variables \*/

 const unsigned type; /\* Type of free space section \*/

 size\_t serial\_size; /\* Size of serialized form of section \*/

 unsigned flags; /\* Class flags \*/

 void \*cls\_private; /\* Class private information \*/

 /\* Class methods \*/

 herr\_t (\*init\_cls)(struct H5FS\_section\_class\_t \*, void \*); /\* Routine to initialize \*/

 /\* class-specific settings \*/

 herr\_t (\*term\_cls)(struct H5FS\_section\_class\_t \*); /\* Routine to terminate \*/

 /\* class-specific settings \*/

 /\* Object methods \*/

 herr\_t (\*add)(H5FS\_section\_info\_t \*, unsigned \*, void \*); /\* Routine called when \*/

 /\* section is about to be added to manager \*/

 herr\_t (\*serialize)(const struct H5FS\_section\_class\_t \*,

 const H5FS\_section\_info\_t \*, uint8\_t \*);/\* Routine to serialize \*/

/\* a "live" section into a buffer \*/

 H5FS\_section\_info\_t \*(\*deserialize)(const struct H5FS\_section\_class\_t \*,

hid\_t dxpl\_id, const uint8\_t \*, haddr\_t, hsize\_t, unsigned \*);

/\* Routine to deserialize a buffer into a "live" section \*/

 htri\_t (\*can\_merge)(const H5FS\_section\_info\_t \*, const H5FS\_section\_info\_t \*, void \*);

 /\* Routine to determine if two nodes are mergable \*/

 herr\_t (\*merge)(H5FS\_section\_info\_t \*, H5FS\_section\_info\_t \*, void \*);

/\* Routine to merge two nodes \*/

 htri\_t (\*can\_shrink)(const H5FS\_section\_info\_t \*, void \*); /\* Routine to determine \*/

/\* if node can shrink container \*/

 herr\_t (\*shrink)(H5FS\_section\_info\_t \*\*, void \*); /\* Routine to shrink container \*/

 herr\_t (\*free)(H5FS\_section\_info\_t \*); /\* Routine to free node \*/

 herr\_t (\*valid)(const struct H5FS\_section\_class\_t \*, const H5FS\_section\_info\_t \*);

 /\* Routine to check if a section is valid \*/

 H5FS\_section\_info\_t \*(\*split)(H5FS\_section\_info\_t \*, hsize\_t);

/\* Routine to create the split section \*/

 herr\_t (\*debug)(const H5FS\_section\_info\_t \*, FILE \*, int , int );

/\* Routine to dump debugging information about a section \*/

} H5FS\_section\_class\_t;

/\* Class info for "simple" free space sections \*/

H5FS\_section\_class\_t H5MF\_FSPACE\_SECT\_CLS\_SIMPLE[1] = {{

 /\* Class variables \*/

 H5MF\_FSPACE\_SECT\_SIMPLE, /\* Section type \*/

 0, /\* Extra serialized size \*/

 H5FS\_CLS\_MERGE\_SYM | H5FS\_CLS\_ADJUST\_OK, /\* Class flags \*/

 NULL, /\* Class private info \*/

 /\* Class methods \*/

 NULL, /\* Initialize section class \*/

 NULL, /\* Terminate section class \*/

 /\* Object methods \*/

 NULL, /\* Add section \*/

 NULL, /\* Serialize section \*/

 H5MF\_sect\_simple\_deserialize, /\* Deserialize section \*/

 H5MF\_sect\_simple\_can\_merge, /\* Can sections merge? \*/

 H5MF\_sect\_simple\_merge, /\* Merge sections \*/

 H5MF\_sect\_simple\_can\_shrink, /\* Can section shrink container?\*/

 H5MF\_sect\_simple\_shrink, /\* Shrink container w/section \*/

 H5MF\_sect\_simple\_free, /\* Free section \*/

 H5MF\_sect\_simple\_valid, /\* Check validity of section \*/

 H5MF\_sect\_simple\_split, /\* Split section node for alignment \*/

 NULL, /\* Dump debugging for section \*/

}};

On entry, H5FS\_create() performs some sanity checks, and then calls H5FS\_new()[[53]](#footnote-54) to allocate the free list manager data structures. After some sanity checks, H5FS\_new() starts by allocating an instance of H5FS\_t[[54]](#footnote-55), the base address of will eventually be stored in the fs\_man field in the files instance of H5F\_file\_t (see above) at the index indicated by the free space type. The definition of H5FS\_t is reproduced below:

/\* Free space header info \*/

struct H5FS\_t {

 /\* Information for H5AC cache functions, \_must\_ be first field in structure \*/

 H5AC\_info\_t cache\_info;

 /\* Stored information \*/

 /\* Statistics about sections managed \*/

 hsize\_t tot\_space; /\* Total amount of space tracked \*/

 hsize\_t tot\_sect\_count; /\* Total # of sections tracked \*/

 hsize\_t serial\_sect\_count; /\* # of serializable sections tracked \*/

 hsize\_t ghost\_sect\_count; /\* # of un-serializable sections tracked \*/

 /\* Creation parameters \*/

 H5FS\_client\_t client; /\* Type of user of this free space manager \*/

 unsigned nclasses; /\* Number of section classes handled \*/

 unsigned shrink\_percent; /\* Percent of "normal" serialized size \*/

 /\* to shrink serialized space at \*/

 unsigned expand\_percent; /\* Percent of "normal" serialized size to \*/

 /\* expand serialized space at \*/

 unsigned max\_sect\_addr; /\* Size of address space free sections are within \*/

 /\* (log2 of actual value) \*/

 hsize\_t max\_sect\_size; /\* Maximum size of section to track \*/

 /\* Serialized section information \*/

 haddr\_t sect\_addr; /\* Address of the section info in the file \*/

 hsize\_t sect\_size; /\* Size of the section info in the file \*/

 hsize\_t alloc\_sect\_size; /\* Allocated size of the section info in the file \*/

 /\* Computed/cached values \*/

 unsigned rc; /\* Count of outstanding references to struct \*/

 haddr\_t addr; /\* Address of free space header on disk \*/

 size\_t hdr\_size; /\* Size of free space header on disk \*/

 H5FS\_sinfo\_t \*sinfo; /\* Section information \*/

 unsigned sinfo\_lock\_count; /\* # of times the section info has been locked \*/

 hbool\_t sinfo\_protected; /\* Whether the section info was protected when locked \*/

 hbool\_t sinfo\_modified; /\* Whether the section info has been modified while \*/

 /\* locked \*/

 H5AC\_protect\_t sinfo\_accmode; /\* Access mode for protecting the section info \*/

 size\_t max\_cls\_serial\_size; /\* Max. additional size of serialized form of section \*/

 hsize\_t threshold; /\* Threshold for alignment \*/

 hsize\_t alignment; /\* Alignment \*/

 /\* Memory data structures (not stored directly) \*/

 H5FS\_section\_class\_t \*sect\_cls; /\* Array of section classes for this free list \*/

};

The instance of H5FS\_t is allocated via a call to H5FL\_CALLOC()[[55]](#footnote-56). While this seems to indicate that HDF5 will normally maintain a free list of instances of H5FS\_t so as to avoid the overhead of repeated calloc() and free() calls, for most practical purposes, this should be the same as regular calloc() call. Thus, except as specified, all fields are initialized to 0 or NULL. After storing the newly allocated instance of H5FS\_t in the fspace local variable, H5FS\_new() proceeds as follows:

1. Set fspace->nclasses equal to the value of the nclasses parameter (1 in this case).
2. Set fspace->sect\_cls equal to H5FL\_SEQ\_MALLOC(H5FS\_section\_class\_t, nclasses). The H5FL\_SEQ\_MALLOC()[[56]](#footnote-57) macro is defined as follows:

#define H5FL\_SEQ\_MALLOC(t,elem) \

(t \*)H5FL\_seq\_malloc(&(H5FL\_SEQ\_NAME(t)),elem H5FL\_TRACK\_INFO)

with the H5FL\_SEQ\_NAME()[[57]](#footnote-58) macro being defined as:

#define H5FL\_SEQ\_NAME(t) H5\_##t##\_seq\_free\_list

and the H5FL\_TRACK macro being either undefined, or defined as:

#define H5FL\_TRACK\_INFO ,\_\_FILE\_\_, FUNC, \_\_LINE\_\_

Depending on whether H5FL\_TRACK is defined. Thus, assuming that H5FL\_TRACK is undefined, the above macro resolves to:

(H5FS\_section\_class\_t )H5FL\_seq\_malloc(&(H5\_ H5FS\_section\_class\_t\_seq\_free\_list), \

nclasses)

or, since we know that nclasses is 1:

(H5FS\_section\_class\_t )H5FL\_seq\_malloc(&(H5\_H5FS\_section\_class\_t\_seq\_free\_list),1)

After some sanity checking, the call to H5FL\_seq\_malloc()[[58]](#footnote-59) resolves to:

H5FL\_blk\_malloc(&(((&(H5\_H5FS\_section\_class\_t\_seq\_free\_list))->queue),

(&(H5\_H5FS\_section\_class\_t\_seq\_free\_list))->size \* 1);

H5FL\_blk\_malloc()[[59]](#footnote-60) is one of a group of functions that maintain free lists of dynamically allocated blocks of memory using instances of H5FL\_blk\_head\_t[[60]](#footnote-61). The objective seems to be to avoid the performance overhead occasioned by heavy use of the usual malloc() and free() library calls. This seems like overkill for file level free list managers, which are created at most eight times in a file open/close cycle, and usually only once. However, the free list managers are also used for the fractal heaps, and there may be an issue there.

In any case, the net effect of all the above is to allocate an instance of H5FS\_section\_class\_t and store its address in fspace->sect\_cls. For now at least, I don’t see the need to investigate this further.

1. Copy the single instance H5FS\_section\_class\_t in the array of same pointed to by the classes parameter into the instance of H5FS\_section\_class\_t pointed to by fspace->sect\_cls. Note that if an initialization routine were specified in the provided instance of H5FS\_section\_class\_t, it would be called here. The function also sets fspace->max\_cls\_serial\_size equal to the maximum of the serial\_size fields of the supplied array of instances for H5FS\_section\_class\_t. As this array of length one in this case, and the serial\_size field of its one element contains zero, this is also a NO-OP in the case at hand.
2. Perform the following initializations:

 /\* Initialize non-zero information for new free space manager \*/

 fspace->addr = HADDR\_UNDEF;

 fspace->hdr\_size = H5FS\_HEADER\_SIZE(f);

 fspace->sect\_addr = HADDR\_UNDEF;

1. Return the address of the new instance of H5FS\_t.

After H5FS\_new() returns the address of the newly allocated and partially initialized instance of H5FS\_t, H5FS\_create() stores the address in fspace and proceeds as follows:

1. Perform the following initializations from the instance of H5FS\_create\_t passed to H5FS\_create() in the fs\_create parameter:

/\* Initialize creation information for free space manager \*/

fspace->client = fs\_create->client;

fspace->shrink\_percent = fs\_create->shrink\_percent;

fspace->expand\_percent = fs\_create->expand\_percent;

fspace->max\_sect\_addr = fs\_create->max\_sect\_addr;

fspace->max\_sect\_size = fs\_create->max\_sect\_size;

Given the initialization of the fs\_create parameter given above, these initialization resolve to:

fspace->client = H5FS\_CLIENT\_FILE\_ID;

fspace->shrink\_percent = H5MF\_FSPACE\_SHRINK;

fspace->expand\_percent = H5MF\_FSPACE\_EXPAND;

fspace->max\_sect\_addr = 1 + H5V\_log2\_gen((uint64\_t)f->shared->maxaddr);

fspace->max\_sect\_size = f->shared->maxaddr;

1. Initialize the alignment and threshold fields of \*fspace with the values of the parameters of the same name as follows:

fspace->alignment = alignment;

fspace->threshold = threshold;

Given the values passed in, these initializations resolve to:

fspace->alignment = f->shared->alignment;

fspace->threshold = f->shared->threshold;

1. If the fs\_addr parameter is not NULL, allocate space for the free space header in the file and insert the header into the metadata cache. Since the fs\_addr parameter is NULL in this case, this is a NO-OP. Since the file space strategy is H5F\_FILE\_SPACE\_ALL, we have to allocate file space for the free list manager eventually. However, it seems that we don’t do this here.
2. Set the reference count field in \*fspace to 1 as follows:

fspace->rc = 1;

If the comments in the code are to be believed, this may be an error, as the reason given for the operation is fact that we have just inserted the new instance of H5FS\_t in the metadata data cache. However, this has not been done, since the fs\_addr parameter is NULL.

1. Return the address of the newly allocated and initialized instance of H5FS\_t to the caller.

Popping back up the call stack, when H5FS\_create() reports success and returns the newly allocated and initialized instance of H5FS\_t, H5MF\_alloc\_create() stores the address in f->shared->fs\_man[type], and sets f->shared->fs\_state[type] = H5F\_FS\_STATE\_OPEN. Here, type is the free space type of the newly allocated and initialized free space manager.

At this point, one might be forgiven for expecting the free space manager data structures to be fully initialized. However, that is not the case, as the data structure appears to be created lazily. Thus to continue our examination of the free list manager data structures, we must look at what happens when an entry is inserted into the free list.

To do this, we return to H5MF\_xfree(), and follow the thread from where the function attempts to insert a block of freed file space into the free list. This process starts with a call to H5MF\_sect\_simple\_new()[[61]](#footnote-62). This function takes as parameters the base address and size of the piece of file space to be freed, and proceeds as follows:

1. Allocate a new instance of H5MF\_free\_section\_t[[62]](#footnote-63) via a call to the H5FL\_MALLOC()[[63]](#footnote-64) macro, and saving the address of the new instance in the sect local var.

The definition of the H5FL\_MALLOC() calls is as follows:

/\* Allocate an object of type 't' \*/

#define H5FL\_MALLOC(t) (t \*)H5FL\_reg\_malloc(&(H5FL\_REG\_NAME(t)) H5FL\_TRACK\_INFO)

The definition of the H5FL\_REG\_NAME()[[64]](#footnote-65) macro is:

#define H5FL\_REG\_NAME(t) H5\_##t##\_reg\_free\_list

As the reader may recall, the H5FL\_TRACK\_INFO macro resolves to the empty string when H5FL\_TRACK is undefined, and thus the call to H5FL\_MALLOC() given below:

sect = H5FL\_MALLOC(H5MF\_free\_section\_t);

normally resolves to:

sect=(H5MF\_free\_section\_t \*)H5FL\_reg\_malloc(H5\_H5MF\_free\_section\_t\_reg\_free\_list);

On examination, it seems that H5FL\_reg\_malloc()[[65]](#footnote-66) is again part of an attempt to avoid the overhead of many calls to malloc() and free() by maintaining a free list of (in this case) instances of H5MF\_free\_section\_t. As before, I will not investigate this further at present, as it seems orthogonal to the questions at hand.

In any case, it seems that the net effect of the above is a malloc() of an instance of H5MF\_free\_section\_t.

The definition of this data structure is somewhat peculiar – at present, it is contains only one field, which is an instance of H5FS\_section\_info\_t[[66]](#footnote-67). The remainder of the structure is commented out. Clearly, there are plans for H5MF\_free\_section\_t, but other than mentioning this, I shall not attempt to delve further at present.

For later reference, the definitions of H5MF\_free\_section\_t (minus all commented out fields) and of H5FL\_section\_info\_t are reproduced below:

/\* File free space section info \*/

typedef struct H5MF\_free\_section\_t {

 H5FS\_section\_info\_t sect\_info; /\* Free space section information (must be

 /\* first in struct) \*/

 /\* many commented out fields elided \*/

} H5MF\_free\_section\_t;

/\* Free space section info \*/

struct H5FS\_section\_info\_t {

 haddr\_t addr; /\* Offset of free space section in the address space \*/

 hsize\_t size; /\* Size of free space section \*/

 unsigned type; /\* Type of free space section (i.e. class) \*/

 H5FS\_section\_state\_t state; /\* Whether the section is in "serialized" or \*/

 /\* "live" form \*/

};

1. Initialize the new instance of H5MF\_free\_section\_t as follows:

 /\* Set the information passed in \*/

 sect->sect\_info.addr = sect\_off; /\* base addr of the file space to be freed \*/

 sect->sect\_info.size = sect\_size; /\* length of the file space to be freed \*/

 /\* Set the section's class & state \*/

 sect->sect\_info.type = H5MF\_FSPACE\_SECT\_SIMPLE;

 sect->sect\_info.state = H5FS\_SECT\_LIVE;

1. Return a pointer to the new instance of H5MF\_free\_section\_t to the caller.

When H5MF\_sect\_simple\_new() returns, H5MF\_xfree() stores the pointer returned in the local variable node. It then initializes an instance of H5MF\_sect\_ud\_t[[67]](#footnote-68) as shown below (structure definition followed by initialization – in H5MF\_xfree(), udata is a local variable of type H5MF\_sect\_ud\_t). Note that the last two fields are not initialized.

/\* User data for free space manager section callbacks \*/

typedef struct H5MF\_sect\_ud\_t {

 /\* Down \*/

 H5F\_t \*f; /\* Pointer to file to operate on \*/

 hid\_t dxpl\_id; /\* DXPL for VFD operations \*/

 H5FD\_mem\_t alloc\_type; /\* Type of memory being allocated \*/

 hbool\_t allow\_sect\_absorb; /\* Whether sections are allowed to absorb \*/

 /\* a block aggregator \*/

 /\* Up \*/

 H5MF\_shrink\_type\_t shrink; /\* Type of shrink operation to perform \*/

 H5F\_blk\_aggr\_t \*aggr; /\* Aggregator block to operate on \*/

} H5MF\_sect\_ud\_t;

/\* Construct user data for callbacks \*/

udata.f = f; /\* this is the file pointer passed into H5MF\_xfree() \*/

udata.dxpl\_id = dxpl\_id; /\* this is the DXPL ID passed into H5MF\_xfree() \*/

udata.alloc\_type = alloc\_type; /\* this is the type of the file space being freed \*/

udata.allow\_sect\_absorb = TRUE;

After udata is initialized, H5MF\_xfree() tests to see if the size of the block to be freed exceeds f->shared->fs\_threshold. If it does, the function calls H5FS\_sect\_add()[[68]](#footnote-69) as shown below:

 H5FS\_sect\_add(f, dxpl\_id, f->shared->fs\_man[fs\_type], \

 (H5FS\_section\_info\_t \*)node, H5FS\_ADD\_RETURNED\_SPACE, &udata)

Note the coercion of node to pointer to H5FS\_section\_info\_t.

