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() and de-allocated via calls to H5MF_xfree(). This may be a bit of an oversimplification, as other routines may be used for the purposes (H5MF_try_extend() and possibly H5MF_alloc_tmp() 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 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.

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 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. 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, 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.
    a. 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.
    b. 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.
3. 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.
4. 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.
    a. 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.
    b. 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()`, which, in the case of a metadata space allocation request, in turn calls `H5MF_aggr_alloc()` 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:

```
/* 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() 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()`.

     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

) 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()`.

- 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()`. 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()`.

4. 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().

5. 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:
    a. Verify that the address of the piece of file space to be freed is defined, and that its length is greater than zero.
    b. Verify that the address of the piece of space to be freed is not zero – the address of the super block.
    c. 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.
    d. 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()`, 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. 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()` 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()`, and thus appears to be arbitrary.

4. 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()` 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()`. 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.
   a. 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.
   b. 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()` macro.

3. Check to see if a free space manager has been defined for fs_type. If it hasn't, call `H5MF_alloc_open()` 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()` 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()`. 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()` 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()`.

5. 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.
6. 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`. 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`. 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`.

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. This structure also contains fields for the section type and state.

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. 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 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, 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. 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, 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` 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.

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 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()` shortly after the instance of H5F_file_t is allocated in `H5F_new()`. 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
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

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`. This triggers a call to `H5MF_alloc_start()` – which performs some sanity checking and then calls H5MF_alloc_create(). That function in turn loads an instance of `H5FS_create_t` with parameters needed to initialize the free list manager, calls `H5FS_create()`, 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`, 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()` to allocate the free list manager data structures. After some sanity checks, H5FS_new() starts by allocating an instance of `H5FS_t`, 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()`. 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()` 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()` 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()

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()` is one of a group of functions that maintain free lists of dynamically allocated blocks of memory using instances of H5FL_blk_head_t. 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.

3. 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.
4. 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;
```

5. 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;
```

2. 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;
   ```

3. 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.

4. 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`.

5. 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()`. 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` via a call to the `H5FL_MALLOC()` 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()` 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: