HDF5 File Format Specification Version 3.0

I. Introduction

The format of an HDF5 file on disk encompasses several key ideas of the HDF4 and AIO file formats as well as addressing some shortcomings therein. The new format is more self-describing than the HDF4 format and is more uniformly applied to data objects in the file.

An HDF5 file appears to the user as a directed graph. The nodes of this graph are the higher-level HDF5 objects that are exposed by the HDF5 APIs:

- Groups
- Datasets
- Committed (formerly Named) datatypes

At the lowest level, as information is actually written to the disk, an HDF5 file is made up of the following objects:

- A superblock
- B-tree nodes
- Heap blocks
- Object headers
- Object data
- Free space

The HDF5 Library uses these low-level objects to represent the higher-level objects that are then presented to the user or to applications through the APIs. For instance, a group is an object header that contains a message that points to a local heap (for storing the links to objects in the group) and to a B-tree (which indexes the links). A dataset is an object header that contains messages that describe the datatype, dataspace, layout, filters, external files, fill value, and other elements with the layout message pointing to either a raw data chunk or to a B-tree that points to raw data chunks.

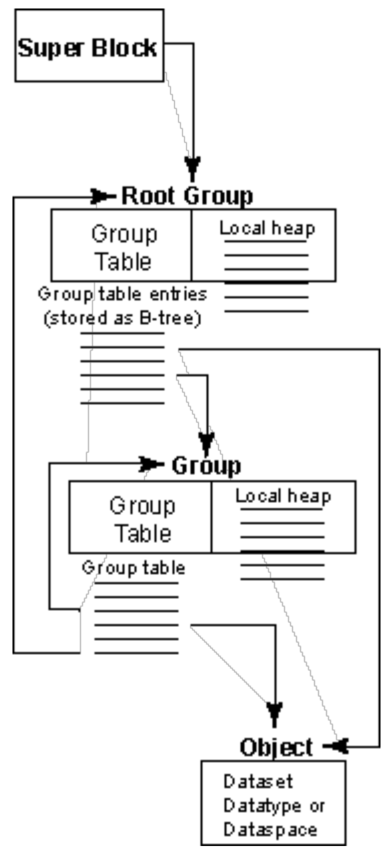

Figure 1: Relationships among the HDF5 root group, other groups, and objects

I.A. This Document

This document describes the lower-level data objects; the higher-level objects and their properties are described in the *HDF5 User's Guide*.

Three levels of information comprise the file format. Level 0 contains basic information for identifying and defining information about the file. Level 1 information contains the information about the pieces of a file shared by many objects in the file (such as B-trees and heaps). Level 2 is the rest of the file and contains all of the data objects with each object partitioned into header information, also known as *metadata*, and data.

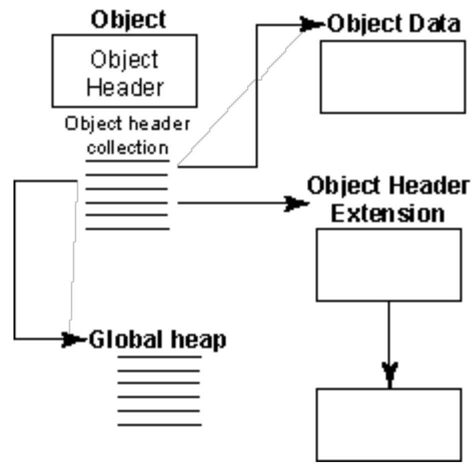

Figure 2: HDF5 objects -- datasets, datatypes, or dataspaces

The various components of the lower-level data objects are described in pairs of tables. The first table shows the format layout, and the second table describes the fields. The titles of format layout tables begin with "Layout". The titles of the tables where the fields are described begin with "Fields". For example, the table that describes the format of the version 2 B-tree header has a title of "Layout: Version 2 B-tree Header", and the fields in the version 2 B-tree header are described in the table titled "Fields: Version 2 B-tree Header".

The sizes of various fields in the following layout tables are determined by looking at the number of columns the field spans in the table. There are exceptions:

- The size may be overridden by specifying a size in parentheses
- The size of addresses is determined by the *Size of Offsets* field in the superblock and is indicated in this document with a superscripted 'O'
- The size of length fields is determined by the *Size of Lengths* field in the superblock and is indicated in this document with a superscripted 'L'

Values for all fields in this document should be treated as unsigned integers, unless otherwise noted in the description of a field. Additionally, all metadata fields are stored in little-endian byte order.

All checksums used in the format are computed with the Jenkins' lookup3 algorithm.

Whenever a bit flag or field is mentioned for an entry, bits are numbered from the lowest bit position in the entry.

Various format tables in this document have cells with "This space inserted only to align table nicely". These entries in the table are just to make the table presentation nicer and do not represent any values or padding in the file.

I.B. Changes for HDF5 1.10

The following sections have been changed or added for the 1.10 release:

- In the "Disk Format: Level 0A - Format Signature and Superblock" section, version 3 of the superblock was added.
- In the "Disk Format: Level 0C - Superblock Extension" section, a link to the Data Storage message was added.

- In the "Disk Format: Level 1A2 - Version 2 B-trees" section, additional B-tree types were added. Tables that describe the type 10 and 11 record layouts were added at the end of the section.
- The "Disk Format: Level 1F - Global Heap Block for Virtual Datasets" was added.
- "The Data Layout Message" section was changed. The name was changed, and version 4 of the data layout message was added for the virtual type.
- The "The File Space Info Message" header message type was added.
- "Appendix C: Types of Indexes for Dataset Chunks" was added. Five indexing types were added.

II. Disk Format: Level 0 - File Metadata

II.A. Disk Format: Level 0A - Format Signature and Superblock

The superblock may begin at certain predefined offsets within the HDF5 file, allowing a block of unspecified content for users to place additional information at the beginning (and end) of the HDF5 file without limiting the HDF5 Library's ability to manage the objects within the file itself. This feature was designed to accommodate wrapping an HDF5 file in another file format or adding descriptive information to an HDF5 file without requiring the modification of the actual file's information. The superblock is located by searching for the HDF5 format signature at byte offset 0, byte offset 512, and at successive locations in the file, each a multiple of two of the previous location; in other words, at these byte offsets: 0, 512, 1024, 2048, and so on.

The superblock is composed of the format signature, followed by a superblock version number and information that is specific to each version of the superblock.

Currently, there are four versions of the superblock format:

- Version 0 is the default format.
- Version 1 is the same as version 0 but with the "*Indexed Storage Internal Node K*" field for storing non-default B-tree 'K' value.
- Version 2 has some fields eliminated and compressed from superblock format versions 0 and 1. It has added checksum support and superblock extension to store additional superblock metadata.
- Version 3 is the same as version 2 except that the field "*File Consistency Flags*" is used for file locking. This format version will enable support for the latest version.

Versions 0 and 1 of the superblock are described below:

Layout: Superblock (Versions 0 and 1)

byte	byte	byte	byte
Format Signature *(8 bytes)*			
Version # of Superblock	Version # of File's Free Space Storage	Version # of Root Group Symbol Table Entry	Reserved *(zero)*
Version Number of Shared Header Message Format	Size of Offsets	Size of Lengths	Reserved *(zero)*
Group Leaf Node K		Group Internal Node K	
File Consistency Flags			
Indexed Storage Internal Node K[1]		Reserved *(zero)*[1]	
Base Address[O]			
Address of File Free space Info[O]			
End of File Address[O]			
Driver Information Block Address[O]			
Root Group Symbol Table Entry			

(Items marked with a '1' in the above table are new in version 1 of the superblock.)

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Superblock (Versions 0 and 1)

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>Format Signature</td><td>
This field contains a constant value and can be used to quickly identify a file as being an HDF5 file. The constant value is designed to allow easy identification of an HDF5 file and to allow certain types of data corruption to be detected. The file signature of an HDF5 file always contains the following values:
<table>
<tr><td>Decimal:</td><td>137</td><td>72</td><td>68</td><td>70</td><td>13</td><td>10</td><td>26</td><td>10</td></tr>
<tr><td>Hexadecimal:</td><td>89</td><td>48</td><td>44</td><td>46</td><td>0d</td><td>0a</td><td>1a</td><td>0a</td></tr>
<tr><td>ASCII C Notation:</td><td>\211</td><td>H</td><td>D</td><td>F</td><td>\r</td><td>\n</td><td>\032</td><td>\n</td></tr>
</table>
This signature both identifies the file as an HDF5 file and provides for immediate detection of common file-transfer problems. The first two bytes distinguish HDF5 files on systems that expect the first two bytes to identify the file type uniquely. The first byte is chosen as a non-ASCII value to reduce the probability that a text file may be misrecognized as an HDF5 file; also, it catches bad file transfers that clear bit 7. Bytes two through four name the format. The CR-LF sequence catches bad file transfers that alter newline sequences. The control-Z character stops file display under MS-DOS. The final line feed checks for the inverse of the CR-LF translation problem. (This is a direct descendent of the PNG file signature.)
This field is present in version 0+ of the superblock.
</td></tr>
<tr><td>Version Number of the Superblock</td><td>
This value is used to determine the format of the information in the superblock. When the format of the information in the superblock is changed, the version number is incremented to the next integer and can be used to determine how the information in the superblock is formatted.
Values of 0, 1 and 2 are defined for this field (the format of version 2 is described below, not here).
This field is present in version 0+ of the superblock.
</td></tr>
<tr><td>Version Number of the File’s Free Space Information</td><td>
This value is used to determine the format of the file’s free space information.
The only value currently valid in this field is ‘0’, which indicates that the file’s free space is as described below.
This field is present in versions 0 and 1 of the superblock.
</td></tr>
</table>

Version Number of the Root Group Symbol Table Entry	This value is used to determine the format of the information in the Root Group Symbol Table Entry. When the format of the information in that field is changed, the version number is incremented to the next integer and can be used to determine how the information in the field is formatted. The only value currently valid in this field is ‘0’, which indicates that the root group symbol table entry is formatted as described below. *This field is present in version 0 and 1 of the superblock.*
Version Number of the Shared Header Message Format	This value is used to determine the format of the information in a shared object header message. Since the format of the shared header messages differs from the other private header messages, a version number is used to identify changes in the format. The only value currently valid in this field is ‘0’, which indicates that shared header messages are formatted as described below. *This field is present in version 0 and 1 of the superblock.*
Size of Offsets	This value contains the number of bytes used to store addresses in the file. The values for the addresses of objects in the file are offsets relative to a base address, usually the address of the superblock signature. This allows a wrapper to be added after the file is created without invalidating the internal offset locations. *This field is present in version 0+ of the superblock.*
Size of Lengths	This value contains the number of bytes used to store the size of an object. *This field is present in version 0+ of the superblock.*
Group Leaf Node K	Each leaf node of a group B-tree will have at least this many entries but not more than twice this many. If a group has a single leaf node then it may have fewer entries. This value must be greater than zero. See the description of B-trees below. *This field is present in version 0 and 1 of the superblock.*
Group Internal Node K	Each internal node of a group B-tree will have at least this many entries but not more than twice this many. If

	the group has only one internal node then it might have fewer entries. This value must be greater than zero. See the description of B-trees below. *This field is present in version 0 and 1 of the superblock.*
File Consistency Flags	This field is unused and should be ignored. *This field is present in version 0+ of the superblock.*
Indexed Storage Internal Node K	Each internal node of an indexed storage B-tree will have at least this many entries but not more than twice this many. If the index storage B-tree has only one internal node then it might have fewer entries. This value must be greater than zero. See the description of B-trees below. *This field is present in version 1 of the superblock.*
Base Address	This is the absolute file address of the first byte of the HDF5 data within the file. The library currently constrains this value to be the absolute file address of the superblock itself when creating new files; future versions of the library may provide greater flexibility. When opening an existing file and this address does not match the offset of the superblock, the library assumes that the entire contents of the HDF5 file have been adjusted in the file and adjusts the base address and end of file address to reflect their new positions in the file. Unless otherwise noted, all other file addresses are relative to this base address. *This field is present in version 0+ of the superblock.*
Address of Global Free-space Index	The file's free space is not persistent for version 0 and 1 of the superblock. Currently this field always contains the undefined address. *This field is present in version 0 and 1 of the superblock.*
End of File Address	This is the absolute file address of the first byte past the end of all HDF5 data. It is used to determine whether a file has been accidently truncated and as an address where file data allocation can occur if space from the free list is not used. *This field is present in version 0+ of the superblock.*
Driver Information Block Address	This is the relative file address of the file driver information block which contains driver-specific

	information needed to reopen the file. If there is no driver information block then this entry should be the undefined address. *This field is present in version 0 and 1 of the superblock.*
Root Group Symbol Table Entry	This is the symbol table entry of the root group, which serves as the entry point into the group graph for the file. *This field is present in version 0 and 1 of the superblock.*

Versions 2 and 3 of the superblock are described below:

Layout: Superblock (Versions 2 and 3)

byte	byte	byte	byte
Format Signature *(8 bytes)*			
Version # of Superblock	Size of Offsets	Size of Lengths	File Consistency Flags
Base Address^{O}			
Superblock Extension Address^{O}			
End of File Address^{O}			
Root Group Object Header Address^{O}			
Superblock Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Superblock (Versions 2 and 3)

Field Name	Description
Format Signature	This field is the same as described for versions 0 and 1 of the superblock.
Version Number of the Superblock	This field has a value of 2 and has the same meaning as for versions 0 and 1.
Size of Offsets	This field is the same as described for versions 0 and 1 of the superblock.
Size of Lengths	This field is the same as described for versions 0 and 1 of the superblock.
File Consistency Flags	For superblock version 2: This field is unused and should be ignored. For superblock version 3: This value contains flags to ensure file consistency for file locking. Currently, the following bit flags are defined: • Bit 0 if set indicates that the file has been opened for write access. • Bit 1 is reserved for future use. • Bit 2 if set indicates that the file has been opened for single-writer/multiple-reader (SWMR) write access. • Bits 3-7 are reserved for future use. Bit 0 should be set as the first action when a file has been opened for write access. Bit 2 should be set when a file has been opened for SWMR write access. These two bits should be cleared only as the final action when closing a file. *This field is present in version 0+ of the superblock.* *The size of this field has been reduced from 4 bytes in superblock format versions 0 and 1 to 1 byte.*
Base Address	This field is the same as described for versions 0 and 1 of the superblock.
Superblock Extension Address	The field is the address of the object header for the superblock extension. If there is no extension then this entry should be the undefined address.
End of File Address	This field is the same as described for versions 0 and 1 of the superblock.