On invocation, H5FS\_sect\_add() proceeds as follows after some sanity checks. Note that within the function, fspace is the third parameter given above. It is a pointer to the top level free space manager structure (an instance of H5FS\_t) – in this case the free list manager data structures for the file space of the type being freed.

1. Call H5FS\_sinfo\_lock(f, dxpl\_id, fspace, H5AC\_WRITE)

According to its header comment, the purpose of H5FS\_sinfo\_lock()[[69]](#footnote-70) is to “Make certain the section info for the free space manager is in memory”. From a brief look at the code, it appears to do this by either loading the section info from file via the metadata data cache, locking it if it is already in the cache, or allocating it if it doesn’t exist at all. As the current objective is to divine the structure and use of the free list manager data structures, I will concern myself only with the allocation branch at this time.

This allocation is done via a call to H5FS\_sinfo\_new(H5F\_t \*f, H5FS\_t \* fspace)[[70]](#footnote-71), with (in this case) f being a pointer to the file in question, and fspace being a pointer free space manager data structures being extended. After some sanity checking, H5FS\_sinfo\_new() proceeds as follows:

* 1. Allocate a new instance of H5FS\_sinfo\_t[[71]](#footnote-72) via the macro call H5FL\_CALLOC(H5FS\_sinfo\_t).

Having traced through a couple of these macro calls already, I will assume that this is yet another dodge to avoid lots of malloc() and free() calls through maintenance of a free list. I will assume that the net effect of this call is the same as a regular calloc(). This may be wrong, so best keep an eye out for contradictions.

* 1. Initialize the non-zero fields of the new instance of H5FS\_sinfo\_t as follows:

/\* Set non-zero values \*/

sinfo->nbins = H5V\_log2\_gen(fspace->max\_sect\_size);

sinfo->sect\_prefix\_size = (size\_t)H5FS\_SINFO\_PREFIX\_SIZE(f);

sinfo->sect\_off\_size = (fspace->max\_sect\_addr + 7) / 8;

sinfo->sect\_len\_size = H5V\_limit\_enc\_size((uint64\_t)fspace->max\_sect\_size);

Recall that fspace->max\_sect\_size has been initialized to f->shared->maxaddr. While this value is user selectable and is driven by the current storage technology, today a typical value would be 264.

Similarly, fspace->max\_sect\_addr is initialized to

1 + H5V\_log2\_gen((uint64\_t)f->shared->maxaddr).

Examining the code for H5V\_log2\_gen()[[72]](#footnote-73) and using the above typical value for f->shared->maxaddr yields a typical value of 65.

Combining these typical values and examining the code for H5V\_linit\_enc\_size()[[73]](#footnote-74) gives the following typical values for the above fields:

sinfo->nbins = 64;

sinfo->sect\_prefix\_size = some relatively small number;

sinfo->sect\_off\_size = (65 + 7) / 8 = 9;

sinfo->sect\_len\_size = 8;

Note the imprecise number given for sinfo->sect\_prefix\_size. If desired, a more precise value could be obtained by unraveling the H5FS\_SINFO\_PREFIX\_SIZE(f)[[74]](#footnote-75) macro. However, since this is just the size of a header on disk, the exact value doesn’t seem important in the present context.

* 1. Allocate space for the section size bins via a call to

H5FL\_SEQ\_CALLOC(H5FS\_bin\_t, (size\_t)sinfo->nbins).

Per previous investigations, this has the net effect of calloc-ing an array of instances of H5FS\_bin\_t[[75]](#footnote-76) of length sinfo->nbins.

Store the base address of the new array in sinfo->bins.

* 1. Set fspace->sinfo = sinfo.
	2. Return sinfo.

On the return of H5FS\_sinfo\_new(),H5FS\_sinfo\_lock() redundantly sets fspace->sinfo equal to the return value, and increments fspace->sinfo\_lock\_count prior to returning.

Before continuing, I reproduce the definitions of H5FS\_info\_t and H5FS\_bin\_t below:

/\* Free space section info \*/

typedef struct H5FS\_sinfo\_t {

 /\* Information for H5AC cache functions, \_must\_ be first field in structure \*/

 H5AC\_info\_t cache\_info;

/\* Stored information \*/

 H5FS\_bin\_t \*bins; /\* Array of lists of lists of free sections \*/

/\* Computed/cached values \*/

 hbool\_t dirty; /\* Whether this info in memory is out of \*/

 /\* sync w/info in file \*/

 unsigned nbins; /\* Number of bins \*/

 size\_t serial\_size; /\* Total size of all serializable sections \*/

 size\_t tot\_size\_count; /\* Total number of differently sized sections \*/

 size\_t serial\_size\_count; /\* Total number of differently sized \*/

 /\* serializable sections \*/

 size\_t ghost\_size\_count; /\* Total number of differently sized \*/

 /\* un-serializable \*/

 /\* sections \*/

 unsigned sect\_prefix\_size; /\* Size of the section serialization prefix \*/

 /\* (in bytes) \*/

 unsigned sect\_off\_size; /\* Size of a section offset (in bytes) \*/

 unsigned sect\_len\_size; /\* Size of a section length (in bytes) \*/

 H5FS\_t \*fspace; /\* Pointer to free space manager that owns \*/

 /\* sections \*/

/\* Memory data structures (not stored directly) \*/

 H5SL\_t \*merge\_list; /\* Skip list to hold sections for detecting \*/

 /\* merges \*/

} H5FS\_sinfo\_t;

/\* Free space section bin info \*/

typedef struct H5FS\_bin\_t {

 size\_t tot\_sect\_count; /\* Total # of sections in this bin \*/

 size\_t serial\_sect\_count; /\* # of serializable sections in this bin \*/

 size\_t ghost\_sect\_count; /\* # of un-serializable sections in this bin \*/

 H5SL\_t \*bin\_list; /\* Skip list of differently sized sections \*/

} H5FS\_bin\_t;

1. Call the add callback if it exists. In this case, fspace->sect\_cls[sect->type]->add is NULL, so this is a NO-OP.
2. Check to see if the H5FS\_ADD\_RETURNED\_SPACE flag is set. In this case it is, so attempt to merge the returned section of memory with existing sections on the free list. Do this via a call to H5FS\_sect\_merge(fspace, &sect, op\_data)[[76]](#footnote-77). Note that the op\_data passed through here is the instance of H5MF\_sect\_ud\_t that was passed as the last parameter in the call to H5FS\_sect\_add(). The sect parameter points to the section describing the memory to be freed.

After some initial sanity checking, H5FS\_sec\_merge() proceeds by scanning through the skip list pointed to by space->sinfo->merge\_list for pieces of free memory that can be merged with the section described by the op\_data parameter. As the skip list has not yet been defined, this is a NO-OP, and we proceed with our examination of H5FS\_sect\_add().

1. Add the newly freed section to the free list. Do this via a call to H5FS\_sect\_link()[[77]](#footnote-78) as shown below:

H5FS\_sect\_link(fspace, sect, flags)

 After initial sanity checking, H5FS\_sect\_link() proceeds as follows:

* 1. Obtain a pointer to the instance of H5FS\_section\_class\_t associated with the type of the section to be added to the free list, and save this pointer in the cls local variable. Do this via the following lines of C code:

/\* Get section's class \*/

cls = &fspace->sect\_cls[sect->type];

Recall that in the context of file free space management, sect->type is initialized to H5MF\_FSPACE\_SECT\_SIMPLE[[78]](#footnote-79). As the H5MF\_FSPACE\_SECT\_SIMPLE macro is defined as follows:

#define H5MF\_FSPACE\_SECT\_SIMPLE 0

The above C code resolves to:

/\* Get section's class \*/

cls = &fspace->sect\_cls[0];

Thus, cls resolves to a pointer a copy of the instance of H5FS\_section\_class\_t that was passed into H5FD\_create() (see above).

* 1. Insert the newly freed section in the bin appropriate to its size. Do this via the following call:

H5FS\_sect\_link\_size(fspace->sinfo, cls, sect)

After initial sanity checking, H5FS\_sect\_link\_size()[[79]](#footnote-80) proceeds as follows:

* + 1. Determine which bin the newly freed section of file space should be filed in. Do this by computing the log base 2 of the size of the section via a call to H5V\_log2\_gen()[[80]](#footnote-81). Call this value n.
		2. Determine whether a skip list has been created for the target bin. Do this by testing to see if fspace->sinfo->bins[n].bin\_list is NULL. If no such skip list exists, create one for this bin via a call to H5SL\_create(H5SL\_TYPE\_HSIZE)[[81]](#footnote-82).
		3. Determine whether the skip list contains a list of entries of size equal to that of the newly freed section of file space. If such a list exists, its head will be an instance of H5FS\_node\_t[[82]](#footnote-83) stored in the skip list associated with the bin. Search the skip list via the call:

(H5FS\_node\_t \*)H5SL\_search(sinfo->bins[bin].bin\_list, &sect->size)

and store the result in the local variable fspace\_node.

If there is no such instance of H5FS\_node\_t, allocate one via a call to H5FL\_MALLOC(H5FS\_node\_t) and store the address in fspace\_node. Initialize the new instance as follows:

fspace\_node->sect\_size = sect->size;

fspace\_node->serial\_count = fspace\_node->ghost\_count = 0;

fspace\_node->sect\_list = H5SL\_create(H5SL\_TYPE\_HADDR);

Observe that after this initialization, fspace\_node->sect\_list points to an empty skip list.

Insert the new instance of H5FS\_node\_t into the section list in the target bin as follows:

 H5SL\_insert(sinfo->bins[n].bin\_list, fspace\_node,

 &fspace\_node->sect\_size)

and then increment fspace->sinfo->tot\_size\_count.

For reference, the definition of H5FS\_node\_t is reproduced below:

/\* Free space node for free space sections of the same size \*/

typedef struct H5FS\_node\_t {

 hsize\_t sect\_size; /\* Size of all sections on list \*/

 size\_t serial\_count; /\* # of serializable sections on list \*/

 size\_t ghost\_count; /\* # of un-serializable sections on list \*/

 H5SL\_t \*sect\_list; /\* Skip list to hold pointers to actual \*/

 /\* free list section node \*/

} H5FS\_node\_t;

* + 1. Insert the newly freed section into the appropriate bin. First increment the total section count as follows:

fspace->sinfo->bins[bin].tot\_sect\_count++;

Then increment the serialize-able section count[[83]](#footnote-84) as follows:

fspace\_node->serial\_count++;

If this is the first entry in \*fspace\_node, also increment the number of different size sections in the free list as follows:

fspace->sinfo->serial\_size\_count++;

Finally, insert the newly freed section of file space into the skip list attached to \*fspace\_node as follows:

H5SL\_insert(fspace\_node->sect\_list, sect, &sect->addr);

* 1. Update the rest of the free list manager data structures for the insertion. Do this via the following call:

H5FS\_sect\_link\_rest(fspace, cls, sect, flags)

After initial sanity checking, H5FS\_sect\_link\_rest()[[84]](#footnote-85) proceeds as follows:

* + 1. If the H5FS\_CLS\_SEPAR\_OBJ flag has not been set (which it hasn’t in this case), check to see if fspace->sinfo->merge\_list is NULL. If it is, create a new skip list via the call H5SL\_create(H5SL\_TYPE\_HADDR), and set fspace->sinfo->merge\_list equal to its base address.

Link the instance of H5FS\_section\_info\_t describing the newly freed section of file space into the skip list pointed to by fspace->sinfo->merge\_list via a call to H5SL\_insert(fspace->sinfo->merge\_list, sect, &sect->addr).

* + 1. Update section info and check to see if we need more room for the serialized free space sections via a call to H5FS\_sect\_increase(fspace, cls, flags).

This function appears to do a bunch of book keeping needed in preparation for writing a free list managers data structures to disk. I’ll bypass this for now, as it doesn’t seem central to the matter of the free list managers data structures.

* + 1. Add the size of the section being freed to the total amount of space in the free list as follows:

fspace->tot\_space += sect->size;

* 1. Return.
1. Call H5FS\_sinfo\_unlock(f, dxpl\_id, fspace, sinfo\_modified). H5FS\_sinfo\_unlock()[[85]](#footnote-86) is the inverse of the H5FS\_sinfo\_lock() call discussed above. Were the H5FS\_sinfo\_t in the metadata cache, this call would mark it dirty if appropriate, and then unprotect it. However, in this case, we created the instance of H5FS\_sinfo\_t in the call to H5FS\_sinfo\_lock() at the beginning of H5FS\_sect\_add(), so at least for purposes of examining the initialization of the free space manager data structures, this call is largely a NO-OP.

While H5MF\_xfree() continues from here, this appears to cover enough of the construction of the data structures used by the free space managers for current purposes. Needless to say, we should revisit this section if necessary.

### A Detailed Look at File Space Allocation in the Free List Managers

For purposes of this discussion, the entry point for file space allocation is H5MF\_alloc()[[86]](#footnote-87). As we have already examined this function in some detail in the contexts of the aggregators and initialization of the free list manager data structures, we will skip lightly over those sections of the code, and concentrate on the actual allocation of space by the free list managers.

After initial sanity checking, H5MF\_alloc() proceeds as follows when the file space strategy is H5F\_FILE\_SPACE\_ALL\_PERSIST:

1. Map memory type of request to free space type – call the result fs\_type.
2. Verify that the file space strategy is either H5F\_FILE\_SPACE\_ALL\_PERSIST or H5F\_FILE\_SPACE\_ALL. This is done via the H5F\_HAVE\_FREE\_SPACE\_MANAGER()[[87]](#footnote-88) macro.
3. Check to see if a free space manager has been defined for fs\_type. If it hasn’t, call H5MF\_alloc\_open()[[88]](#footnote-89). After some sanity checking and verifying that the target free space manager has not been initialized, this function calls H5FS\_open()[[89]](#footnote-90), and marks the free space manager as having been initialized.
4. Test to see if the free space manager contains a section of free space of size sufficient to satisfy the file space request. Do this via a call to H5FS\_sect\_find()[[90]](#footnote-91).

Omitting some book keeping, H5FS\_sect\_find() calls H5FS\_sinfo\_lock(), calls H5FS\_sect\_find\_node()[[91]](#footnote-92) in an attempt to find a suitable section of free file space, and then calls H5FS\_sinfo\_unlock().

H5FS\_sect\_find\_node() is a complex routine, that searches the free list manager data structure for the smallest piece of free file space that will satisfy the space request under the current alignment constraints. The essence of the algorithm is to map the request size to the appropriate bin, and thence to the instance of H5FS\_node\_t whose sect\_size field exceeds the request size by the smallest margin. If no such instance of H5FS\_node\_t exists, the search proceeds to the next largest bin. When so configured, alignment is also considered in selecting a candidate. If a suitable candidate section is found, it is removed from the free list manager data structures. If it contains excess space, the excess is clipped off and reinserted into the free list manager data structures.

1. If H5FS\_sect\_find() is successful, handle some book keeping and then use the section of memory found to satisfy the file space allocation request.
2. If H5FS\_sect\_find() is unsuccessful, the desired section of file space is allocated via a call to H5MF\_aggr\_vfd\_alloc() as discussed earlier in this document.

### A Detailed Look at File Space De-Allocation in the Free List Managers

As before, the entry point for file space de-allocation in the presence of the free list managers is H5MF\_xfree()[[92]](#footnote-93). As many of the details have already been explored in the above detailed exploration of the free list manager data structures, this exploration will skip over most data structure details.

The function proceeds as follows when the file space strategy is H5F\_FILE\_SPACE\_ALL\_PERSIST:

1. Perform a variety of initial sanity checks. Specifically:
	1. Verify that the address of the piece of file space to be freed is defined, and that its length is greater than zero.
	2. Verify that the address of the piece of space to be freed is not zero – the address of the super block.
	3. Verify that the file space to be freed does not intersect with the metadata accumulator. Recall that the metadata accumulator attempts to buffer adjacent metadata writes so that they can be combined in a single write operation.
	4. Verify that the file space to be freed is not “temporary” file space. Recall that temporary file space is file space beyond the end of the file that is assigned to temporary pieces of metadata that should never be written to file.
2. Map the allocation type of the file space to be freed (passed in via the alloc\_type parameter) to the free space type as discussed in section 2.3.3.1 above. Call the free space type fs\_type.
3. Check to see if a free space manager has been defined for fs\_type.

If it hasn’t, first check to see if the file space to be freed is at the end of file. If it is, it can be freed by simply reducing the EOF. If this fails, check to see if the file space to be freed can be put into either the metadata or small raw data aggregator for re-use. Note that either of these options allows us to avoid starting up a free space manager.

Test these conditions by calling H5MF\_try\_shrink()[[93]](#footnote-94), which performs the tests and does the free by adjusting the EOF or incorporating the freed file space into an aggregator if possible. If H5MF\_try\_shrink() succeeds, exit indicating success.

If H5MF\_try\_shrink() fails, check to see if the size of the file space to be freed is less than the fs\_threshold field of the files shared instance H5F\_file\_t[[94]](#footnote-95). If it is, drop the file space to be freed on the floor, and exit indicating success.

Finally, if all the above attempts to avoid setting up a free space manager fail, call H5MF\_alloc\_open()[[95]](#footnote-96) to setup the free space manager for the desired free space type. See the above exploration of the free list manager data structures for details.

1. If we get to this point, a free space manager exists to handle the piece of file space to be freed.

First, allocate an instance of H5MF\_free\_section\_t, and initialize it so that it describes the chunk of file space to be freed.

Then, if the size of the piece of file space to be freed is greater than or equal to the fs\_threshold, call H5FS\_sect\_add()[[96]](#footnote-97) to add the file space to be freed to the appropriate free list. Note that H5FS\_sec\_add() will attempt to merge the newly freed piece of file space with existing free space in the free list. Depending on whether we choose to allow pieces of metadata to span metadata aggregation blocks, this may have implications for this RFC, and may require further investigation.

After initial sanity checking, H5FS\_sect\_add() proceeds as follows:

* 1. Call H5FS\_sinfo\_lock()[[97]](#footnote-98).
	2. Call the add callback if it exists. In the case of file space, the callback does not exists, to this is a NO-OP.
	3. If the H5FS\_ADD\_RETURNED\_SPACE flag is set (which it is, in this case), attempt to merge the newly freed section of file space with an existing section of free file space. Do this via a call to H5FS\_sect\_merge()[[98]](#footnote-99). While H5FS\_sect\_merge() is a complex routine, conceptually, it does the following:
		1. Let A be the newly freed section.
		2. Find the entry in the merge list whose base address most closely precedes the base address of the newly freed section. Call this entry B.
		3. Find the entry in the merge list that occurs in the merge list just after B. Call this entry C.
		4. Determine whether A and B can merge. For the file space free lists, this is done by calling H5MF\_sect\_simple\_can\_merge()[[99]](#footnote-100). This function returns TRUE if A and B are adjacent. If they can, first remove B from the free list via a call to H5FS\_sect\_remove\_real()[[100]](#footnote-101), and then merge A into B. For the file space free lists, this is done via a call to H5MF\_sect\_simple\_merge()[[101]](#footnote-102). While this function merges A into B and then discards A, to simplify the remainder of this discussion, assume that B is merged into A and then discarded.
		5. Determine if A and C can merge. If they can, merge them as above.
		6. If possible, shrink the container. I don’t believe this applies to the file space free lists, but I mention it here, as it can result in A being discarded.
		7. Return the (possibly merged) newly freed section to the caller.