Root Group Object Header Address	This is the address of the root group object header, which serves as the entry point into the group graph for the file.
Superblock Checksum	The checksum for the superblock.

II.B. Disk Format: Level 0B - File Driver Info

The **driver information block** is an optional region of the file which contains information needed by the file driver to reopen a file. The format is described below:

Layout: Driver Information Block

byte	byte	byte	byte
Version	Reserved		
Driver Information Size			
Driver Identification *(8 bytes)*			
Driver Information *(variable size)*			

Fields: Driver Information Block

Field Name	Description
Version	The version number of the Driver Information Block. This document describes version 0.
Driver Information Size	The size in bytes of the *Driver Information* field.
Driver Identification	This is an eight-byte ASCII string without null termination which identifies the driver and/or version number of the Driver Information Block. The predefined driver encoded in this field by the HDF5 Library is identified by the letters NCSA followed by the first four characters of the driver name. If the Driver Information block is not the original version then the last letter(s) of the identification will be replaced by a version number in ASCII, starting with 0. Identification for user-defined drivers is also eight-byte long. It can be arbitrary but should be unique to avoid the four character prefix “NCSA”.
Driver Information	Driver information is stored in a format defined by the file driver (see description below).

The two drivers encoded in the *Driver Identification* field are as follows:

- Multi driver:

  The identifier for this driver is “NCSAmulti”. This driver provides a mechanism for segregating raw data and different types of metadata into multiple files. These files are viewed by the library as a single virtual HDF5 file with a single file address. A maximum of 6 files will be created for the following data: superblock, B-tree, raw data, global heap, local heap, and object header. More than one type of data can be written to the same file.
- Family driver

  The identifier for this driver is “NCSAfami” and is encoded in this field for library version 1.8 and after. This driver is designed for systems that do not support files larger than 2 gigabytes by splitting the HDF5 file address space across several smaller files. It does nothing to segregate metadata and raw data; they are mixed in the address space just as they would be in a single contiguous file.

The format of the *Driver Information* field for the above two drivers are described below:

Layout: Multi Driver Information

byte	byte	byte	byte
Member Mapping	Member Mapping	Member Mapping	Member Mapping
Member Mapping	Member Mapping	Reserved	Reserved
Address of Member File 1			
End of Address for Member File 1			
Address of Member File 2			
End of Address for Member File 2			
... ...			
Address of Member File N			
End of Address for Member File N			
Name of Member File 1 *(variable size)*			
Name of Member File 2 *(variable size)*			
... ...			
Name of Member File N *(variable size)*			

Fields: Multi Driver Information

Field Name	Description
Member Mapping	These fields are integer values from 1 to 6 indicating how the data can be mapped to or merged with another type of data. **Member Mapping** / **Description** 1 The superblock data. 2 The B-tree data. 3 The raw data. 4 The global heap data. 5 The local heap data. 6 The object header data. For example, if the third field has the value 3 and all the rest have the value 1, it means there are two files: one for raw data, and one for superblock, B-tree, global heap, local heap, and object header.
Reserved	These fields are reserved and should always be zero.
Address of Member File N	This field Specifies the virtual address at which the member file starts. N is the number of member files.
End of Address for Member File N	This field is the end of the allocated address for the member file.
Name of Member File N	This field is the null-terminated name of the member file and its length should be multiples of 8 bytes. Additional bytes will be padded with *NULL*s. The default naming convention is *%s-X.h5*, where *X* is one of the letters *s* (for superblock), *b* (for B-tree), *r* (for raw data), *g* (for global heap), *l* (for local heap), and *o* (for object header). The name of the whole HDF5 file will substitute the *%s* in the string.

Layout: Family Driver Information

byte	byte	byte	byte
Size of Member File			

Fields: Family Driver Information

Field Name	Description
Size of Member File	This field is the size of the member file in the family of files.

II.C. Disk Format: Level 0C - Superblock Extension

The *superblock extension* is used to store superblock metadata which is either optional, or added after the version of the superblock was defined. Superblock extensions may only exist when version 2 or later of the superblock is used. A superblock extension is an object header which may hold the following messages:

- Shared Message Table message containing information to locate the master table of shared object header message indices.
- B-tree ‘K’ Values message containing non-default B-tree ‘K’ values.
- Driver Info message containing information needed by the file driver in order to reopen a file. See also the “Disk Format: Level 0B - File Driver Info” section above.
- File Space Info message containing information about file space handling in the file.

III. Disk Format: Level 1 - File Infrastructure

III.A. Disk Format: Level 1A - B-trees and B-tree Nodes

B-trees allow flexible storage for objects which tend to grow in ways that cause the object to be stored discontiguously. B-trees are described in various algorithms books including “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. B-trees are used in several places in the HDF5 file format, when an index is needed for another data structure.

The version 1 B-tree structure described below is the original index structure. The version 1 B-trees are being phased out in favor of the version 2 B-trees described below. Note that both types of structures may be found in the same file depending on the application settings when creating the file.

III.A.1. Disk Format: Level 1A1 - Version 1 B-trees

Version 1 B-trees in HDF5 files are an implementation of the B-link tree. The sibling nodes at a particular level in the tree are stored in a doubly-linked list. See the “Efficient Locking for Concurrent Operations on B-trees” paper by Phillip Lehman and S. Bing Yao as published in the *ACM Transactions on Database Systems*, Vol. 6, No. 4, December 1981.

The B-trees implemented by the file format contain one more key than the number of children. In other words, each child pointer out of a B-tree node has a left key and a right key. The pointers out of internal nodes point to sub-trees while the pointers out of leaf nodes point to symbol nodes and raw data chunks. Aside from that difference, internal nodes and leaf nodes are identical.

Layout: B-tree Nodes

byte	byte	byte	byte
Signature			
Node Type	Node Level	Entries Used	
Address of Left Sibling^O			
Address of Right Sibling^O			
Key 0 *(variable size)*			
Address of Child 0^O			
Key 1 *(variable size)*			
Address of Child 1^O			
...			
Key 2*K* *(variable size)*			
Address of Child 2*K*^O			
Key 2*K*+1 *(variable size)*			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: B-tree Nodes

Field Name	Description
Signature	The ASCII character string “TREE” is used to indicate the beginning of a B-tree node. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Node Type	Each B-tree points to a particular type of data. This field indicates the type of data as well as implying the maximum degree *K* of the tree and the size of each Key field. **Node Type** — **Description** 0 — This tree points to group nodes. 1 — This tree points to raw data chunk nodes.
Node Level	The node level indicates the level at which this node appears in the tree (leaf nodes are at level zero). Not only does the level indicate whether child pointers point to sub-trees or to data, but it can also be used to help file consistency checking utilities reconstruct damaged trees.
Entries Used	This determines the number of children to which this node points. All nodes of a particular type of tree have the same maximum degree, but most nodes will point to less than that number of children. The valid child pointers and keys appear at the beginning of the node and the unused pointers and keys appear at the end of the node. The unused pointers and keys have undefined values.
Address of Left Sibling	This is the relative file address of the left sibling of the current node. If the current node is the left-most node at this level then this field is the undefined address.
Address of Right Sibling	This is the relative file address of the right sibling of the current node. If the current node is the right-most node at this level then this field is the undefined address.
Keys and Child Pointers	Each tree has 2*K*+1 keys with 2*K* child pointers interleaved between the keys. The number of keys and child pointers actually containing valid values is determined by the node’s *Entries Used* field. If that field is *N*, then the B-tree contains *N* child pointers

	and *N*+1 keys.
Key	The format and size of the key values is determined by the type of data to which this tree points. The keys are ordered and are boundaries for the contents of the child pointer; that is, the key values represented by child *N* fall between Key *N* and Key *N*+1. Whether the interval is open or closed on each end is determined by the type of data to which the tree points. The format of the key depends on the node type. For nodes of node type 0 (group nodes), the key is formatted as follows: A single field of *Size of Lengths* bytes: Indicates the byte offset into the local heap for the first object name in the subtree which that key describes. For nodes of node type 1 (chunked raw data nodes), the key is formatted as follows: Bytes 1-4: Size of chunk in bytes. Bytes 4-8: Filter mask, a 32-bit bit field indicating which filters have been skipped for this chunk. Each filter has an index number in the pipeline (starting at 0, with the first filter to apply) and if that filter is skipped, the bit corresponding to its index is set. (*D + 1*) 64-bit fields: The offset of the chunk within the dataset where *D* is the number of dimensions of the dataset, and the last value is the offset within the dataset's datatype and should always be zero. For example, if a chunk in a 3-dimensional dataset begins at the position [5,5,5], there will be three such 64-bit values, each with the value of 5, followed by a 0 value.
Child Pointer	The tree node contains file addresses of subtrees or data depending on the node level. Nodes at Level 0 point to data addresses, either raw data chunks or group nodes. Nodes at non-zero levels point to other nodes of the same B-tree. For raw data chunk nodes, the child pointer is the address of a single raw data chunk. For group nodes, the child pointer points to a symbol table, which contains information for multiple symbol table entries.

Conceptually, each B-tree node looks like this:

key[0] child[0] key[1] child[1] key[2] ... ... key[*N*-1] child[*N*-1] key[*N*]

where child[*i*] is a pointer to a sub-tree (at a level above Level 0) or to data (at Level 0). Each key[*i*] describes an *item* stored by the B-tree (a chunk or an object of a group node). The range of values represented by child[*i*] is indicated by key[*i*] and key[*i*+1].

The following question must next be answered: "Is the value described by key[*i*] contained in child[*i*-1] or in child[*i*]?" The answer depends on the type of tree. In trees for groups (node type 0), the object described by key[*i*] is the greatest object contained in child[*i*-1] while in chunk trees (node type 1) the chunk described by key[*i*] is the least chunk in child[*i*].

That means that key[0] for group trees is sometimes unused; it points to offset zero in the heap, which is always the empty string and compares as "less-than" any valid object name.

And key[*N*] for chunk trees is sometimes unused; it contains a chunk offset which compares as "greater-than" any other chunk offset and has a chunk byte size of zero to indicate that it is not actually allocated.

III.A.2. Disk Format: Level 1A2 - Version 2 B-trees

Version 2 (v2) B-trees are "traditional" B-trees with one major difference. Instead of just using a simple pointer (or address in the file) to a child of an internal node, the pointer to the child node contains two additional pieces of information: the number of records in the child node itself, and the total number of records in the child node and all its descendants. Storing this additional information allows fast array-like indexing to locate the n^{th} record in the B-tree.

The entry into a version 2 B-tree is a header which contains global information about the structure of the B-tree. The *root node address* field in the header points to the B-tree root node, which is either an internal or leaf node, depending on the value in the header's *depth* field. An internal node consists of records plus pointers to further leaf or internal nodes in the tree. A leaf node consists of solely of records. The format of the records depends on the B-tree type (stored in the header).

Layout: Version 2 B-tree Header

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4">Signature</td></tr>
<tr><td>Version</td><td>Type</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Node Size</td></tr>
<tr><td colspan="2">Record Size</td><td colspan="2">Depth</td></tr>
<tr><td>Split Percent</td><td>Merge Percent</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">
Root Node Address^O

</td></tr>
<tr><td colspan="2">Number of Records in Root Node</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">
Total Number of Records in B-tree^L

</td></tr>
<tr><td colspan="4">Checksum</td></tr>
</table>

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Version 2 B-tree Header

Field Name	Description
Signature	The ASCII character string “BTHD” is used to indicate the header of a version 2 (v2) B-tree node.
Version	The version number for this B-tree header. This document describes version 0.
Type	This field indicates the type of B-tree: **Value** — **Description** 0 — This B-tree is used for testing only. This value should *not* be used for storing records in actual HDF5 files. 1 — This B-tree is used for indexing indirectly accessed, non-filtered ‘huge’ fractal heap objects. 2 — This B-tree is used for indexing indirectly accessed, filtered ‘huge’ fractal heap objects. 3 — This B-tree is used for indexing directly accessed, non-filtered ‘huge’ fractal heap objects. 4 — This B-tree is used for indexing directly accessed, filtered ‘huge’ fractal heap objects. 5 — This B-tree is used for indexing the ‘name’ field for links in indexed groups. 6 — This B-tree is used for indexing the ‘creation order’ field for links in indexed groups. 7 — This B-tree is used for indexing shared object header messages. 8 — This B-tree is used for indexing the ‘name’ field for indexed attributes. 9 — This B-tree is used for indexing the ‘creation order’ field for indexed attributes. 10 — This B-tree is used for indexing chunks of datasets with no filters and with more than one dimension of unlimited extent. 11 — This B-tree is used for indexing chunks of datasets with filters and more than one dimension of unlimited extent. The format of records for each type is described below.

Node Size	This is the size in bytes of all B-tree nodes.
Record Size	This field is the size in bytes of the B-tree record.
Depth	This is the depth of the B-tree.
Split Percent	The percent full that a node needs to increase above before it is split.
Merge Percent	The percent full that a node needs to be decrease below before it is split.
Root Node Address	This is the address of the root B-tree node. A B-tree with no records will have the undefined address in this field.
Number of Records in Root Node	This is the number of records in the root node.
Total Number of Records in B-tree	This is the total number of records in the entire B-tree.
Checksum	This is the checksum for the B-tree header.