The above is a bit of an oversimplification. While it is probably good in the context of describing file free space management, the implementer will likely have to explore the container shrinking code in greater detail, as it may provide a mechanism for releasing small raw data or metadata aggregation blocks.

* 1. Assuming it still exists, insert the newly freed and possibly merged section of file space into the free list via a call to H5FS\_sect\_link()[[102]](#footnote-103). This function was discussed in some detail in the section on the free list manager data structures, so we will skip over it here other than to say that the function will (if successful) insert the newly freed section into both the bin appropriate to its size, and into the merge list.
	2. Call H5FS\_sinfo\_unlock()[[103]](#footnote-104).
1. Otherwise, if the size of the piece of file space to be freed is less than fs\_threshold, attempt to merge it with an existing piece of file space on the appropriate free list via a call to H5FS\_sect\_try\_merge()[[104]](#footnote-105). Again, depending on whether we choose to allow pieces of metadata to span metadata aggregation blocks, the particulars of this attempt may require further investigation. If the attempt to merge fails, just drop the free space on the floor.

After initial sanity checking, F5FS\_sect\_try\_merge() proceeds as follows:

* 1. Call H5FS\_sinfo\_lock()[[105]](#footnote-106).
	2. Call H5FS\_sect\_merge().
	3. If the newly freed section of file space has increased in size after the call to H5FS\_sect\_merge()[[106]](#footnote-107) (i.e. if one or more entries on the free list have been merged into it), insert the newly freed section of file space into free list via a call to H5FS\_sect\_link()[[107]](#footnote-108).
	4. Call H5FS\_sinfo\_unlock()[[108]](#footnote-109).
1. Free the instance of H5MF\_free\_section\_t allocated in 4 above if it still exists. This will happen if all attempts to insert it in the appropriate free list failed. Then return.

# Design Objectives and Tradeoffs

As discussed in the introduction, the principle design objective to be addressed in this RFC is aggregating small raw data and metadata space allocations into pages suitable for caching by a level-2 page buffer (discussed in the companion RFC). To facilitate efficient I/O, it must be possible to align the pages on boundaries congruent with the underlying file system block size. Finally, it would simplify matters greatly if we could infer the base address and length of the page containing a piece of metadata from the base address and length of the piece of metadata in questions. If we are unable to do this, we will have to maintain an index to perform this mapping operation.

While this objective is clear enough, it raises a number of other questions that must be answered before we can settle on a final design. For the most part, these questions are tightly coupled – thus I shall outline the questions first, and not offer my proposed answers until the end of this section.

## Do We Need to Support the Split and Multi File Drivers?

The entire purpose of small raw data and metadata aggregation in a fashion that allows the use of a level-2 page cache is to get rid of the small I/O operation that the split and multi file drivers are intended to address. Thus, by this argument, there is no reason to support the split and multi file drivers.

On the other hand, from a conceptual perspective at least, the choice of file driver and file space allocation strategy should be orthogonal. Thus, by this argument, we should support all file drivers.

## Do We Allow Small Pieces of Raw Data or Metadata to Cross Aggregator Block Boundaries?

Given that a space allocation request is smaller than an aggregator block, in principal, we have a choice as to whether we should allow a small piece of raw data or metadata to span the boundary between two adjacent aggregator blocks.

Such allocations will complicate the management of the level-2 page cache, but would allow more efficient space allocation.

On the other hand, preventing such allocations will require either changes in the free list code to prevent it, or allocation of markers at the beginning (and possibly the end) of aggregator blocks to keep the free list code from merging them. Note that this latter solution need not be a file format change – if we write nothing to the markers, they will simply look like waste space.

More to the point, large pieces of metadata will have to cross aggregator block boundaries, so we probably need the supporting code regardless.

## How Do We Allocate Large Pieces of Raw Data?

The decision to aggregate small raw data allocation into aggregator blocks leads inevitably to the question of how we will allocate large[[109]](#footnote-110) pieces of raw data. Since the major motivation behind small raw data and metadata aggregation is to facilitate the function of a level-2 page cache, we must consider its function in selecting our solution, even though the page cache is the subject of another RFC.

This leads us to consider the how the page cache would be used with large pieces of raw data. In particular, it is worth noting that we may or may not want to use it for large pieces of metadata depending on circumstances.

For example, individual chunks in a chunked dataset will always be written and read as atomic entities – and thus (assuming the chunk size exceeds the aggregator block size), it makes sense to bypass the level-2 page cache when accessing them – particularly since we already maintain a chunk cache.

In contrast, contiguous datasets are frequently too large to be read and written all at once. Further, we may be called upon to make many small reads and/or writes throughout a contiguous dataset when performing I/O on small selections within the dataset. Thus in this case at least, it makes sense to employ the page cache where practical to aggregate the small reads and writes into page reads and writes to the extent possible.

In essence, this leaves us two basic design options.

On the one hand, we can satisfy large raw data file space allocations by allocating sufficient adjacent aggregator blocks, inserting them in the free list, allowing the free list manager to combine the blocks, and then satisfy the request with the resulting large chunk of free space.

On the other hand, we can ignore the aggregator model completely for large raw data file space allocation requests, and simply proceed more or less as we do now – satisfying such requests out of existing free space if available, and extending the file if necessary.

It should be noted that the latter option allows us to cheat a bit. If aggregator blocks all have size equal to their alignment, and if both raw data and metadata aggregator blocks are of the same size, they will implicitly define a paging scheme for the entire file – allowing us to page in sections of raw data when desirable.

## How Do We Allocate Large Pieces of Metadata?

The question of how to allocate large[[110]](#footnote-111) pieces of metadata is similar to that of how to allocate large pieces of raw with a few key differences.

1. Large pieces of metadata are rare.
2. Unlike raw data, large pieces of metadata will always be written and read as atomic entities.

At first glance, these two observations would suggest that the best solution is to allocate large pieces of metadata independently from the aggregator system as per the second option for large pieces of raw data. However, this neglects one consideration.

At present, finding all the metadata in an HDF5 file is difficult, as it requires walking the file. However, if we aggregate all metadata into metadata aggregator blocks, the problem becomes much easier – we can put a header on each metadata aggregator block and link them together, or maintain an index, or whatever. However, if we allocate large pieces of metadata outside the aggregator block system, the metadata aggregation block header solution becomes un-workable. The index solution remains do-able, albeit more complex, as the index must handle both metadata aggregation blocks and large pieces of metadata proper.

## Do We Want to be Able to Infer Metadata Aggregator Block Base Address and Length from Metadata Base Address and Length?

Clearly the answer to this question is yes. However there are tradeoffs that we explore below:

If we require that metadata aggregator block alignment and length are equal[[111]](#footnote-112), we can infer the base address and length of a metadata aggregator block from the base address of any piece of metadata it contains.

Further, even if we allow metadata entries to cross metadata aggregator block boundaries, the combination of the metadata aggregator block alignment and size, and the base address and length of a piece of metadata allows us to infer the base addresses and lengths of all metadata aggregator blocks in which the requested metadata entry resides.

Maintaining the ability to infer metadata aggregator block address and length from metadata entry base address and length allows us to avoid constructing a map of metadata aggregator blocks, which would otherwise be required to map metadata read/write requests to the containing metadata aggregator blocks.

In closing we should note that this is possible because metadata is always written and read as an atomic entity. We don’t have this with raw data – for instance, we may do small writes and reads inside a contiguous data set. This shouldn’t pose any particular problems as long as we either allocate all raw data within raw data aggregators, or use the implicit paging scheme discussed at the end of section 3.3 above.

## Do We Want to be Able to Map all the Metadata Aggregator Blocks in the File?

While we may be able to cache metadata aggregator blocks without a map of same, the ability to map metadata aggregator blocks easily may still be desirable.

A small header at the beginning of each metadata aggregator block containing a link to the next metadata aggregator block would allow us to walk the list of metadata aggregator blocks in the HDF5 file. This in turn would allow us to locate all metadata aggregator blocks in the file cheaply.

If, in addition, we require that all metadata reside in metadata aggregator blocks, we would be able to locate all metadata in the file cheaply.

An alternate solution would be to maintain an index of all metadata aggregator blocks (and possibly large pieces of metadata) in the file.

Either solution opens the possibility of stripping all the metadata out of a file to, and putting it back again if we handle space allocation correctly. In particular, this seems to have some application to the metadata server concepts currently being developed for LBNL.

## How do We Handle Interactions with SWMR?

Handling the potential for destructive interference between the page cache and SWMR (Single Reader Multiple Writer) is an issue that belongs more in the companion Page Cache RFC than in this RFC. However, it does have some implications that should be mentioned here.

Conceptually, SWMR requires that metadata associated with any single HDF5 file object be flushed in priority order. Thus all metadata of SWMR priority n (for the given HDF5 file object) must be flushed before any metadata of SWMR priority n – 1.