Layout: Version 2 B-tree Internal Node

byte	byte	byte	byte
Signature			
Version	Type	Records 0, 1, 2...N-1 *(variable size)*	
Child Node Pointer 0[O]			
Number of Records N_0 for Child Node 0 *(variable size)*			
Total Number of Records for Child Node 0 *(optional, variable size)*			
Child Node Pointer 1[O]			
Number of Records N_1 for Child Node 1 *(variable size)*			
Total Number of Records for Child Node 1 *(optional, variable size)*			
...			
Child Node Pointer N[O]			
Number of Records N_n for Child Node N *(variable size)*			
Total Number of Records for Child Node N *(optional, variable size)*			
Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Version 2 B-tree Internal Node

Field Name	Description
Signature	The ASCII character string “BTIN” is used to indicate the internal node of a B-tree.
Version	The version number for this B-tree internal node. This document describes version 0.
Type	This field is the type of the B-tree node. It should always be the same as the B-tree type in the header.
Records	The size of this field is determined by the number of records for this node and the record size (from the header). The format of records depends on the type of B-tree.
Child Node Pointer	This field is the address of the child node pointed to by the internal node.
Number of Records in Child Node	This is the number of records in the child node pointed to by the corresponding *Node Pointer*. The number of bytes used to store this field is determined by the maximum possible number of records able to be stored in the child node. The maximum number of records in a child node is computed in the following way: • Subtract the fixed size overhead for the child node (for example, its signature, version, checksum, and so on and *one* pointer triplet of information for the child node (because there is one more pointer triplet than records in each internal node)) from the size of nodes for the B-tree. • Divide that result by the size of a record plus the pointer triplet of information stored to reach each child node from this node. Note that leaf nodes do not encode any child pointer triplets, so the maximum number of records in a leaf node is just the node size minus the leaf node overhead, divided by the record size. Also note that the first level of internal nodes above the leaf nodes do not encode the *Total Number of Records in Child Node* value in the child pointer triplets (since it is the same as the *Number of Records in Child Node*), so the maximum number of records in these nodes is computed with the

	equation above, but using (*Child Pointer*, *Number of Records in Child Node*) pairs instead of triplets. The number of bytes used to encode this field is the least number of bytes required to encode the maximum number of records in a child node value for the child nodes below this level in the B-tree. For example, if the maximum number of child records is 123, one byte will be used to encode these values in this node; if the maximum number of child records is 20000, two bytes will be used to encode these values in this node; and so on. The maximum number of bytes used to encode these values is 8 (in other words, an unsigned 64-bit integer).
Total Number of Records in Child Node	This is the total number of records for the node pointed to by the corresponding *Node Pointer* and all its children. This field exists only in nodes whose depth in the B-tree node is greater than 1 (in other words, the “twig” internal nodes, just above leaf nodes, do not store this field in their child node pointers). The number of bytes used to store this field is determined by the maximum possible number of records able to be stored in the child node and its descendants. The maximum possible number of records able to be stored in a child node and its descendants is computed iteratively, in the following way: The maximum number of records in a leaf node is computed, then that value is used to compute the maximum possible number of records in the first level of internal nodes above the leaf nodes. Multiplying these two values together determines the maximum possible number of records in child node pointers for the level of nodes two levels above leaf nodes. This process is continued up to any level in the B-tree. The number of bytes used to encode this value is computed in the same way as for the *Number of Records in Child Node* field.
Checksum	This is the checksum for this node.

Layout: Version 2 B-tree Leaf Node

byte	byte	byte	byte
Signature			
Version	Type	Record 0, 1, 2...N-1 *(variable size)*	
Checksum			

Fields: Version 2 B-tree Leaf Node

Field Name	Description
Signature	The ASCII character string “BTLF“ is used to indicate the leaf node of a version 2 (v2) B-tree.
Version	The version number for this B-tree leaf node. This document describes version 0.
Type	This field is the type of the B-tree node. It should always be the same as the B-tree type in the header.
Records	The size of this field is determined by the number of records for this node and the record size (from the header). The format of records depends on the type of B-tree.
Checksum	This is the checksum for this node.

The record layout for each stored (in other words, non-testing) B-tree type is as follows:

Layout: Version 2 B-tree, Type 1 Record Layout - Indirectly Accessed, Non-filtered, ‘Huge’ Fractal Heap Objects

byte	byte	byte	byte
Huge Object Address[O]			
Huge Object Length[L]			
Huge Object ID[L]			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Version 2 B-tree, Type 1 Record Layout - Indirectly Accessed, Non-filtered, ‘Huge’ Fractal Heap Objects

Field Name	Description
Huge Object Address	The address of the huge object in the file.
Huge Object Length	The length of the huge object in the file.
Huge Object ID	The heap ID for the huge object.

Layout: Version 2 B-tree, Type 2 Record Layout - Indirectly Accessed, Filtered, 'Huge' Fractal Heap Objects

byte	byte	byte	byte
Filtered Huge Object Address[O]			
Filtered Huge Object Length[L]			
Filter Mask			
Filtered Huge Object Memory Size[L]			
Huge Object ID[L]			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Version 2 B-tree, Type 2 Record Layout - Indirectly Accessed, Filtered, 'Huge' Fractal Heap Objects

Field Name	Description
Filtered Huge Object Address	The address of the filtered huge object in the file.
Filtered Huge Object Length	The length of the filtered huge object in the file.
Filter Mask	A 32-bit bit field indicating which filters have been skipped for this chunk. Each filter has an index number in the pipeline (starting at 0, with the first filter to apply) and if that filter is skipped, the bit corresponding to its index is set.
Filtered Huge Object Memory Size	The size of the de-filtered huge object in memory.
Huge Object ID	The heap ID for the huge object.

Layout: Version 2 B-tree, Type 3 Record Layout - Directly Accessed, Non-filtered, 'Huge' Fractal Heap Objects

byte	byte	byte	byte
Huge Object Address^{O}			
Huge Object Length^{L}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Version 2 B-tree, Type 3 Record Layout - Directly Accessed, Non-filtered, 'Huge' Fractal Heap Objects

Field Name	Description
Huge Object Address	The address of the huge object in the file.
Huge Object Length	The length of the huge object in the file.

Layout: Version 2 B-tree, Type 4 Record Layout - Directly Accessed, Filtered, 'Huge' Fractal Heap Objects

byte	byte	byte	byte
Filtered Huge Object Address^{O}			
Filtered Huge Object Length^{L}			
Filter Mask			
Filtered Huge Object Memory Size^{L}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Version 2 B-tree, Type 4 Record Layout - Directly Accessed, Filtered, ‘Huge’ Fractal Heap Objects

Field Name	Description
Filtered Huge Object Address	The address of the filtered huge object in the file.
Filtered Huge Object Length	The length of the filtered huge object in the file.
Filter Mask	A 32-bit bit field indicating which filters have been skipped for this chunk. Each filter has an index number in the pipeline (starting at 0, with the first filter to apply) and if that filter is skipped, the bit corresponding to its index is set.
Filtered Huge Object Memory Size	The size of the de-filtered huge object in memory.

Layout: Version 2 B-tree, Type 5 Record Layout - Link Name for Indexed Group

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4">Hash of Name</td></tr>
<tr><td colspan="4">ID (bytes 1-4)</td></tr>
<tr><td colspan="3">ID (bytes 5-7)</td><td></td></tr>
</table>

Fields: Version 2 B-tree, Type 5 Record Layout - Link Name for Indexed Group

Field Name	Description
Hash	This field is hash value of the name for the link. The hash value is the Jenkins’ lookup3 checksum algorithm applied to the link’s name.
ID	This is a 7-byte sequence of bytes and is the heap ID for the link record in the group’s fractal heap.

Layout: Version 2 B-tree, Type 6 Record Layout - Creation Order for Indexed Group

byte	byte	byte	byte
Creation Order *(8 bytes)*			
ID *(bytes 1-4)*			
ID *(bytes 5-7)*			

Fields: Version 2 B-tree, Type 6 Record Layout - Creation Order for Indexed Group

Field Name	Description
Creation Order	This field is the creation order value for the link.
ID	This is a 7-byte sequence of bytes and is the heap ID for the link record in the group's fractal heap.

Layout: Version 2 B-tree, Type 7 Record Layout - Shared Object Header Messages (Sub-type 0 - Message in Heap)

byte	byte	byte	byte
Message Location	*This space inserted only to align table nicely*		
Hash			
Reference Count			
Heap ID *(8 bytes)*			

Fields: Version 2 B-tree, Type 7 Record Layout - Shared Object Header Messages (Sub-type 0 - Message in Heap)

Field Name	Description
Message Location	This field Indicates the location where the message is stored: **Value** — **Description** 0 — Shared message is stored in shared message index heap. 1 — Shared message is stored in object header.

Layout: Version 2 B-tree, Type 7 Record Layout - Shared Object Header Messages (Sub-type 1 - Message in Object Header)

byte	byte	byte	byte
Message Location	*This space inserted only to align table nicely*		
Hash			
Reserved (zero)	Message Type	Object Header Index	
Object Header Address^{O}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Version 2 B-tree, Type 7 Record Layout - Shared Object Header Messages (Sub-type 1 - Message in Object Header)

Field Name	Description
Message Location	This field Indicates the location where the message is stored: **Value** / **Description** 0 — Shared message is stored in shared message index heap. 1 — Shared message is stored in object header.
Hash	This field is hash value of the shared message. The hash value is the Jenkins' lookup3 checksum algorithm applied to the shared message.
Message Type	The object header message type of the shared message.
Object Header Index	This field indicates that the shared message is the n^{th} message of its type in the specified object header.
Object Header Address	The address of the object header containing the shared message.

Layout: Version 2 B-tree, Type 8 Record Layout - Attribute Name for Indexed Attributes

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4">Heap ID (8 bytes)</td></tr>
<tr><td>Message Flags</td><td colspan="3">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Creation Order</td></tr>
<tr><td colspan="4">Hash of Name</td></tr>
</table>

Fields: Version 2 B-tree, Type 8 Record Layout - Attribute Name for Indexed Attributes

Field Name	Description
Heap ID	This is an 8-byte sequence of bytes and is the heap ID for the attribute in the object’s attribute fractal heap.
Message Flags	The object header message flags for the attribute message.
Creation Order	This field is the creation order value for the attribute.
Hash	This field is hash value of the name for the attribute. The hash value is the Jenkins’ lookup3 checksum algorithm applied to the attribute’s name.

Layout: Version 2 B-tree, Type 9 Record Layout - Creation Order for Indexed Attributes

byte	byte	byte	byte
Heap ID *(8 bytes)*			
Message Flags	*This space inserted only to align table nicely*		
Creation Order			

Fields: Version 2 B-tree, Type 9 Record Layout - Creation Order for Indexed Attributes

Field Name	Description
Heap ID	This is an 8-byte sequence of bytes and is the heap ID for the attribute in the object’s attribute fractal heap.
Message Flags	The object header message flags for the attribute message.
Creation Order	This field is the creation order value for the attribute.

Layout: Version 2 B-tree, Type 10 Record Layout - Non-filtered Dataset Chunks

byte	byte	byte	byte
Address[O]			
Dimension 0 Scaled Offset *(8 bytes)*			
Dimension 1 Scaled Offset *(8 bytes)*			
...			
Dimension #n Scaled Offset *(8 bytes)*			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Version 2 B-tree, Type 10 Record Layout - Non-filtered Dataset Chunks

Field Name	Description
Address	This field is the address of the dataset chunk in the file.
Dimension #n Scaled Offset	This field is the scaled offset of the chunk within the dataset. *n* is the number of dimensions for the dataset. The first scaled offset stored in the list is for the slowest changing dimension, and the last scaled offset stored is for the fastest changing dimension. Scaled offset is calculated by dividing the chunk dimension sizes into the chunk offsets.

Layout: Version 2 B-tree, Type 11 Record Layout - Filtered Dataset Chunks

byte	byte	byte	byte
Address[O]			
Chunk Size *(variable size; at most 8 bytes)*			
Filter Mask			
Dimension 0 Scaled Offset *(8 bytes)*			
Dimension 1 Scaled Offset *(8 bytes)*			
...			
Dimension #n Scaled Offset *(8 bytes)*			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Version 2 B-tree, Type 11 Record Layout - Filtered Dataset Chunks

Field Name	Description
Address	This field is the address of the dataset chunk in the file.
Chunk Size	This field is the size of the dataset chunk in bytes.
Filter Mask	This field is the filter mask which indicates the filter to skip for the dataset chunk. Each filter has an index number in the pipeline and if that filter is skipped, the bit corresponding to its index is set.
Dimension #n Scaled Offset	This field is the scaled offset of the chunk within the dataset. *n* is the number of dimensions for the dataset. The first scaled offset stored in the list is for the slowest changing dimension, and the last scaled offset stored is for the fastest changing dimension.

III.B. Disk Format: Level 1B - Group Symbol Table Nodes

A group is an object internal to the file that allows arbitrary nesting of objects within the file (including other groups). A group maps a set of link names in the group to a set of relative file addresses of objects in the file. Certain metadata for an object to which the group points can be cached in the group's symbol table entry in addition to being in the object's header.

An HDF5 object name space can be stored hierarchically by partitioning the name into components and storing each component as a link in a group. The link for a non-ultimate component points to the group containing the next component. The link for the last component points to the object being named.

One implementation of a group is a collection of symbol table nodes indexed by a B-tree. Each symbol table node contains entries for one or more links. If an attempt is made to add a link to an already full symbol table node containing 2*K* entries, then the node is split and one node contains *K* symbols and the other contains *K*+1 symbols.

Layout: Symbol Table Node (A Leaf of a B-tree)

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4" align="center">Signature</td></tr>
<tr><td>Version Number</td><td>Reserved (zero)</td><td colspan="2" align="center">Number of Symbols</td></tr>
<tr><td colspan="4" align="center">

Group Entries

</td></tr>
</table>

Fields: Symbol Table Node (A Leaf of a B-tree)

Field Name	Description
Signature	The ASCII character string “SNOD” is used to indicate the beginning of a symbol table node. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version Number	The version number for the symbol table node. This document describes version 1. (There is no version ‘0’ of the symbol table node)
Number of Entries	Although all symbol table nodes have the same length, most contain fewer than the maximum possible number of link entries. This field indicates how many entries contain valid data. The valid entries are packed at the beginning of the symbol table node while the remaining entries contain undefined values.
Symbol Table Entries	Each link has an entry in the symbol table node. The format of the entry is described below. There are 2*K* entries in each group node, where *K* is the “Group Leaf Node K” value from the superblock.

III.C. Disk Format: Level 1C - Symbol Table Entry

Each symbol table entry in a symbol table node is designed to allow for very fast browsing of stored objects. Toward that design goal, the symbol table entries include space for caching certain constant metadata from the object header.

Layout: Symbol Table Entry

byte	byte	byte	byte
Link Name Offset[O]			
Object Header Address[O]			
Cache Type			
Reserved *(zero)*			
Scratch-pad Space *(16 bytes)*			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Symbol Table Entry

Field Name	Description
Link Name Offset	This is the byte offset into the group's local heap for the name of the link. The name is null terminated.
Object Header Address	Every object has an object header which serves as a permanent location for the object's metadata. In addition to appearing in the object header, some of the object's metadata can be cached in the scratch-pad space.
Cache Type	The cache type is determined from the object header. It also determines the format for the scratch-pad space: **Type** / **Description** 0 — No data is cached by the group entry. This is guaranteed to be the case when an object header has a link count greater than one. 1 — Group object header metadata is cached in the scratch-pad space. This implies that the symbol table entry refers to another group. 2 — The entry is a symbolic link. The first four bytes of the scratch-pad space are the offset into the local heap for the link value. The object header address will be undefined.
Reserved	These four bytes are present so that the scratch-pad space is aligned on an eight-byte boundary. They are always set to zero.
Scratch-pad Space	This space is used for different purposes, depending on the value of the Cache Type field. Any metadata about an object represented in the scratch-pad space is duplicated in the object header for that object. Furthermore, no data is cached in the group entry scratch-pad space if the object header for the object has a link count greater than one.

Format of the Scratch-pad Space

The symbol table entry scratch-pad space is formatted according to the value in the Cache Type field.

If the Cache Type field contains the value zero (0) then no information is stored in the scratch-pad space.

If the Cache Type field contains the value one (1), then the scratch-pad space contains cached metadata for another object header in the following format:

Layout: Object Header Scratch-pad Format

byte	byte	byte	byte
Address of B-tree[O]			
Address of Name Heap[O]			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Object Header Scratch-pad Format

Field Name	Description
Address of B-tree	This is the file address for the root of the group's B-tree.
Address of Name Heap	This is the file address for the group's local heap, in which are stored the group's symbol names.

If the Cache Type field contains the value two (2), then the scratch-pad space contains cached metadata for a symbolic link in the following format:

Layout: Symbolic Link Scratch-pad Format

byte	byte	byte	byte
Offset to Link Value			

Fields: Symbolic Link Scratch-pad Format

Field Name	Description
Offset to Link Value	The value of a symbolic link (that is, the name of the thing to which it points) is stored in the local heap. This field is the 4-byte offset into the local heap for the start of the link value, which is null terminated.

III.D. Disk Format: Level 1D - Local Heaps

A local heap is a collection of small pieces of data that are particular to a single object in the HDF5 file. Objects can be inserted and removed from the heap at any time. The address of a heap does not change once the heap is created. For example, a group stores addresses of objects in symbol table nodes with the names of links stored in the group's local heap.

Layout: Local Heap

byte	byte	byte	byte
Signature			
Version	Reserved *(zero)*		
Data Segment Size[L]			
Offset to Head of Free-list[L]			
Address of Data Segment[O]			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Local Heap

Field Name	Description
Signature	The ASCII character string “HEAP” is used to indicate the beginning of a heap. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	Each local heap has its own version number so that new heaps can be added to old files. This document describes version zero (0) of the local heap.
Data Segment Size	The total amount of disk memory allocated for the heap data. This may be larger than the amount of space required by the objects stored in the heap. The extra unused space in the heap holds a linked list of free blocks.
Offset to Head of Free-list	This is the offset within the heap data segment of the first free block (or the undefined address if there is no free block). The free block contains Size of Lengths bytes that are the offset of the next free block (or the value ‘1’ if this is the last free block) followed by Size of Lengths bytes that store the size of this free block. The size of the free block includes the space used to store the offset of the next free block and the size of the current block, making the minimum size of a free block 2 * Size of Lengths.
Address of Data Segment	The data segment originally starts immediately after the heap header, but if the data segment must grow as a result of adding more objects, then the data segment may be relocated, in its entirety, to another part of the file.

Objects within a local heap should be aligned on an 8-byte boundary.

III.E. Disk Format: Level 1E - Global Heap

Each HDF5 file has a global heap which stores various types of information which is typically shared between datasets. The global heap was designed to satisfy these goals:

A. Repeated access to a heap object must be efficient without resulting in repeated file I/O requests. Since global heap objects will typically be shared among several datasets, it is probable that the object will be accessed repeatedly.
B. Collections of related global heap objects should result in fewer and larger I/O requests. For instance, a dataset of object references will have a global heap object for each reference. Reading the entire set of object references should result in a few large I/O requests instead of one small I/O request for each reference.
C. It should be possible to remove objects from the global heap and the resulting file hole

should be eligible to be reclaimed for other uses.

The implementation of the heap makes use of the memory management already available at the file level and combines that with a new object called a *collection* to achieve goal B. The global heap is the set of all collections. Each global heap object belongs to exactly one collection, and each collection contains one or more global heap objects. For the purposes of disk I/O and caching, a collection is treated as an atomic object, addressing goal A.

When a global heap object is deleted from a collection (which occurs when its reference count falls to zero), objects located after the deleted object in the collection are packed down toward the beginning of the collection, and the collection's global heap object 0 is created (if possible), or its size is increased to account for the recently freed space. There are no gaps between objects in each collection, with the possible exception of the final space in the collection, if it is not large enough to hold the header for the collection's global heap object 0. These features address goal C.

The HDF5 Library creates global heap collections as needed, so there may be multiple collections throughout the file. The set of all of them is abstractly called the "global heap", although they do not actually link to each other, and there is no global place in the file where you can discover all of the collections. The collections are found simply by finding a reference to one through another object in the file. For example, data of variable-length datatype elements is stored in the global heap and is accessed via a global heap ID. The format for global heap IDs is described at the end of this section.

For more information on global heaps for virtual datasets, see "Disk Format: Level 1F - Global Heap Block for Virtual Datasets."

Layout: A Global Heap Collection

byte	byte	byte	byte
Signature			
Version	Reserved (zero)		
Collection Size[L]			
Global Heap Object 1			
Global Heap Object 2			
...			
Global Heap Object *N*			
Global Heap Object 0 (free space)			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: A Global Heap Collection

Field Name	Description
Signature	The ASCII character string “GCOL” is used to indicate the beginning of a collection. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	Each collection has its own version number so that new collections can be added to old files. This document describes version one (1) of the collections (there is no version zero (0)).
Collection Size	This is the size in bytes of the entire collection including this field. The default (and minimum) collection size is 4096 bytes which is a typical file system block size. This allows for 127 16-byte heap objects plus their overhead (the collection header of 16 bytes and the 16 bytes of information about each heap object).
Global Heap Object 1 through *N*	The objects are stored in any order with no intervening unused space.
Global Heap Object 0	Global Heap Object 0 (zero), when present, represents the free space in the collection. Free space always appears at the end of the collection. If the free space is too small to store the header for Object 0 (described below) then the header is implied and the collection contains no free space.

Layout: Global Heap Object

byte	byte	byte	byte
Heap Object Index		Reference Count	
Reserved (zero)			
Object Size[L]			
Object Data			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Global Heap Object

Field Name	Description
Heap Object Index	Each object has a unique identification number within a collection. The identification numbers are chosen so that new objects have the smallest value possible with the exception that the identifier 0 always refers to the object which represents all free space within the collection.
Reference Count	All heap objects have a reference count field. An object which is referenced from some other part of the file will have a positive reference count. The reference count for Object 0 is always zero.
Reserved	Zero padding to align next field on an 8-byte boundary.
Object Size	This is the size of the object data stored for the object. The actual storage space allocated for the object data is rounded up to a multiple of eight.
Object Data	The object data is treated as a one-dimensional array of bytes to be interpreted by the caller.

The format for the ID used to locate an object in the global heap is described here:

Layout: Global Heap ID

byte	byte	byte	byte
Collection Address^O			
Object Index			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Global Heap ID

Field Name	Description
Collection Address	This field is the address of the global heap collection where the data object is stored.
ID	This field is the index of the data object within the global heap collection.

III.F. Disk Format: Level 1F - Global Heap Block for Virtual Datasets

The layout for the global heap block used with virtual datasets is described below. For more information on global heaps, see "Disk Format: Level 1E - Global Heap."

Layout: Global Heap Block for Virtual Dataset

byte	byte	byte	byte
Version	*This space inserted only to align table nicely*		
Num Entries[L]			
Source Filename #1 *(variable size)*			
Source Dataset #1 *(variable size)*			
Source Selection #1 *(variable size)*			
Virtual Selection #1 *(variable size)*			
. . .			
Source Filename #n *(variable size)*			
Source Dataset #n *(variable size)*			
Source Selection #n *(variable size)*			
Virtual Selection #n *(variable size)*			
Checksum			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Global Heap Block for Virtual Dataset

Field Name	Description
Version	The version number for the block; the value is 0.
Num Entries	The number of entries in the block.
Source Filename #n	The source file name where the source dataset is located.
Source Dataset #n	The source dataset name that is mapped to the virtual dataset.
Source Selection #n	The dataspace selection in the source dataset that is mapped to the virtual selection.
Virtual Selection #n	This is the dataspace selection in the virtual dataset that is mapped to the source selection.
Checksum	This is the checksum for the block.

Layout: Dataspace Selection

byte	byte	byte	byte
Selection Type			
Selection Info (*variable size*)			

Fields: Dataspace Selection

Field Name	Description
Selection Type	There are 4 types of selection: **Value** / **Description** 0 — H5S_SEL_NONE: Nothing selected 1 — H5S_SEL_POINTS: Sequence of points selected 2 — H5S_SEL_HYPER: Hyperslab selected 3 — H5S_SEL_ALL: Entire extent selected
Selection Info	**Value** / **Description** 0 — H5S_SEL_NONE 1 — H5S_SEL_POINTS 2 — H5S_SEL_HYPER 3 — H5S_SEL_ALL

Layout: Selection Info for H5S_SEL_NONE

byte	byte	byte	byte
Version			
Reserved *(zero, 8 bytes)*			

Fields: Selection Info for H5S_SEL_NONE

Field Name	Description
Version	The version number for the H5S_SEL_NONE Selection Info; the value is 1.

Layout: Selection Info for H5S_SEL_POINTS

byte	byte	byte	byte
Version			
Reserved *(zero)*			
Length			
Rank			
Num Points			
Point #1: coordinate #1			
. . .			
Point #1: coordinate #u			
. . .			
Point #n: coordinate #1			
. . .			
Point #n: coordinate #u			

Fields: Selection Info for H5S_SEL_POINTS

Field Name	Description
Version	The version number for the H5S_SEL_POINTS Selection Info; the value is 1.
Length	The size in bytes from *Length* to the end of the Selection Info.
Rank	The number of dimensions.
Num Points	The number of points in the selection.
Point #n: coordinate #u	The array of points in the selection. The points selected are #1 to #n where n is *Num Points*. The list of coordinates for each point are #1 to #u where u is *Rank*.

Layout: Selection Info for H5S_SEL_HYPER

byte	byte	byte	byte
Version			
Hyperslab Selection Info (*variable size*)			

Fields: Selection Info for H5S_SEL_HYPER

Field Name	Description
Version	The version number for the H5S_SEL_HYPER Selection Info. The value is either 1 or 2.
Hyperslab Selection Info	Depending on *version*: **Version** / **Description** 1 — Version 1 hyperslab selection info for dataspace without H5S_UNLIMITED 2 — Version 2 hyperslab selection info for dataspace with H5S_UNLIMITED

Layout: Version 1 Hyperslab Selection Info

byte	byte	byte	byte
Reserved			
Length			
Rank			
Num Blocks			
Start offset #1 for block #1 *(8 bytes)*			
. . .			
Start offset #n for block #1 *(8 bytes)*			
End offset #1 for block #1 *(8 bytes)*			
. . .			
End offset #n for block #1 *(8 bytes)*			
. . .			
. . .			
. . .			
Start offset #1 for block #u *(8 bytes)*			
. . .			
Start offset #n for block #u *(8 bytes)*			

End offset #1 for block #u *(8 bytes)*
. . .
End offset #n for block #u *(8 bytes)*

Fields: Version 1 Hyperslab Selection Info

Field Name	Description
Length	The size in bytes from the field *Length* to the end of the Selection Info.
Rank	The number of dimensions in the dataspace.
Num Blocks	The number of blocks in the selection.
Start offset #n for block #u	The offset #n of the starting element in block #u. #n is from 1 to *Rank*. #u is from 1 to *Num Blocks* moving from the fastest changing dimension to the slowest changing dimension.
End offset #n for block #u	The offset #n of the ending element in block #u. #n is from 1 to *Rank*. #u is from 1 to *Num Blocks* moving from the fastest changing dimension to the slowest changing dimension.

Layout: Version 2 Hyperslab Selection Info

byte	byte	byte	byte
flags	*This space inserted only to align table nicely*		
Length			
Rank			
Start #1 *(8 bytes)*			
Stride #1 *(8 bytes)*			
Count #1 *(8 bytes)*			
Block #1 *(8 bytes)*			
. . .			
Start #n *(8 bytes)*			
Stride #n *(8 bytes)*			
Count #n *(8 bytes)*			
Block #n *(8 bytes)*			

Fields: Version 2 Hyperslab Selection Info

Field Name	Description
Length	The size in bytes from the field *Length* to the end of the Selection Info.
Rank	The number of dimensions in the dataspace.
Start #n	The offset of the starting element in the block. #n is from 1 to *Rank*.
Stride #n	The number of elements to move in each dimension. #n is from 1 to *Rank*.
Count #n	The number of blocks to select in each dimension. #n is from 1 to *Rank*.
Block #n	The size (in elements) of each block in each dimension. #n is from 1 to *Rank*.

Layout: Selection Info for H5S_SEL_ALL

byte	byte	byte	byte
Version			
Reserved *(zero, 8 bytes)*			

Fields: Selection Info for H5S_SEL_ALL

Field Name	Description
Version	The version number for the H5S_SEL_ALL Selection Info; the value is 1.

III.G. Disk Format: Level 1G - Fractal Heap

Each fractal heap consists of a header and zero or more direct and indirect blocks (described below). The header contains general information as well as initialization parameters for the doubling table. The *Root Block Address* in the header points to the first direct or indirect block in the heap.

Fractal heaps are based on a data structure called a *doubling table*. A doubling table provides a mechanism for quickly extending an array-like data structure that minimizes the number of empty blocks in the heap, while retaining very fast lookup of any element within the array. More information on fractal heaps and doubling tables can be found in the RFC "Private Heaps in

HDF5."

The fractal heap implements the doubling table structure with indirect and direct blocks. Indirect blocks in the heap do not actually contain data for objects in the heap, their "size" is abstract - they represent the indexing structure for locating the direct blocks in the doubling table. Direct blocks contain the actual data for objects stored in the heap.

All indirect blocks have a constant number of block entries in each row, called the *width* of the doubling table (stored in the heap header). The number of rows for each indirect block in the heap is determined by the size of the block that the indirect block represents in the doubling table (calculation of this is shown below) and is constant, except for the "root" indirect block, which expands and shrinks its number of rows as needed.