In general, no HDF5 file object will generate enough metadata of any particular SWMR priority to fill a metadata aggregator block, and in any case, we will want to hide the particulars of individual HDF5 objects from the metadata aggregators, the free list, and the page cache. Thus it will probably be necessary to assign a SWMR priority to metadata aggregator blocks and free list entries, along with the free list file space type. If we do this, and allocate metadata associated with HDF5 file objects accordingly, we will be able to meet SWMR write requirements by writing all metadata aggregator blocks of SWMR priority n before writing any such blocks of SWMR priority n – 1.

While the above sounds good, there are several flies in the ointment.

First, SWMR priority is context sensitive – if we are applying SWMR to a group, we might require that the associated heap be flushed before the B-tree, and that the B-tree be flushed before the header. Thus, for instance, elements of identical fractal heaps might have different SWMR priorities depending on how the fractal heap is employed. Assuming we go this route, we will have to compute these relative SWMR priorities at metadata allocation time.

To make matters worse, as Quincey has pointed out, we use some data structures that grow from the top (i.e. B-trees that grow by splitting the root node and adding a new node “above” it) and others that grow from the bottom (i.e. fractal heaps which get bigger by adding nodes on the bottom). While I think this circle can be squared, (say by leaving several priority levels for a B-tree to grow into, or possibly by allowing creation of new priority levels between existing levels.), the exercise will be painful and quite possibly not worth the effort.

Second, all this multiplies the maximum number of free lists by the highest possible SWMR write priority. Fortunately this number should be relatively low – although the above computation of relative SWMR priorities will increase it significantly.

Third, the page cache will need to know the SWMR priority of any page it loads. The obvious way to do this will be to place it in a metadata aggregator block header. This, of course, requires a file format change.

The bottom line is that if we want to avoid disabling SWMR whenever the metadata aggregator and the page cache are enabled, we must modify the free list and space allocation calls to accept a SWMR priority along with an allocation type, modify the existing metadata allocation code to compute and SWMR priorities, and we must modify the free list code to maintain free lists for all free list file space type / SWMR priority combinations currently in use.

As Quincey has suggested, an alternate solution is to convert to a write through page cache when SWMR is enabled. I can’t think of any reason why this wouldn’t work, but I don’t see it as buying us much either.

## How do We Handle Interactions with Metadata Journaling?

Another potential problem for the metadata aggregator / page cache combination is how to meet the write ordering requirements of metadata journaling, which basically require that metadata entries may be written to disk if and only if the associated journal entry has made it to disk.

Unlike the SWMR case, the journaling case makes it much harder to aggregate metadata entries that must be written together. The only reasonably simple solution that comes to mind is to periodically flush the journal file, then flush the metadata cache, and finally flush all dirty metadata aggregator blocks in the page cache.

Quincey has suggested moving to a write through page cache in this case as well. Again, the solution seems workable to me, but not particularly useful. That said, it is an easy solution to the problem that should buy us a little, albeit at the cost of modifying the page cache.

Other more complex solutions are possible, but they will result in increased metadata writes.

## Selected Tradeoffs

As should be evident from the above discussion, the tradeoffs involved in the design of the small raw data and metadata aggregator blocks and associated free list modifications are tightly coupled. In the following table, I present a summary of my preferred choice of tradeoffs.

While I will argue for this particular selection shortly, the main point here is that the selection of trade offs has a major effect on the lower level design details. Any change in this selection will have to be reflected in the remainder of this document, the associated Page Cache RFC, and the associated implementations.

|  |  |
| --- | --- |
| Design Consideration: | Trade Off Decision: |
| Do We Need to Support the Split and Multi File Drivers? | Yes – If it isn’t too painful.  |
| Do We Allow Small Pieces of Raw Data or Metadata to Cross Aggregator Block Boundaries? | YES – Assuming we don’t go with a write through cache, we will probably need to reverse this for SWMR. |
| How Do We Allocate Large Pieces of Raw Data? | Independently, with implied paging scheme. Note that this implies that raw and metadata aggregator blocks must be the same size, and that aggregator block size and alignment must be equal. |
| How Do We Allocate Large Pieces of Metadata? | Independently. Retain the option of adding large pieces of metadata to index should this be desired. |
| Require ability to infer metadata aggregator block address and length from metadata base address and length? | YES – Note that this implies that metadata aggregator block size and alignment must be the same. |
| Require ability to easily map all the metadata aggregator blocks (and therefore all metadata) in the file? | NO – But keep the option open. It may be needed for metadata server contemplated for LBNL. If we do it, we will probably construct an index. |
| How do We Handle Interactions with SWMR? | Turn off page cache when SWMR is enabled for now. Examine practicality of assigning SWMR priority levels at metadata allocation time. Failing that, consider shifting to a write through cache. |
| How do We Handle Interactions with Metadata Journaling? | Turn off the page cache when metadata journaling is enabled for now. Consider shifting to a write through cache. |

In my judgment, the two critical design considerations are the ability to infer metadata aggregator block address and length from metadata base address and length, and how we will allocate large pieces of raw data and metadata.

The ability to infer metadata aggregator block address and length from metadata base address and length avoids the necessity of maintaining a metadata aggregator block map, and thus greatly simplifies the level-2 page cache. I think we should require this ability, which in turn requires metadata aggregator block alignment and length to be equal.

The fact that elements of raw data need not be written and read as atomic entities forces us to either allocate all raw data within the raw data aggregator system, or require an implicit division of the HDF5 file into pages of equal length. As we will wish to bypass the level-2 page cache is some cases (i.e. chunk writes and reads), I think we should allocate large pieces of raw data independently. This requires that the HDF5 file be implicitly divided into pages, which in turn requires that the raw data and metadata aggregators have identical length and alignment.

Since large pieces of raw data will span implicit page boundaries, there is no reason to require that small pieces of raw data or metadata not span aggregator block boundaries. This in turn makes aggregator block headers less attractive (although they may still be needed for SWMR). Hence there is no reason not to allocate large pieces of metadata independently as well.

Note that this means that if we ever need to maintain a map of metadata in an HDF5 file, we will probably have to use the index approach.

The question of whether to support the split and multi file drivers is somewhat orthogonal to the above. Quincey has solved that issue by fiat, as he has indicated that he wants to depreciate those file drivers in their current forms.

While on the subject of fiats, I should also note that he has indicated that the new aggregator functionality should replace the old.

This leaves only the problems of interactions with SWMR and metadata journaling. In both cases, the problems are nasty. Thus the initial solution is to turn off the level-2 page cache when these features are enabled. That said, particularly in the case of SWMR, we might be able to come up with a reasonable solution. We should try to leave in hooks to facilitate their implementation when and if we are prepared to make the necessary effort.

# Conceptual Overview of Proposed Solution

Before embarking on a discussion of the specifics of the proposed solution, an overview of the revised system for file space allocation and re-use should be useful – both as a target for criticism and as a conceptual frame work on which to hang the specifics. To add some structure to this overview, let us start at the bottom and work up.

## The Allocator of Last Resort

Since HDF5 manages files, in theory the file space available is limited only by the address space. Thus, at the lowest level, the allocator of last resort allocates space by increasing the end of file sufficiently to service the request, and returning the base address of the new file space to the caller.

## The Large Allocation Free List Manager

At the next level up, I propose a free list[[112]](#footnote-113) for large file space allocations – that is to say, allocations of size greater than or equal to the aggregator block size. As presently envisioned, it would handle space requests of all memory types, and would only service space allocation/de-allocation requests for blocks of size greater than or equal to the aggregator block size.

As with the current free list managers, it would retain the ability to enforce alignment requirements on the blocks of file space used to satisfy file space requests. The large allocation free list manager would perform very much like the current free list managers, accepting de-allocated file space to be placed on the free list, combining adjacent pieces of free space, and satisfying space requests either from the free list, or by requesting space from the allocator of last resort, and satisfying requests from the space so obtained.

## The Small Allocation Free List Managers

At the top level, I propose two free list managers for small[[113]](#footnote-114) data allocations – one for raw data, and one for metadata[[114]](#footnote-115). These free list managers would service all space requests of size less than the aggregation block size. They would service these requests out of their free lists, or failing that, out of aggregator blocks obtained from the large allocation free list manager. If an entire aggregator block becomes free, it will be returned to the large allocation free list manager.

As with the current free list managers, the small allocation free list managers would accept small space de-allocations of the appropriate type, place then on their free lists, and combine adjacent free space. Unlike current free list managers, they would ignore alignment requests, as the level-2 page cache will be expected to swap in and out the aligned aggregator blocks.

## API and File Format Implications

To support the above, we will need to specify the aggregator block size and alignment at file creation time, and store this value in a superblock message. We will also need to be able to detect when the file has been modified by an earlier version of HDF5 that does not support the new file space allocation scheme.

If we implement the implied paging system made possible by setting the aggregation block size equal to the alignment, and allow small pieces of metadata and raw data to cross aggregation block boundaries, we shouldn’t have to disable the page cache in this situation, although we would have to force all reads and writes to look first at the page cache, as there would be no guarantee that a small piece of metadata, a small piece of raw data, and parts of large pieces of raw data and metadata might not reside in parts of the same page. This should not cause any hardship with contiguous data sets, or with small pieces of raw data or metadata. It may be a bit of a bother with large chunks of raw data.

In any case, this situation will break any indexing scheme to find all metadata – a point we should make note of.