Blocks in the first *two* rows of an indirect block are *Starting Block Size* number of bytes in size, and the blocks in each subsequent row are twice the size of the blocks in the previous row. In other words, blocks in the third row are twice the *Starting Block Size*, blocks in the fourth row are four times the *Starting Block Size*, and so on. Entries for blocks up to the *Maximum Direct Block Size* point to direct blocks, and entries for blocks greater than that size point to further indirect blocks (which have their own entries for direct and indirect blocks).

The number of rows of blocks, *nrows*, in an indirect block of size *iblock_size* is given by the following expression:

$$nrows = (\log_2(iblock\_size) - \log_2(<Starting\ Block\ Size> * <Width>)) + 1$$

The maximum number of rows of direct blocks, *max_dblock_rows*, in any indirect block of a fractal heap is given by the following expression:

$$max\_dblock\_rows = (\log_2(<Max.\ Direct\ Block\ Size>) - \log_2(<Starting\ Block\ Size>)) + 2$$

Using the computed values for *nrows* and *max_dblock_rows*, along with the *Width* of the doubling table, the number of direct and indirect block entries (*K* and *N* in the indirect block description, below) in an indirect block can be computed:

$$K = \mathrm{MIN}(nrows, max\_dblock\_rows) * Width$$

If *nrows* is less than or equal to *max_dblock_rows*, *N* is 0. Otherwise, *N* is simply computed:

$$N = K - (max\_dblock\_rows * Width)$$

The size indirect blocks on disk is determined by the number of rows in the indirect block (computed above). The size of direct blocks on disk is exactly the size of the block in the doubling table.

Layout: Fractal Heap Header

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4">Signature</td></tr>
<tr><td>Version</td><td colspan="3">This space inserted only to align table nicely</td></tr>
<tr><td colspan="2">Heap ID Length</td><td colspan="2">I/O Filters’ Encoded Length</td></tr>
<tr><td>Flags</td><td colspan="3">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Maximum Size of Managed Objects</td></tr>
<tr><td colspan="4">Next Huge Object ID^L</td></tr>
<tr><td colspan="4">v2 B-tree Address of Huge Objects^O</td></tr>
<tr><td colspan="4">Amount of Free Space in Managed Blocks^L</td></tr>
<tr><td colspan="4">Address of Managed Block Free Space Manager^O</td></tr>
<tr><td colspan="4">Amount of Managed Space in Heap^L</td></tr>
<tr><td colspan="4">Amount of Allocated Managed Space in Heap^L</td></tr>
<tr><td colspan="4">Offset of Direct Block Allocation Iterator in Managed Space^L</td></tr>
<tr><td colspan="4">Number of Managed Objects in Heap^L</td></tr>
<tr><td colspan="4">Size of Huge Objects in Heap^L</td></tr>
<tr><td colspan="4">Number of Huge Objects in Heap^L</td></tr>
<tr><td colspan="4">Size of Tiny Objects in Heap^L</td></tr>
</table>

<table>
<tr><td colspan="2">Number of Tiny Objects in Heap^{L}</td></tr>
<tr><td>Table Width</td><td>This space inserted only to align table nicely</td></tr>
<tr><td colspan="2">Starting Block Size^{L}</td></tr>
<tr><td colspan="2">Maximum Direct Block Size^{L}</td></tr>
<tr><td>Maximum Heap Size</td><td>Starting # of Rows in Root Indirect Block</td></tr>
<tr><td colspan="2">Address of Root Block^{O}</td></tr>
<tr><td>Current # of Rows in Root Indirect Block</td><td>This space inserted only to align table nicely</td></tr>
<tr><td colspan="2">Size of Filtered Root Direct Block (optional)^{L}</td></tr>
<tr><td colspan="2">I/O Filter Mask (optional)</td></tr>
<tr><td colspan="2">I/O Filter Information (optional, variable size)</td></tr>
<tr><td colspan="2">Checksum</td></tr>
</table>

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Fractal Heap Header

Field Name	Description
Signature	The ASCII character string “FRHP” is used to indicate the beginning of a fractal heap header. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Heap ID Length	This is the length in bytes of heap object IDs for this heap.
I/O Filters’ Encoded Length	This is the size in bytes of the encoded *I/O Filter Information*.
Flags	This field is the heap status flag and is a bit field indicating additional information about the fractal heap. **Bit(s) Description** 0 — If set, the ID value to use for huge object has wrapped around. If the value for the *Next Huge Object ID* has wrapped around, each new huge object inserted into the heap will require a search for an ID value. 1 — If set, the direct blocks in the heap are checksummed. 2-7 — Reserved
Maximum Size of Managed Objects	This is the maximum size of managed objects allowed in the heap. Objects greater than this this are ‘huge’ objects and will be stored in the file directly, rather than in a direct block for the heap.
Next Huge Object ID	This is the next ID value to use for a huge object in the heap.
v2 B-tree Address of Huge Objects	This is the address of the v2 B-tree used to track huge objects in the heap. The type of records stored in the *v2 B-tree* will be determined by whether the address and length of a huge object can fit into a heap ID (if yes, it is a “directly” accessed huge object) and whether there is a filter used on objects in the heap.

Amount of Free Space in Managed Blocks	This is the total amount of free space in managed direct blocks (in bytes).
Address of Managed Block Free Space Manager	This is the address of the *Free-space Manager* for managed blocks.
Amount of Managed Space in Heap	This is the total amount of managed space in the heap (in bytes), essentially the upper bound of the heap's linear address space.
Amount of Allocated Managed Space in Heap	This is the total amount of managed space (in bytes) actually allocated in the heap. This can be less than the *Amount of Managed Space in Heap* field, if some direct blocks in the heap's linear address space are not allocated.
Offset of Direct Block Allocation Iterator in Managed Space	This is the linear heap offset where the next direct block should be allocated at (in bytes). This may be less than the *Amount of Managed Space in Heap* value because the heap's address space is increased by a "row" of direct blocks at a time, rather than by single direct block increments.
Number of Managed Objects in Heap	This is the number of managed objects in the heap.
Size of Huge Objects in Heap	This is the total size of huge objects in the heap (in bytes).
Number of Huge Objects in Heap	This is the number of huge objects in the heap.
Size of Tiny Objects in Heap	This is the total size of tiny objects that are packed in heap IDs (in bytes).
Number of Tiny Objects in Heap	This is the number of tiny objects that are packed in heap IDs.
Table Width	This is the number of columns in the doubling table for managed blocks. This value must be a power of two.
Starting Block Size	This is the starting block size to use in the doubling table for managed blocks (in bytes). This value must be a power of two.
Maximum Direct Block Size	This is the maximum size allowed for a managed direct block. Objects inserted into

	the heap that are larger than this value (less the number of bytes of direct block prefix/suffix) are stored as 'huge' objects. This value must be a power of two.
Maximum Heap Size	This is the maximum size of the heap's linear address space for managed objects (in bytes). The value stored is the log2 of the actual value, that is: the number of bits of the address space. 'Huge' and 'tiny' objects are not counted in this value, since they do not store objects in the linear address space of the heap.
Starting # of Rows in Root Indirect Block	This is the starting number of rows for the root indirect block. A value of 0 indicates that the root indirect block will have the maximum number of rows needed to address the heap's *Maximum Heap Size*.
Address of Root Block	This is the address of the root block for the heap. It can be the undefined address if there is no data in the heap. It either points to a direct block (if the *Current # of Rows in the Root Indirect Block* value is 0), or an indirect block.
Current # of Rows in Root Indirect Block	This is the current number of rows in the root indirect block. A value of 0 indicates that *Address of Root Block* points to direct block instead of indirect block.
Size of Filtered Root Direct Block	This is the size of the root direct block, if filters are applied to heap objects (in bytes). This field is only stored in the header if the *I/O Filters' Encoded Length* is greater than 0.
I/O Filter Mask	This is the filter mask for the root direct block, if filters are applied to heap objects. This mask has the same format as that used for the filter mask in chunked raw data records in a v1 B-tree. This field is only stored in the header if the *I/O Filters' Encoded Length* is greater than 0.
I/O Filter Information	This is the I/O filter information encoding direct blocks and huge objects, if filters are applied to heap objects. This field is encoded as a Filter Pipeline message. The size of this field is determined by *I/O Filters'*

	Encoded Length.
Checksum	This is the checksum for the header.

Layout: Fractal Heap Direct Block

byte	byte	byte	byte
Signature			
Version	*This space inserted only to align table nicely*		
Heap Header Address[O]			
Block Offset *(variable size)*			
Checksum *(optional)*			
Object Data *(variable size)*			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Fractal Heap Direct Block

Field Name	Description
Signature	The ASCII character string “FHDB” is used to indicate the beginning of a fractal heap direct block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Heap Header Address	This is the address for the fractal heap header that this block belongs to. This field is principally used for file integrity checking.
Block Offset	This is the offset of the block within the fractal heap’s address space (in bytes). The number of bytes used to encode this field is the *Maximum Heap Size* (in the heap’s header) divided by 8 and rounded up to the next highest integer, for values that are not a multiple of 8. This value is principally used for file integrity checking.
Checksum	This is the checksum for the direct block. This field is only present if bit 1 of *Flags* in the heap’s header is set.
Object Data	This section of the direct block stores the actual data for objects in the heap. The size of this section is determined by the direct block’s size minus the size of the other fields stored in the direct block (for example, the *Signature*, *Version*, and others including the *Checksum* if it is present).

Layout: Fractal Heap Indirect Block

byte	byte	byte	byte
Signature			
Version	*This space inserted only to align table nicely*		
Heap Header Address[O]			
Block Offset *(variable size)*			
Child Direct Block #0 Address[O]			
Size of Filtered Direct Block #0 *(optional)* [L]			
Filter Mask for Direct Block #0 *(optional)*			
Child Direct Block #1 Address[O]			
Size of Filtered Direct Block #1 *(optional)*[L]			
Filter Mask for Direct Block #1 *(optional)*			
...			
Child Direct Block #K-1 Address[O]			
Size of Filtered Direct Block #K-1 *(optional)*[L]			
Filter Mask for Direct Block #K-1 *(optional)*			
Child Indirect Block #0 Address[O]			
Child Indirect Block #1 Address[O]			
...			
Child Indirect Block #N-1 Address[O]			

Checksum

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Fractal Heap Indirect Block

Field Name	Description
Signature	The ASCII character string “FHIB” is used to indicate the beginning of a fractal heap indirect block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Heap Header Address	This is the address for the fractal heap header that this block belongs to. This field is principally used for file integrity checking.
Block Offset	This is the offset of the block within the fractal heap’s address space (in bytes). The number of bytes used to encode this field is the *Maximum Heap Size* (in the heap’s header) divided by 8 and rounded up to the next highest integer, for values that are not a multiple of 8. This value is principally used for file integrity checking.
Child Direct Block #K Address	This field is the address of the child direct block. The size of the [uncompressed] direct block can be computed by its offset in the heap’s linear address space.
Size of Filtered Direct Block #K	This is the size of the child direct block after passing through the I/O filters defined for this heap (in bytes). If no I/O filters are present for this heap, this field is not present.
Filter Mask for Direct Block #K	This is the I/O filter mask for the filtered direct block. This mask has the same format as that used for the filter mask in chunked raw data records in a v1 B-tree. If no I/O filters are present for this heap, this field is not present.
Child Indirect Block #N Address	This field is the address of the child indirect block. The size of the indirect block can be computed by its offset in the heap’s linear address space.
Checksum	This is the checksum for the indirect block.

An object in the fractal heap is identified by means of a fractal heap ID, which encodes information to locate the object in the heap. Currently, the fractal heap stores an object in one of three ways, depending on the object’s size:

Type	Description
Tiny	When an object is small enough to be encoded in the heap ID, the object's data is embedded in the fractal heap ID itself. There are two sub-types for this type of object: normal and extended. The sub-type for tiny heap IDs depends on whether the heap ID is large enough to store objects greater than 16 bytes or not. If the heap ID length is 18 bytes or smaller, the 'normal' tiny heap ID form is used. If the heap ID length is greater than 18 bytes in length, the "extended" form is used. See the format description below for both sub-types.
Huge	When the size of an object is larger than *Maximum Size of Managed Objects* in the *Fractal Heap Header*, the object's data is stored on its own in the file and the object is tracked/indexed via a version 2 B-tree. All huge objects for a particular fractal heap use the same v2 B-tree. All huge objects for a particular fractal heap use the same format for their huge object IDs. Depending on whether the IDs for a heap are large enough to hold the object's retrieval information and whether I/O pipeline filters are applied to the heap's objects, 4 sub-types are derived for huge object IDs for this heap:

Sub-type	Description
Directly accessed, non-filtered	The object's address and length are embedded in the fractal heap ID itself and the object is directly accessed from them. This allows the object to be accessed without resorting to the B-tree.
Directly accessed, filtered	The filtered object's address, length, filter mask and de-filtered size are embedded in the fractal heap ID itself and the object is accessed directly with them. This allows the object to be accessed without resorting to the B-tree.
Indirectly accessed, non-filtered	The object is located by using a B-tree key embedded in the fractal heap ID to retrieve the address and length from the version 2 B-tree for huge objects. Then, the address and length are used to access the object.
Indirectly accessed, filtered	The object is located by using a B-tree key embedded in the fractal heap ID to retrieve the filtered object's address, length, filter mask and de-filtered size from the version 2 B-tree for huge objects. Then, this information is used to access the object.

Managed — When the size of an object does not meet the above two conditions, the object is stored and managed via the direct and indirect blocks based on the doubling table.

The specific format for each type of heap ID is described below:

Layout: Fractal Heap ID for Tiny Objects (Sub-type 1 - 'Normal')

byte	byte	byte	byte
Version, Type, and Length	*This space inserted only to align table nicely*		
Data *(variable size)*			

Fields: Fractal Heap ID for Tiny Objects (Sub-type 1 - ‘Normal’)

Field Name	Description
Version, Type, and Length	This is a bit field with the following definition: **Bit** / **Description** 6-7: The current version of ID format. This document describes version 0. 4-5: The ID type. Tiny objects have a value of 2. 0-3: The length of the tiny object. The value stored is one less than the actual length (since zero-length objects are not allowed to be stored in the heap). For example, an object of actual length 1 has an encoded length of 0, an object of actual length 2 has an encoded length of 1, and so on.
Data	This is the data for the object.