# Proposed Aggregator and Free List Revisions

Having chosen our objectives, examined the current space allocation and de-allocation code, and outlined a proposed solution at the conceptual level, it is now time to consider the details.

## Proposed API Additions

Given our requirement that we be able to infer the base address and length of the containing metadata aggregation block(s) from the base address and length of a piece of metadata, the size and alignment of both metadata and raw data aggregation blocks must all be equal. This allows us to both enable and configure metadata and raw data aggregation with one value – the aggregator block size (an aggregator block size of 0 will turn off aggregation). Per our discussion above, this will have to be done at file creation time, as aggregator block size and alignment must be fixed for the life of the file.

This in turn can be done with the pair of File Creation Property List (FCPL) routines, which get and set the aggregator size. The proposed signatures are given below:

*herr\_t* H5Pset\_aggregator\_block\_size( *hid\_t* fcpl\_id, *hsize\_t* size )

*herr\_t* H5Pget\_aggregator\_block\_size( *hid\_t* fcpl\_id, *hsize\_t\** size\_ptr )

As should be expected, these routines will set and get the aggregator block size stored in the target FCPL. When the FCPL is used to create a file, the aggregator block size will be stored in a super block message, and will remain unchanged for the life of the file.

As the new metadata and small raw data aggregation scheme will likely be enabled by default, we will need to specify a default value for the aggregator block size. While this need not be done now, it does seem appropriate to suggest that we use different values for serial and parallel compiles – say 64 KB and 1 MB.

Note that H5Pget/set\_aggregator\_block\_size() will supersede the existing H5Pget/set\_meta\_block\_size() and H5Pget/set\_small\_data\_block\_size() API calls. While I expect that these calls will remain in the library for some time for compatibility reasons, they will become no-ops.

At first glance, it might seem that we also need API calls to control the number and type assignments of the large and small allocation free lists. However, on reflection, it should be apparent that these values are implicit in the choice of VFD. In all cases other than the multi file driver we will need one large allocation list and two small allocation lists. In the case of the multi file driver, we will need one large and one small allocation list for each sub-file, and the number of sub-files and their type allocations will not change in the life of the file.[[115]](#footnote-116)

While this is more a matter for the companion page cache RFC, it may be useful to observe that at least for the first cut, any adjustments for SWMR or metadata journaling can be controlled by the switches controlling these functions. Hence in this case as well, it seems that the above pair of property list calls are all we need to control the new metadata and small raw data aggregation scheme.

## Proposed Functional Changes to the Aggregator Code

As intimated in the conceptual overview, the existing small raw data and metadata aggregator code will be removed, and be replaced by the small allocation free list managers. While there will be some higher level changes needed in the free space manager code, in this section I consider only the changes needed to allow the free space manager code to handle small raw data and metadata aggregation.

In this role, the small allocation free list managers will function much like the existing free list managers, with the following deltas:

1. They will only service file space requests of size less than the metadata aggregator block size.
2. Alignment constraints will be ignored when servicing requests, as we will assume that aggregator blocks will be swapped in and out whole. /\* we probably still want to enforce word alignment in allocations. \*/
3. When in need of space to allocate, the small allocation free list managers will obtain space to allocate from the appropriate large allocation free list. This space will be in the form of aggregation blocks of size and alignment equal to the aggregation block size. Thus, unlike the current free list managers, space requests should never fail unless the EOF cannot be extended for whatever reason.
4. Space released to the small allocation free list managers will be combined where possible. If the list ever contains a block of free space that contains a properly aligned aggregation block, that block should be released to the appropriate large allocation free list manager. Note that any space before or after the block will be snipped off before it is released.

For the present at least, the plan is to allow small pieces of metadata (and also small pieces of raw data) to cross aggregator block boundaries. If we choose to disallow this, we will need some mechanism for enforcing the policy.

The easiest solution will be to allocate a divider block at the beginning of each aggregator block when the block is allocated to the small allocation free list manager. This allocation by its existence would keep free space in adjacent aggregator blocks of the same free space type from being merged. Since we don’t have to write anything in the divider blocks, no additional file format change is required. For efficiency reasons, we would probably want to make the divider block at least one word long, which would necessitate changing the request size dividing point between the large and small allocation free list managers accordingly.

Alternatively, we could modify the free list code to be aware of aggregation blocks and their boundaries, and to avoid crossing them when allocating small pieces of metadata or raw data. I would expect this to be the more painful solution, but it should be doable.

## Proposed Functional Changes to the Free List Code

The large allocation free list managers should behave exactly as the free list managers do at present, save that they should obtain space by extending the EOF if a suitable free block is not available. Thus in this case as well, space requests should never fail unless the EOF cannot be extended.

Thus the major functional changes in the free list code proper will be in the higher levels of the free list manager code, and will be directed at the following issues:

1. Setting up the additional free lists – typically one for large allocations of all memory types and two for small allocations (one for raw data, and one for metadata of all types).
2. Modification of the top level space allocation code to route space requests between the free list managers according to both memory type and request size, instead of routing space requests based purely on request type, as is done at present.

I discuss these issues in the following subsections.

### Managing Additional Free Lists

The obvious way of handling this issue is to duplicate the existing fs\_man, fs\_map, and fs\_state fields in H5F\_file\_t for the small free lists – say sfs\_man, sfs\_map, and sfs\_state. We could then use either the fs\_ or sfs\_ version of the arrays depending on the size of the file space allocation or de-allocation request exceeds the aggregation block size. If aggregation block size is zero, all requests would be routed to the large free space manager – which is exactly the desired behavior.

### Space Request Routing

The simple solution seems to be to modify H5MF\_alloc() to use the request size to choose between either the small and large free space managers prior to mapping the space request type to free space type, and then mapping free space type to the appropriate free list manager. This change (along with its cognate in H5MF\_xfree()) should be quite trivial, and completely localized.

## Proposed File Format Changes

Looking at the file format specification, it seems that file free space is to be stored in a superblock message, which then points to the tables containing the contents of the various free lists.

While I haven’t located the code that handles saving and reloading file space free lists[[116]](#footnote-117), I gather that this functionality is implemented in the trunk, but not in the 1.8 branch. As such, it doesn’t exist in any release version of HDF5, which means that we can modify the current approach without introducing any new backward/forward compatibility issues.

Given this, I am inclined to extend the current file space free list superblock message to include both the small and large free lists. It probably makes sense to include the aggregator block size in this message as well. The message should be flagged so as to require any version of the library that doesn’t understand it to make note of the fact if it opens the file read/write.

## Proposed Modifications to Tools

As best I can tell, none required.

# Backward/Forward Compatibility Issues

As best I can tell, the only backward/forward compatibility issue raised by the small raw data / metadata aggregation modifications proposed in this RFC is the possibility of a round trip from a version that understands the modifications to one that doesn’t and back again.

In this scenario, it will be possible for large and small pieces of raw data and metadata to become intermingled in a single aggregator block. In such cases, we will have to force all reads and writes through the page buffer – assuming we don’t choose to do so regardless. However, this is an issue for the companion page buffer RFC.

Perhaps more importantly, this scenario will break any index of metadata that we may have constructed.

All of these issues should be quite manageable as long as we know that they exist.

# Recommendation

With the exception of specifying the structure of the superblock message storing the contents of the large and small allocation free lists, this RFC seems to be reasonably mature. Assuming that readers of this version find no major issues, I believe it is time to put this document to one side for a while, and concentrate on the companion page buffer RFC.

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

|  |  |
| --- | --- |
| *July 26, 2011* | Version 0 – Initial outline, and first cut at section 1. |
| *October 14, 2011:* | Version 1 – First cut at sections 2, and 3. Sent to Quincey for comments on Section 3 (Design Considerations and Tradeoffs).  |
| *November 11, 2011* | Version 2 – Title change and general re-work to reflect expanded mandate. Added section 4.  |
| *January 9, 2012* | Version 3 – Re-worked to reflect Quincey’s most recent set of comments. Added investigation of the free list managers and their data structures in section 3, as it was needed to address some design issues intelligently. Added sections 5, 6, and 7. |

1. Defined in H5MF.c. [↑](#footnote-ref-2)
2. Defined in H5MF.c. [↑](#footnote-ref-3)
3. Defined in H5MF.c. [↑](#footnote-ref-4)
4. Defined in H5MF.c. [↑](#footnote-ref-5)
5. See definition of H5F\_file\_space\_type\_t in H5Fpublic.h. [↑](#footnote-ref-6)
6. And therefore the H5F\_FILE\_SPACE\_VFD case is not covered. [↑](#footnote-ref-7)
7. alloc\_size is a value passed in at file creation/open time via a property list, defaulting to 2048. [↑](#footnote-ref-8)
8. Deriving algorithms from code is a tricky process, particularly if the code has been highly hand optimized – as is the case here. While I probably have the general thrust correct, I am sure that errors have crept in. If you find an error, please point it out. Perhaps more importantly, I have made no attempt to divine the reasoning behind the algorithm. [↑](#footnote-ref-9)
9. Defined in H5Fpkg.h. [↑](#footnote-ref-10)
10. Defined in H5MFaggr.c. [↑](#footnote-ref-11)
11. Defined in H5MFaggr.c. [↑](#footnote-ref-12)
12. Defined in H5Fpkg.h. The instances of H5F\_blk\_aggt\_t used to maintain the small raw data and metadata aggregator for a file are stored in the sdata\_aggr, and meta\_aggr fields of the H5F\_file\_t structure. The instance of H5F\_file\_t associated with an HDF5 file is pointed to by the shared field of the H5F\_t structure, which is the main structure for managing HDF5 files. All these structures are defined in H5Fpkg.h [↑](#footnote-ref-13)
13. Defined in H5F.c. [↑](#footnote-ref-14)
14. Defined in H5FDspace.c. [↑](#footnote-ref-15)
15. Defined in H5FDspace.c. [↑](#footnote-ref-16)
16. Defined in H5FDspace.c. [↑](#footnote-ref-17)
17. Defined in H5MF.c. [↑](#footnote-ref-18)
18. Defined in H5MF.c. [↑](#footnote-ref-19)
19. Defined in H5Fpkg.h. [↑](#footnote-ref-20)
20. Defined in H5MF.c. [↑](#footnote-ref-21)
21. Defined in H5Pfcpl.c. [↑](#footnote-ref-22)
22. Defined in H5FSsection.c. [↑](#footnote-ref-23)
23. Defined in H5FSsection.c. [↑](#footnote-ref-24)
24. Defined in H5Fpkg.h. [↑](#footnote-ref-25)
25. Defined in H5MF.c. [↑](#footnote-ref-26)
26. Defined in H5FS.c. [↑](#footnote-ref-27)
27. Defined in H5FSsection.c. [↑](#footnote-ref-28)
28. Defined in H5FSsection.c. [↑](#footnote-ref-29)
29. Defined in H5FSpkg.h. [↑](#footnote-ref-30)
30. Defined in H5FSprivate.h. [↑](#footnote-ref-31)
31. Defined in H5MFsection.c. [↑](#footnote-ref-32)
32. Defined in H5FS\_private.h. [↑](#footnote-ref-33)
33. The type of a section may not be relevant in free list managers for HDF5 file space. [↑](#footnote-ref-34)
34. I’m not sure exactly what this field is used for. At a guess, it has something to do with saving free space data to file and reloading it – but the reader is warned that this is just a guess. [↑](#footnote-ref-35)
35. Defined in H5FSpkg.h. [↑](#footnote-ref-36)
36. Defined in H5FSpkg.h. [↑](#footnote-ref-37)
37. Defined in H5FSpkg.h. [↑](#footnote-ref-38)
38. Defined in H5Fpublic.h [↑](#footnote-ref-39)
39. Defined in H5MF.c. [↑](#footnote-ref-40)
40. Defined in H5Fpkg.h. [↑](#footnote-ref-41)
41. Actually, there is some question in my mind about this. If a file is opened twice with different file drivers, it would seem that HDF5 would have to create two distinct instances of H5F\_file\_t since the file driver data is stored in that structure. Given the huge potential for file corruption, I would think that we would disallow this maneuver. I don’t know whether we do so or not – as it is tangential to the matters at hand, I’ll pass this issue on to Quincey. [↑](#footnote-ref-42)
42. Defined in H5Fpkg.h. [↑](#footnote-ref-43)
43. Defined in H5FD.c. [↑](#footnote-ref-44)
44. Defined in H5F.c. [↑](#footnote-ref-45)
45. Defined in H5FDpublic.h. Maps all memory types to a single free list type. [↑](#footnote-ref-46)
46. Defined in H5FDpublic.h. Maps each memory type to its own free list type. [↑](#footnote-ref-47)
47. Typically, this will happen in a call to H5MF\_xfree() – defined in H5MF.c. [↑](#footnote-ref-48)
48. Defined in H5MF.c. [↑](#footnote-ref-49)
49. Defined in H5MF.c. [↑](#footnote-ref-50)
50. Defined in H5FSprivate.h. [↑](#footnote-ref-51)
51. Defined in H5FS.c [↑](#footnote-ref-52)
52. Defined in H5MFsection.c [↑](#footnote-ref-53)
53. Defined in H5FS.c. [↑](#footnote-ref-54)
54. Defined in H5FSpkg.h. [↑](#footnote-ref-55)
55. Defined in H5FLprivate.h. [↑](#footnote-ref-56)
56. Defined in H5FLprivate.h. [↑](#footnote-ref-57)
57. Defined in H5FLprivate.h. [↑](#footnote-ref-58)
58. Defined in H5FL.c. [↑](#footnote-ref-59)
59. Defined in H5FL.c. [↑](#footnote-ref-60)
60. Defined in H5FLprivate.h [↑](#footnote-ref-61)
61. Defined in H5MFsection.c. [↑](#footnote-ref-62)
62. Defined in H5MFpkg.h. [↑](#footnote-ref-63)
63. Defined in H5FLprivate.h. [↑](#footnote-ref-64)
64. Defined in H5FLprivate.h. [↑](#footnote-ref-65)
65. Defined in H5FL.c. [↑](#footnote-ref-66)
66. Defined in H5FLprivate.h. [↑](#footnote-ref-67)
67. Defined in H5MFpkg.h. [↑](#footnote-ref-68)
68. Defined in H5FSsection.c. [↑](#footnote-ref-69)
69. Defined in H5FSsection.c. [↑](#footnote-ref-70)
70. Defined in H5FSsection.c. [↑](#footnote-ref-71)
71. Defined in H5FSpkg.h. [↑](#footnote-ref-72)
72. Defined in H5Vprivate.h. [↑](#footnote-ref-73)
73. Defined in H5Vprivate.h. [↑](#footnote-ref-74)
74. Defined in H5FSpkg.h. [↑](#footnote-ref-75)
75. Defined in H5FSpkg.h. [↑](#footnote-ref-76)
76. Defined in H5FSsection.c. [↑](#footnote-ref-77)
77. Defined in H5FSsection.c. [↑](#footnote-ref-78)
78. Defined in H5MFpkg.h [↑](#footnote-ref-79)
79. Defined in H5FSsection.c. [↑](#footnote-ref-80)
80. Defined in H5Vprivate.h. [↑](#footnote-ref-81)
81. Defined in H5SL.c. [↑](#footnote-ref-82)
82. Defined in H5FSpkg.h. [↑](#footnote-ref-83)
83. The H5FS\_node\_t data structure also maintains a “ghost\_count” field, which (according to the comments) is for “un-serializable” entries on the list. To date, I haven’t been able to figure out what this is about. As I haven’t run into it yet, it may not be relevant to the file free space role of the free list managers. That said, it is something to keep any eye out for. [↑](#footnote-ref-84)
84. Defined in H5FSsection.c. [↑](#footnote-ref-85)
85. Defined in H5FSsection.c. [↑](#footnote-ref-86)
86. Defined in H5MF.c. [↑](#footnote-ref-87)
87. Defined in H5Fpkg.h. [↑](#footnote-ref-88)
88. Defined in H5MF.c. [↑](#footnote-ref-89)
89. Defined in H5FS.c. [↑](#footnote-ref-90)
90. Defined in H5FSsection.c. [↑](#footnote-ref-91)
91. Defined in H5FSsection.c. [↑](#footnote-ref-92)
92. Defined in H5MF.c. [↑](#footnote-ref-93)
93. Defined in H5MF.c. [↑](#footnote-ref-94)
94. Defined in H5Fpkg.h. [↑](#footnote-ref-95)
95. Defined in H5MF.c. [↑](#footnote-ref-96)
96. Defined in H5FSsection.c. [↑](#footnote-ref-97)
97. Defined in H5FSsection.c. [↑](#footnote-ref-98)
98. Defined in H5FSsection.c. [↑](#footnote-ref-99)
99. Defined in H5MFsection.c. [↑](#footnote-ref-100)
100. Defined in H5FSsection.c. [↑](#footnote-ref-101)
101. Defined in H5MFsection.c. [↑](#footnote-ref-102)
102. Defined in H5FSsection.c. [↑](#footnote-ref-103)
103. Defined in H5FSsection.c. [↑](#footnote-ref-104)
104. Defined in H5FSsection.c. [↑](#footnote-ref-105)
105. Defined in H5FSsection.c. [↑](#footnote-ref-106)
106. In the context of file space free lists, I believe the section must still exist at this point. However, given my limited understanding of the code, best I not say so absolute authority. [↑](#footnote-ref-107)
107. Defined in H5FSsection.c. [↑](#footnote-ref-108)
108. Defined in H5FSsection.c. [↑](#footnote-ref-109)
109. For present purposes, a piece of raw data is “large” if it exceeds the size of an aggregator block. [↑](#footnote-ref-110)
110. For present purposes, a piece of metadata is “large” if it exceeds the size of an aggregator block. [↑](#footnote-ref-111)
111. If the aggregator block size is less than the alignment, we waste space between blocks. If it is greater, we can no longer infer aggregator block address and length from metadata address and length. [↑](#footnote-ref-112)
112. Or possibly several large data free lists – the free list managers will support either. However, with the possible exception of the parallel case, we will not need more than one unless we change our minds about not supporting the split and multi file drivers. [↑](#footnote-ref-113)
113. For current purposes, a space allocation request is small if it is less than or equal to the aggregator block size. [↑](#footnote-ref-114)
114. If we choose to support the multi file driver, the number of small allocation free list managers will also have to increase. [↑](#footnote-ref-115)
115. Unless we grab an image of the file and use the core file driver to open it. As H5Fget\_file\_image() doesn’t work on files opened with the multi file driver, and as we have no plans to ever support the multi file driver in H5Fget\_file\_image(), this is a bit hypothetical. However, it seems prudent to point out the potential issue. [↑](#footnote-ref-116)
116. In truth, I haven’t tried very hard. [↑](#footnote-ref-117)