Layout: Fractal Heap ID for Tiny Objects (Sub-type 2 - ‘Extended’)

byte	byte	byte	byte
Version, Type, and Length	Extended Length	*This space inserted only to align table nicely*	
Data *(variable size)*			

Fields: Fractal Heap ID for Tiny Objects (Sub-type 2 - ‘Extended’)

Field Name	Description
Version, Type, and Length	This is a bit field with the following definition: **Bit** / **Description** 6-7: The current version of ID format. This document describes version 0. 4-5: The ID type. Tiny objects have a value of 2. 0-3: These 4 bits, together with the next byte, form an unsigned 12-bit integer for holding the length of the object. These 4-bits are bits 8-11 of the 12-bit integer. See description for the *Extended Length* field below.
Extended Length	This byte, together with the 4 bits in the previous byte, forms an unsigned 12-bit integer for holding the length of the tiny object. These 8 bits are bits 0-7 of the 12-bit integer formed. The value stored is one less than the actual length (since zero-length objects are not allowed to be stored in the heap). For example, an object of actual length 1 has an encoded length of 0, an object of actual length 2 has an encoded length of 1, and so on.
Data	This is the data for the object.

Layout: Fractal Heap ID for Huge Objects (Sub-types 1 and 2): Indirectly Accessed, Non-filtered/Filtered

byte	byte	byte	byte
Version and Type	*This space inserted only to align table nicely*		
v2 B-tree Key^L *(variable size)*			

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Fractal Heap ID for Huge Objects (Sub-types 1 and 2): Indirectly Accessed, Non-filtered/Filtered

Field Name	Description
Version and Type	This is a bit field with the following definition: **Bit** / **Description** 6-7: The current version of ID format. This document describes version 0. 4-5: The ID type. Huge objects have a value of 1. 0-3: Reserved.
v2 B-tree Key	This field is the B-tree key for retrieving the information from the version 2 B-tree for huge objects needed to access the object. See the description of v2 B-tree records sub-types 1 and 2 for a description of the fields. New key values are derived from *Next Huge Object ID* in the *Fractal Heap Header*.

Layout: Fractal Heap ID for Huge Objects (Sub-type 3): Directly Accessed, Non-filtered

byte	byte	byte	byte
Version and Type	*This space inserted only to align table nicely*		
Address [O]			
Length [L]			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Fractal Heap ID for Huge Objects (Sub-type 3): Directly Accessed, Non-filtered

Field Name	Description
Version and Type	This is a bit field with the following definition: **Bit** / **Description** 6-7: The current version of ID format. This document describes version 0. 4-5: The ID type. Huge objects have a value of 1. 0-3: Reserved.
Address	This field is the address of the object in the file.
Length	This field is the length of the object in the file.

Layout: Fractal Heap ID for Huge Objects (Sub-type 4): Directly Accessed, Filtered

byte	byte	byte	byte
Version and Type	*This space inserted only to align table nicely*		
Address [O]			
Length [L]			
Filter Mask			
De-filtered Size [L]			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Fractal Heap ID for Huge Objects (Sub-type 4): Directly Accessed, Filtered

Field Name	Description
Version and Type	This is a bit field with the following definition: **Bit** / **Description** 6-7: The current version of ID format. This document describes version 0. 4-5: The ID type. Huge objects have a value of 1. 0-3: Reserved.
Address	This field is the address of the filtered object in the file.
Length	This field is the length of the filtered object in the file.
Filter Mask	This field is the I/O pipeline filter mask for the filtered object in the file.
Filtered Size	This field is the size of the de-filtered object in the file.

Layout: Fractal Heap ID for Managed Objects

byte	byte	byte	byte
Version and Type	*This space inserted only to align table nicely*		
Offset *(variable size)*			
Length *(variable size)*			

Fields: Fractal Heap ID for Managed Objects

Field Name	Description
Version and Type	This is a bit field with the following definition: **Bit** **Description** 6-7 The current version of ID format. This document describes version 0. 4-5 The ID type. Managed objects have a value of 0. 0-3 Reserved.
Offset	This field is the offset of the object in the heap. This field's size is the minimum number of bytes necessary to encode the *Maximum Heap Size* value (from the *Fractal Heap Header*). For example, if the value of the *Maximum Heap Size* is less than 256 bytes, this field is 1 byte in length, a *Maximum Heap Size* of 256-65535 bytes uses a 2 byte length, and so on.
Length	This field is the length of the object in the heap. It is determined by taking the minimum value of *Maximum Direct Block Size* and *Maximum Size of Managed Objects* in the *Fractal Heap Header*. Again, the minimum number of bytes needed to encode that value is used for the size of this field.

III.H. Disk Format: Level 1H - Free-space Manager

Free-space managers are used to describe space within a heap or the entire HDF5 file that is not currently used for that heap or file.

The *free-space manager header* contains metadata information about the space being tracked, along with the address of the list of *free space sections* which actually describes the free space. The header records information about free-space sections being tracked, creation parameters for handling free-space sections of a client, and section information used to locate the collection of free-space sections.

The *free-space section list* stores a collection of free-space sections that is specific to each *client* of the free-space manager. For example, the fractal heap is a client of the free space manager and uses it to track unused space within the heap. There are 4 types of section records for the fractal heap, each of which has its own format, listed below.

Layout: Free-space Manager Header

byte	byte	byte	byte
Signature			
Version	Client ID	*This space inserted only to align table nicely*	
Total Space Tracked[L]			
Total Number of Sections[L]			
Number of Serialized Sections[L]			
Number of Un-Serialized Sections[L]			
Number of Section Classes		*This space inserted only to align table nicely*	
Shrink Percent		Expand Percent	
Size of Address Space		*This space inserted only to align table nicely*	
Maximum Section Size [L]			
Address of Serialized Section List[O]			
Size of Serialized Section List Used[L]			
Allocated Size of Serialized Section List[L]			
Checksum			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Free-space Manager Header

Field Name	Description
Signature	The ASCII character string "FSHD" is used to indicate the beginning of the Free-space Manager Header. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This is the version number for the Free-space Manager Header and this document describes version 0.
Client ID	This is the client ID for identifying the user of this free-space manager: **ID** **Description** 0 Fractal heap 1 File 2+ Reserved.
Total Space Tracked	This is the total amount of free space being tracked, in bytes.
Total Number of Sections	This is the total number of free-space sections being tracked.
Number of Serialized Sections	This is the number of serialized free-space sections being tracked.
Number of Un-Serialized Sections	This is the number of un-serialized free-space sections being managed. Un-serialized sections are created by the free-space client when the list of sections is read in.
Number of Section Classes	This is the number of section classes handled by this free space manager for the free-space client.
Shrink Percent	This is the percent of current size to shrink the allocated serialized free-space section list.
Expand Percent	This is the percent of current size to expand the allocated serialized free-space section list.
Size of Address Space	This is the size of the address space that free-space sections are within. This is stored as the $\log_2$ of the actual value (in other words, the number of bits required to store values within

	that address space).
Maximum Section Size	This is the maximum size of a section to be tracked.
Address of Serialized Section List	This is the address where the serialized free-space section list is stored.
Size of Serialized Section List Used	This is the size of the serialized free-space section list used (in bytes). This value must be less than or equal to the *allocated size of serialized section list*, below.
Allocated Size of Serialized Section List	This is the size of serialized free-space section list actually allocated (in bytes).
Checksum	This is the checksum for the free-space manager header.

The free-space sections being managed are stored in a *free-space section list*, described below. The sections in the free-space section list are stored in the following way: a count of the number of sections describing a particular size of free space and the size of the free-space described (in bytes), followed by a list of section description records; then another section count and size, followed by the list of section descriptions for that size; and so on.

Layout: Free-space Section List

byte	byte	byte	byte
Signature			
Version	*This space inserted only to align table nicely*		
Free-space Manager Header Address[O]			
Number of Section Records in Set #0 *(variable size)*			
Size of Free-space Section Described in Record Set #0 *(variable size)*			
Record Set #0 Section Record #0 Offset*(variable size)*			
Record Set #0 Section Record #0 Type	*This space inserted only to align table nicely*		
Record Set #0 Section Record #0 Data *(variable size)*			
...			
Record Set #0 Section Record #K-1 Offset*(variable size)*			
Record Set #0 Section Record #K-1 Type	*This space inserted only to align table nicely*		
Record Set #0 Section Record #K-1 Data *(variable size)*			
Number of Section Records in Set #1 *(variable size)*			
Size of Free-space Section Described in Record Set #1 *(variable size)*			
Record Set #1 Section Record #0 Offset*(variable size)*			
Record Set #1 Section Record #0 Type	*This space inserted only to align table nicely*		
Record Set #1 Section Record #0 Data *(variable size)*			
...			
Record Set #1 Section Record #K-1 Offset*(variable size)*			
Record Set #1 Section Record #K-1 Type	*This space inserted only to align table nicely*		
Record Set #1 Section Record #K-1 Data *(variable size)*			
...			
...			
Number of Section Records in Set #N-1 *(variable size)*			

Size of Free-space Section Described in Record Set #N-1 *(variable size)*	
Record Set #N-1 Section Record #0 Offset*(variable size)*	
Record Set #N-1 Section Record #0 Type	*This space inserted only to align table nicely*
Record Set #N-1 Section Record #0 Data *(variable size)*	
...	
Record Set #N-1 Section Record #K-1 Offset*(variable size)*	
Record Set #N-1 Section Record #K-1 Type	*This space inserted only to align table nicely*
Record Set #N-1 Section Record #K-1 Data *(variable size)*	
Checksum	

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Free-space Section List

Field Name	Description
Signature	The ASCII character string “FSSE” is used to indicate the beginning of the Free-space Section Information. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This is the version number for the Free-space Section List and this document describes version 0.
Free-space Manager Header Address	This is the address of the *Free-space Manager Header*. This field is principally used for file integrity checking.
Number of Section Records for Set #N	This is the number of free-space section records for set #N. The length of this field is the minimum number of bytes needed to store the *number of serialized sections* (from the *free-space manager header*). The number of sets of free-space section records is determined by the *size of serialized section list* in the *free-space manager header*.
Section Size for Record Set #N	This is the size (in bytes) of the free-space section described for *all* the section records in set #N. The length of this field is the minimum number of bytes needed to store the *maximum section size* (from the *free-space manager header*).
Record Set #N Section #K Offset	This is the offset (in bytes) of the free-space section within the client for the free-space manager. The length of this field is the minimum number of bytes needed to store the *size of address space* (from the *free-space manager header*).
Record Set #N Section #K Type	This is the type of the section record, used to decode the *record set #N section #K data* information. The defined record type for *file* client is: **Type** — **Description** 0 — File’s section (a range of actual bytes in file) 1+ — Reserved.

	The defined record types for a *fractal heap* client are: **Type** **Description** 0 Fractal heap “single” section 1 Fractal heap “first row” section 2 Fractal heap “normal row” section 3 Fractal heap “indirect” section 4+ Reserved.
Record Set #N Section #K Data	This is the section-type specific information for each record in the record set, described below.
Checksum	This is the checksum for the *Free-space Section List*.

The section-type specific data for each free-space section record is described below:

Layout: File’s Section Data Record

No additional record data stored

Layout: Fractal Heap “Single” Section Data Record

No additional record data stored

Layout: Fractal Heap “First Row” Section Data Record

Same format as “indirect” section data

Layout: Fractal Heap “Normal Row” Section Data Record

No additional record data stored

Layout: Fractal Heap “Indirect” Section Data Record

byte	byte	byte	byte
Fractal Heap Indirect Block Offset *(variable size)*			
Block Start Row		Block Start Column	
Number of Blocks		*This space inserted only to align table nicely*	

Fields: Fractal Heap “Indirect” Section Data Record

Field Name	Description
Fractal Heap Block Offset	The offset of the indirect block in the fractal heap’s address space containing the empty blocks. The number of bytes used to encode this field is the minimum number of bytes needed to encode values for the *Maximum Heap Size* (in the fractal heap’s header).
Block Start Row	This is the row that the empty blocks start in.
Block Start Column	This is the column that the empty blocks start in.
Number of Blocks	This is the number of empty blocks covered by the section.

III.I. Disk Format: Level 1I - Shared Object Header Message Table

The *shared object header message table* is used to locate object header messages that are shared between two or more object headers in the file. Shared object header messages are stored and indexed in the file in one of two ways: indexed sequentially in a *shared header message list* or indexed with a v2 B-tree. The shared messages themselves are either stored in a fractal heap (when two or more objects share the message), or remain in an object’s header (when only one object uses the message currently, but the message can be shared in the future).

The *shared object header message table* contains a list of shared message index headers. Each index header records information about the version of the index format, the index storage type, flags for the message types indexed, the number of messages in the index, the address where the index resides, and the fractal heap address if shared messages are stored there.

Each index can be either a list or a v2 B-tree and may transition between those two forms as the number of messages in the index varies. Each shared message record contains information used to locate the shared message from either a fractal heap or an object header. The types of messages that can be shared are: *Dataspace, Datatype, Fill Value, Filter Pipeline and Attribute*.

The *shared object header message table* is pointed to from a shared message table message in the superblock extension for a file. This message stores the version of the table format, along with the number of index headers in the table.

Layout: Shared Object Header Message Table

byte	byte	byte	byte
Signature			
Version for index #0	Index Type for index #0	Message Type Flags for index #0	
Minimum Message Size for index #0			
List Cutoff for index #0		v2 B-tree Cutoff for index #0	
Number of Messages for index #0		*This space inserted only to align table nicely*	
Index Address^O for index #0			
Fractal Heap Address^O for index #0			
...			
...			
Version for index #N-1	Index Type for index #N-1	Message Type Flags for index #N-1	
Minimum Message Size for index #N-1			
List Cutoff for index #N-1		v2 B-tree Cutoff for index #N-1	
Number of Messages for index #N-1		*This space inserted only to align table nicely*	
Index Address^O for index #N-1			
Fractal Heap Address^O for index #N-1			
Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Shared Object Header Message Table

Field Name	Description
Signature	The ASCII character string “SMTB” is used to indicate the beginning of the Shared Object Header Message table. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version for index #N	This is the version number for the list of shared object header message indexes and this document describes version 0.
Index Type for index #N	The type of index can be an unsorted list or a v2 B-tree.
Message Type Flags for index #N	This field indicates the type of messages tracked in the index, as follows: **Bits** **Description** 0 If set, the index tracks *Dataspace Messages*. 1 If set, the message tracks *Datatype Messages*. 2 If set, the message tracks *Fill Value Messages*. 3 If set, the message tracks *Filter Pipeline Messages*. 4 If set, the message tracks *Attribute Messages*. 5-15 Reserved (zero). An index can track more than one type of message, but each type of message can only by in one index.
Minimum Message Size for index #N	This is the message size sharing threshold for the index. If the encoded size of the message is less than this value, the message is not shared.
List Cutoff for index #N	This is the cutoff value for the indexing of messages to switch from a list to a v2 B-tree. If the number of messages is greater than this value, the index should be a v2 B-tree.
v2 B-tree Cutoff for index #N	This is the cutoff value for the indexing of messages to switch from a v2 B-tree back to a list. If the number of messages is less than this value, the index should be a list.

Number of Messages for index #N	The number of shared messages being tracked for the index.
Index Address for index #N	This field is the address of the list or v2 B-tree where the index nodes reside.
Fractal Heap Address for index #N	This field is the address of the fractal heap if shared messages are stored there.
Checksum	This is the checksum for the table.

Shared messages are indexed either with a *shared message record list*, described below, or using a v2 B-tree (using record type 7). The number of records in the *shared message record list* is determined in the index's entry in the *shared object header message table*.

Layout: Shared Message Record List

byte	byte	byte	byte
Signature			
Shared Message Record #0			
Shared Message Record #1			
...			
Shared Message Record #N-1			
Checksum			

Fields: Shared Message Record List

Field Name	Description
Signature	The ASCII character string “SMLI” is used to indicate the beginning of a list of index nodes. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Shared Message Record #N	The record for locating the shared message, either in the fractal heap for the index, or an object header (see format for *index nodes* below).
Checksum	This is the checksum for the list.

The record for each shared message in an index is stored in one of the following forms:

Layout: Shared Message Record for Messages Stored in a Fractal Heap

byte	byte	byte	byte
Message Location	*This space inserted only to align table nicely*		
Hash Value			
Reference Count			
Fractal Heap ID			

Fields: Shared Message Record for Messages Stored in a Fractal Heap

Field Name	Description
Message Location	This has a value of 0 indicating that the message is stored in the heap.
Hash Value	This is the hash value for the message.
Reference Count	This is the number of times the message is used in the file.
Fractal Heap ID	This is an 8-byte fractal heap ID for the message as stored in the fractal heap for the index.

Layout: Shared Message Record for Messages Stored in an Object Header

byte	byte	byte	byte
Message Location	*This space inserted only to align table nicely*		
Hash Value			
Reserved	Message Type	Creation Index	
Object Header Address^{O}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Shared Message Record for Messages Stored in an Object Header

Field Name	Description
Message Location	This has a value of 1 indicating that the message is stored in an object header.
Hash Value	This is the hash value for the message.
Message Type	This is the message type in the object header.
Creation Index	This is the creation index of the message within the object header.
Object Header Address	This is the address of the object header where the message is located.

IV. Disk Format: Level 2 - Data Objects

Data objects contain the “real” user-visible information in the file. These objects compose the scientific data and other information which are generally thought of as “data” by the end-user. All the other information in the file is provided as a framework for storing and accessing these data objects.

A data object is composed of header and data information. The header information contains the information needed to interpret the data information for the object as well as additional “metadata” or pointers to additional “metadata” used to describe or annotate each object.

IV.A. Disk Format: Level 2A - Data Object Headers

The header information of an object is designed to encompass all of the information about an object, except for the data itself. This information includes the dataspace, the datatype, information about how the data is stored on disk (in external files, compressed, broken up in blocks, and so on), as well as other information used by the library to speed up access to the data objects or maintain a file’s integrity. Information stored by user applications as attributes is also stored in the object’s header. The header of each object is not necessarily located immediately prior to the object’s data in the file and in fact may be located in any position in the file. The order of the messages in an object header is not significant.

Object headers are composed of a prefix and a set of messages. The prefix contains the information needed to interpret the messages and a small amount of metadata about the object, and the messages contain the majority of the metadata about the object.

IV.A.1. Disk Format: Level 2A1 - Data Object Header Prefix

IV.A.1.a. Version 1 Data Object Header Prefix

Header messages are aligned on 8-byte boundaries for version 1 object headers.

Layout: Version 1 Object Header

byte	byte	byte	byte
Version	Reserved (zero)	Total Number of Header Messages	
Object Reference Count			
Object Header Size			
Header Message Type #1		Size of Header Message Data #1	
Header Message #1 Flags	Reserved (zero)		
Header Message Data #1			
. . .			
Header Message Type #n		Size of Header Message Data #n	
Header Message #n Flags	Reserved (zero)		
Header Message Data #n			

Fields: Version 1 Object Header

Field Name	Description
Version	This value is used to determine the format of the information in the object header. When the format of the object header is changed, the version number is incremented and can be used to determine how the information in the object header is formatted. This is version one (1) (there was no version zero (0)) of the object header.
Total Number of Header Messages	This value determines the total number of messages listed in object headers for this object. This value includes the messages in continuation messages for this object.
Object Reference Count	This value specifies the number of “hard links” to this object within the current file. References to the object from external files, “soft links” in this file and object references in this file are not tracked.
Object Header Size	This value specifies the number of bytes of header message data following this length field that contain object header messages for this object header. This value does not include the size of object header continuation blocks for this object elsewhere in the file.
Header Message #n Type	This value specifies the type of information included in the following header message data. The message types for header messages are defined in sections below.
Size of Header Message #n Data	This value specifies the number of bytes of header message data following the header message type and length information for the current message. The size includes padding bytes to make the message a multiple of eight bytes.
Header Message #n Flags	This is a bit field with the following definition: **Bit** — **Description** 0 — If set, the message data is constant. This is used for messages like the datatype message of a dataset. 1 — If set, the message is *shared* and stored in another location than the object header. The Header Message Data field contains a Shared Message (described in the Data Object Header Messages section below) and the Size of Header

	Message Data field contains the size of that Shared Message. 2 If set, the message should not be shared. 3 If set, the HDF5 decoder should fail to open this object if it does not understand the message's type and the file is open with permissions allowing write access to the file. (Normally, unknown messages can just be ignored by HDF5 decoders) 4 If set, the HDF5 decoder should set bit 5 of this message's flags (in other words, this bit field) if it does not understand the message's type and the object is modified in any way. (Normally, unknown messages can just be ignored by HDF5 decoders) 5 If set, this object was modified by software that did not understand this message. (Normally, unknown messages should just be ignored by HDF5 decoders) (Can be used to invalidate an index or a similar feature) 6 If set, this message is shareable. 7 If set, the HDF5 decoder should always fail to open this object if it does not understand the message's type (whether it is open for read-only or read-write access). (Normally, unknown messages can just be ignored by HDF5 decoders)
Header Message #n Data	The format and length of this field is determined by the header message type and size respectively. Some header message types do not require any data and this information can be eliminated by setting the length of the message to zero. The data is padded with enough zeroes to make the size a multiple of eight.

IV.A.1.b. Version 2 Data Object Header Prefix

Note that the "total number of messages" field has been dropped from the data object header prefix in this version. The number of messages in the data object header is just determined by the messages encountered in all the object header blocks.

Note also that the fields and messages in this version of data object headers have *no* alignment or padding bytes inserted - they are stored packed together.

Layout: Version 2 Object Header

byte	byte	byte	byte
Signature			
Version	Flags	*This space inserted only to align table nicely*	
Access time *(optional)*			
Modification Time *(optional)*			
Change Time *(optional)*			
Birth Time *(optional)*			
Maximum # of compact attributes *(optional)*		Minimum # of dense attributes *(optional)*	
Size of Chunk #0 *(variable size)*	*This space inserted only to align table nicely*		
Header Message Type #1	Size of Header Message Data #1		Header Message #1 Flags
Header Message #1 Creation Order *(optional)*		*This space inserted only to align table nicely*	
Header Message Data #1			
. . .			
Header Message Type #n	Size of Header Message Data #n		Header Message #n Flags
Header Message #n Creation Order *(optional)*		*This space inserted only to align table nicely*	
Header Message Data #n			
Gap *(optional, variable size)*			
Checksum			

Fields: Version 2 Object Header

Field Name	Description
Signature	The ASCII character string “OHDR” is used to indicate the beginning of an object header. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This field has a value of 2 indicating version 2 of the object header.
Flags	This field is a bit field indicating additional information about the object header. **Bit(s) — Description** 0-1 — This two bit field determines the size of the *Size of Chunk #0* field. The values are: **Value — Description** 0 — The *Size of Chunk #0* field is 1 byte. 1 — The *Size of Chunk #0* field is 2 bytes. 2 — The *Size of Chunk #0* field is 4 bytes. 3 — The *Size of Chunk #0* field is 8 bytes. 2 — If set, attribute creation order is tracked. 3 — If set, attribute creation order is indexed. 4 — If set, non-default attribute storage phase change values are stored. 5 — If set, access, modification, change and birth times are stored. 6-7 — Reserved
Access Time	This 32-bit value represents the number of seconds after the UNIX epoch when the object’s raw data was last accessed (in other words, read or written). This field is present if bit 5 of *flags* is set.
Modification Time	This 32-bit value represents the number of seconds after the UNIX epoch when the object’s raw data was last modified (in other words, written). This field is present if bit 5 of *flags* is set.
Change Time	This 32-bit value represents the number of seconds after the UNIX epoch when the object’s metadata

	was last changed. This field is present if bit 5 of *flags* is set.
Birth Time	This 32-bit value represents the number of seconds after the UNIX epoch when the object was created. This field is present if bit 5 of *flags* is set.
Maximum # of compact attributes	This is the maximum number of attributes to store in the compact format before switching to the indexed format. This field is present if bit 4 of *flags* is set.
Minimum # of dense attributes	This is the minimum number of attributes to store in the indexed format before switching to the compact format. This field is present if bit 4 of *flags* is set.
Size of Chunk #0	This unsigned value specifies the number of bytes of header message data following this field that contain object header information. This value does not include the size of object header continuation blocks for this object elsewhere in the file. The length of this field varies depending on bits 0 and 1 of the *flags* field.
Header Message #n Type	Same format as version 1 of the object header, described above.
Size of Header Message #n Data	This value specifies the number of bytes of header message data following the header message type and length information for the current message. The size of messages in this version does *not* include any padding bytes.
Header Message #n Flags	Same format as version 1 of the object header, described above.
Header Message #n Creation Order	This field stores the order that a message of a given type was created in. This field is present if bit 2 of *flags* is set.
Header Message #n Data	Same format as version 1 of the object header, described above.
Gap	A gap in an object header chunk is inferred by the end of the messages for the chunk before the beginning of the chunk’s checksum. Gaps are

	always smaller than the size of an object header message prefix (message type + message size + message flags). Gaps are formed when a message (typically an attribute message) in an earlier chunk is deleted and a message from a later chunk that does not quite fit into the free space is moved into the earlier chunk.
Checksum	This is the checksum for the object header chunk.

The header message types and the message data associated with them compose the critical “metadata” about each object. Some header messages are required for each object while others are optional. Some optional header messages may also be repeated several times in the header itself, the requirements and number of times allowed in the header will be noted in each header message description below.

IV.A.2. Disk Format: Level 2A2 - Data Object Header Messages

Data object header messages are small pieces of metadata that are stored in the data object header for each object in an HDF5 file. Data object header messages provide the metadata required to describe an object and its contents, as well as optional pieces of metadata that annotate the meaning or purpose of the object.

Data object header messages are either stored directly in the data object header for the object or are shared between multiple objects in the file. When a message is shared, a flag in the *Message Flags* indicates that the actual *Message Data* portion of that message is stored in another location (such as another data object header, or a heap in the file) and the *Message Data* field contains the information needed to locate the actual information for the message.

The format of shared message data is described here:

Layout: Shared Message (Version 1)

byte	byte	byte	byte
Version	Type	Reserved (zero)	
Reserved (zero)			
Address^{O}			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Shared Message (Version 1)

Field Name	Description
Version	The version number is used when there are changes in the format of a shared object message and is described here: **Version** / **Description** 0 — Never used. 1 — Used by the library before version 1.6.1.
Type	The type of shared message location: **Value** / **Description** 0 — Message stored in another object's header (a *committed* message).
Address	The address of the object header containing the message to be shared.

Layout: Shared Message (Version 2)

byte	byte	byte	byte
Version	Type	*This space inserted only to align table nicely*	
Address^{O}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Shared Message (Version 2)

Field Name	Description
Version	The version number is used when there are changes in the format of a shared object message and is described here: **Version** **Description** 2 Used by the library of version 1.6.1 and after.
Type	The type of shared message location: **Value** **Description** 0 Message stored in another object's header (a *committed* message).
Address	The address of the object header containing the message to be shared.

Layout: Shared Message (Version 3)

byte	byte	byte	byte
Version	Type	*This space inserted only to align table nicely*	
Location *(variable size)*			

Fields: Shared Message (Version 3)

Field Name	Description
Version	The version number indicates changes in the format of shared object message and is described here: **Version** / **Description** 3 — Used by the library of version 1.8 and after. In this version, the *Type* field can indicate that the message is stored in the fractal heap.
Type	The type of shared message location: **Value** / **Description** 0 — Message is not shared and is not shareable. 1 — Message stored in file's *shared object header message* heap (a *shared* message). 2 — Message stored in another object's header (a *committed* message). 3 — Message stored is not shared, but is sharable.
Location	This field contains either a *Size of Offsets*-bytes address of the object header containing the message to be shared, or an 8-byte fractal heap ID for the message in the file's *shared object header message* heap.

The following is a list of currently defined header messages:

IV.A.2.a. The NIL Message

Header Message Name: NIL
Header Message Type: 0x0000
Length: Varies
Status: Optional; may be repeated.
Description: The NIL message is used to indicate a message which is to be ignored when reading the header messages for a data object. [Possibly one which has been deleted for some reason.]
Format of Data: Unspecified

IV.A.2.b. The Dataspace Message

Header Message Name: Dataspace

Header Message Type: 0x0001

Length: Varies according to the number of dimensions, as described in the following table.

Status: Required for dataset objects; may not be repeated.

Description: The dataspace message describes the number of dimensions (in other words, “rank”) and size of each dimension that the data object has. This message is only used for datasets which have a simple, rectilinear, array-like layout; datasets requiring a more complex layout are not yet supported.

Format of Data: See the tables below.

Layout: Dataspace Message - Version 1

byte	byte	byte	byte
Version	Dimensionality	Flags	Reserved
Reserved			
Dimension #1 Size[L]			
. . .			
Dimension #n Size[L]			
Dimension #1 Maximum Size[L] *(optional)*			
. . .			
Dimension #n Maximum Size[L] *(optional)*			
Permutation Index #1[L] *(optional)*			
. . .			
Permutation Index #n[L] *(optional)*			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Dataspace Message - Version 1

Field Name	Description
Version	This value is used to determine the format of the Dataspace Message. When the format of the information in the message is changed, the version number is incremented and can be used to determine how the information in the object header is formatted. This document describes version one (1) (there was no version zero (0)).
Dimensionality	This value is the number of dimensions that the data object has.
Flags	This field is used to store flags to indicate the presence of parts of this message. Bit 0 (the least significant bit) is used to indicate that maximum dimensions are present. Bit 1 is used to indicate that permutation indices are present.
Dimension #n Size	This value is the current size of the dimension of the data as stored in the file. The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Dimension #n Maximum Size	This value is the maximum size of the dimension of the data as stored in the file. This value may be the special "unlimited" size which indicates that the data may expand along this dimension indefinitely. If these values are not stored, the maximum size of each dimension is assumed to be the dimension's current size.
Permutation Index #n	This value is the index permutation used to map each dimension from the canonical representation to an alternate axis for each dimension. If these values are not stored, the first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.

Version 2 of the dataspace message dropped the optional permutation index value support, as it was never implemented in the HDF5 Library:

Layout: Dataspace Message - Version 2

byte	byte	byte	byte
Version	Dimensionality	Flags	Type
Dimension #1 Size[L]			
. . .			
Dimension #n Size[L]			
Dimension #1 Maximum Size[L] *(optional)*			
. . .			
Dimension #n Maximum Size[L] *(optional)*			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Dataspace Message - Version 2

Field Name	Description
Version	This value is used to determine the format of the Dataspace Message. This field should be '2' for version 2 format messages.
Dimensionality	This value is the number of dimensions that the data object has.
Flags	This field is used to store flags to indicate the presence of parts of this message. Bit 0 (the least significant bit) is used to indicate that maximum dimensions are present.
Type	This field indicates the type of the dataspace: **Value** **Description** 0 A *scalar* dataspace; in other words, a dataspace with a single, dimensionless element. 1 A *simple* dataspace; in other words, a dataspace with a rank greater than 0 and an appropriate number of dimensions. 2 A *null* dataspace; in other words, a dataspace with no elements.
Dimension #n Size	This value is the current size of the dimension of the data as stored in the file. The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Dimension #n Maximum Size	This value is the maximum size of the dimension of the data as stored in the file. This value may be the special "unlimited" size which indicates that the data may expand along this dimension indefinitely. If these values are not stored, the maximum size of each dimension is assumed to be the dimension's current size.

IV.A.2.c. The Link Info Message

Header Message Name: Link Info
Header Message Type: 0x002
Length: Varies
Status: Optional; may not be repeated.

Description: The link info message tracks variable information about the current state of the links for a “new style” group’s behavior. Variable information will be stored in this message and constant information will be stored in the Group Info message.

Format of Data: See the tables below.

Layout: Link Info

byte	byte	byte	byte
Version	Flags	*This space inserted only to align table nicely*	
Maximum Creation Index *(8 bytes, optional)*			
Fractal Heap Address[O]			
Address of v2 B-tree for Name Index[O]			
Address of v2 B-tree for Creation Order Index[O] *(optional)*			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Link Info

Field Name	Description
Version	The version number for this message. This document describes version 0.
Flags	This field determines various optional aspects of the link info message: **Bit** / **Description** 0 — If set, creation order for the links is tracked. 1 — If set, creation order for the links is indexed. 2-7 — Reserved
Maximum Creation Index	This 64-bit value is the maximum creation order index value stored for a link in this group. This field is present if bit 0 of *flags* is set.
Fractal Heap Address	This is the address of the fractal heap to store dense links. Each link stored in the fractal heap is stored as a Link Message. If there are no links in the group, or the group's links are stored "compactly" (as object header messages), this value will be the undefined address.
Address of v2 B-tree for Name Index	This is the address of the version 2 B-tree to index names of links. If there are no links in the group, or the group's links are stored "compactly" (as object header messages), this value will be the undefined address.
Address of v2 B-tree for Creation Order Index	This is the address of the version 2 B-tree to index creation order of links. If there are no links in the group, or the group's links are stored "compactly" (as object header messages), this value will be the undefined address. This field exists if bit 1 of *flags* is set.

IV.A.2.d. The Datatype Message

Header Message Name: Datatype

Header Message Type: 0x0003

Length: Variable

Status: Required for dataset or committed datatype (formerly named datatype) objects; may not be repeated.

Description: The datatype message defines the datatype for each element of a dataset or a common datatype for sharing between multiple datasets. A datatype can describe an atomic type like a fixed- or floating-point type or more complex types like a C struct (compound datatype), array (array datatype), or C++ vector (variable-length datatype).

Datatype messages that are part of a dataset object do not describe how elements are related to one another; the dataspace message is used for that purpose. Datatype messages that are part of a committed datatype (formerly named datatype) message describe a common datatype that can be shared by multiple datasets in the file.

Format of Data: See the tables below.

Layout: Datatype Message

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td>Class and Version</td><td>Class Bit Field, Bits 0-7</td><td>Class Bit Field, Bits 8-15</td><td>Class Bit Field, Bits 16-23</td></tr>
<tr><td colspan="4">Size</td></tr>
<tr><td colspan="4">Properties</td></tr>
</table>

Fields: Datatype Message

Field Name	Description
Class and Version	The version of the datatype message and the datatype's class information are packed together in this field. The version number is packed in the top 4 bits of the field and the class is contained in the bottom 4 bits. The version number information is used for changes in the format of the datatype message and is described here: **Version** — **Description** 0 — Never used 1 — Used by early versions of the library to encode compound datatypes with explicit array fields. See the compound datatype description below for further details. 2 — Used when an array datatype needs to be encoded. 3 — Used when a VAX byte-ordered type needs to be encoded. Packs various other datatype classes more efficiently also. The class of the datatype determines the format for the class bit field and properties portion of the datatype message, which are described below. The following classes are currently defined: **Value** — **Description** 0 — Fixed-Point 1 — Floating-point 2 — Time 3 — String 4 — Bit field 5 — Opaque 6 — Compound 7 — Reference 8 — Enumerated 9 — Variable-Length 10 — Array
Class Bit Fields	The information in these bit fields is specific to each datatype class and is described below. All bits not defined for a datatype class are set to zero.

Size	The size of a datatype element in bytes.
Properties	This variable-sized sequence of bytes encodes information specific to each datatype class and is described for each class below. If there is no property information specified for a datatype class, the size of this field is zero bytes.

Class specific information for the Fixed-point Numbers class (Class 0):

Bits: Fixed-point Bit Field Description

Bits	Meaning
0	**Byte Order.** If zero, byte order is little-endian; otherwise, byte order is big endian.
1, 2	**Padding type.** Bit 1 is the lo_pad bit and bit 2 is the hi_pad bit. If a datum has unused bits at either end, then the lo_pad or hi_pad bit is copied to those locations.
3	**Signed.** If this bit is set then the fixed-point number is in 2’s complement form.
4-23	Reserved (zero).

Layout: Fixed-point Property Description

<table>
<tr><th>Byte</th><th>Byte</th><th>Byte</th><th>Byte</th></tr>
<tr><td colspan="2">Bit Offset</td><td colspan="2">Bit Precision</td></tr>
</table>

Fields: Fixed-point Property Description

Field Name	Description
Bit Offset	The bit offset of the first significant bit of the fixed-point value within the datatype. The bit offset specifies the number of bits “to the right of” the value (which are set to the lo_pad bit value).
Bit Precision	The number of bits of precision of the fixed-point value within the datatype. This value, combined with the datatype element’s size and the Bit Offset field specifies the number of bits “to the left of” the value (which are set to the hi_pad bit value).

Class specific information for the Floating-point Numbers class (Class 1):

Bits: Floating-point Bit Field Description

Bits	Meaning
0, 6	**Byte Order.** These two non-contiguous bits specify the "endianness" of the bytes in the datatype element. **Bit 6** / **Bit 0** / **Description** 0 / 0 / Byte order is little-endian 0 / 1 / Byte order is big-endian 1 / 0 / Reserved 1 / 1 / Byte order is VAX-endian
1, 2, 3	**Padding type.** Bit 1 is the low bits pad type, bit 2 is the high bits pad type, and bit 3 is the internal bits pad type. If a datum has unused bits at either end or between the sign bit, exponent, or mantissa, then the value of bit 1, 2, or 3 is copied to those locations.
4-5	**Mantissa Normalization.** This 2-bit bit field specifies how the most significant bit of the mantissa is managed. **Value** / **Description** 0 / No normalization 1 / The most significant bit of the mantissa is always set (except for 0.0). 2 / The most significant bit of the mantissa is not stored, but is implied to be set. 3 / Reserved.
7	Reserved (zero).
8-15	**Sign Location.** This is the bit position of the sign bit. Bits are numbered with the least significant bit zero.
16-23	Reserved (zero).

Layout: Floating-point Property Description

Byte	Byte	Byte	Byte
Bit Offset		Bit Precision	
Exponent Location	Exponent Size	Mantissa Location	Mantissa Size
Exponent Bias			

Fields: Floating-point Property Description

Field Name	Description
Bit Offset	The bit offset of the first significant bit of the floating-point value within the datatype. The bit offset specifies the number of bits “to the right of” the value.
Bit Precision	The number of bits of precision of the floating-point value within the datatype.
Exponent Location	The bit position of the exponent field. Bits are numbered with the least significant bit number zero.
Exponent Size	The size of the exponent field in bits.
Mantissa Location	The bit position of the mantissa field. Bits are numbered with the least significant bit number zero.
Mantissa Size	The size of the mantissa field in bits.
Exponent Bias	The bias of the exponent field.

Class specific information for the Time class (Class 2):

Bits: Time Bit Field Description

Bits	Meaning
0	**Byte Order.** If zero, byte order is little-endian; otherwise, byte order is big endian.
1-23	Reserved (zero).

Layout: Time Property Description

Byte	Byte
Bit Precision	

Fields: Time Property Description

Field Name	Description
Bit Precision	The number of bits of precision of the time value.

Class specific information for the Strings class (Class 3):

Bits: String Bit Field Description

Bits	Meaning
0-3	**Padding type.** This four-bit value determines the type of padding to use for the string. The values are: **Value** / **Description** 0 — Null Terminate: A zero byte marks the end of the string and is guaranteed to be present after converting a long string to a short string. When converting a short string to a long string the value is padded with additional null characters as necessary. 1 — Null Pad: Null characters are added to the end of the value during conversions from short values to long values but conversion in the opposite direction simply truncates the value. 2 — Space Pad: Space characters are added to the end of the value during conversions from short values to long values but conversion in the opposite direction simply truncates the value. This is the Fortran representation of the string. 3-15 — Reserved
4-7	**Character Set.** The character set used to encode the string. **Value** / **Description** 0 — ASCII character set encoding 1 — UTF-8 character set encoding 2-15 — Reserved
8-23	Reserved (zero).

There are no properties defined for the string class.

Class specific information for the Bit Fields class (Class 4):

Bits: Bitfield Bit Field Description

Bits	Meaning
0	**Byte Order.** If zero, byte order is little-endian; otherwise, byte order is big endian.
1, 2	**Padding type.** Bit 1 is the lo_pad type and bit 2 is the hi_pad type. If a datum has unused bits at either end, then the lo_pad or hi_pad bit is copied to those locations.
3-23	Reserved (zero).

Layout: Bit Field Property Description

Byte	Byte	Byte	Byte
Bit Offset		Bit Precision	

Fields: Bit Field Property Description

Field Name	Description
Bit Offset	The bit offset of the first significant bit of the bit field within the datatype. The bit offset specifies the number of bits “to the right of” the value.
Bit Precision	The number of bits of precision of the bit field within the datatype.

Class specific information for the Opaque class (Class 5):

Bits: Opaque Bit Field Description

Bits	Meaning
0-7	Length of ASCII tag in bytes.
8-23	Reserved (zero).

Layout: Opaque Property Description

Byte	Byte	Byte	Byte
ASCII Tag			

Fields: Opaque Property Description

Field Name	Description
ASCII Tag	This NUL-terminated string provides a description for the opaque type. It is NUL-padded to a multiple of 8 bytes.

Class specific information for the Compound class (Class 6):

Bits: Compound Bit Field Description

Bits	Meaning
0-15	**Number of Members.** This field contains the number of members defined for the compound datatype. The member definitions are listed in the Properties field of the data type message.
16-23	Reserved (zero).

The Properties field of a compound datatype is a list of the member definitions of the compound datatype. The member definitions appear one after another with no intervening bytes. The member types are described with a (recursively) encoded datatype message.

Note that the property descriptions are different for different versions of the datatype version. Additionally note that the version 0 datatype encoding is deprecated and has been replaced with later encodings in versions of the HDF5 Library from the 1.4 release onward.

Layout: Compound Properties Description for Datatype Version 1

<table>
<tr><th>Byte</th><th>Byte</th><th>Byte</th><th>Byte</th></tr>
<tr><td colspan="4" align="center">
Name

</td></tr>
<tr><td colspan="4" align="center">Byte Offset of Member</td></tr>
<tr><td>Dimensionality</td><td colspan="3" align="center">Reserved (zero)</td></tr>
<tr><td colspan="4" align="center">Dimension Permutation</td></tr>
<tr><td colspan="4" align="center">Reserved (zero)</td></tr>
<tr><td colspan="4" align="center">Dimension #1 Size (required)</td></tr>
<tr><td colspan="4" align="center">Dimension #2 Size (required)</td></tr>
<tr><td colspan="4" align="center">Dimension #3 Size (required)</td></tr>
<tr><td colspan="4" align="center">Dimension #4 Size (required)</td></tr>
<tr><td colspan="4" align="center">
Member Type Message

</td></tr>
</table>

Fields: Compound Properties Description for Datatype Version 1

Field Name	Description
Name	This NUL-terminated string provides a description for the opaque type. It is NUL-padded to a multiple of 8 bytes.
Byte Offset of Member	This is the byte offset of the member within the datatype.
Dimensionality	If set to zero, this field indicates a scalar member. If set to a value greater than zero, this field indicates that the member is an array of values. For array members, the size of the array is indicated by the 'Size of Dimension n' field in this message.
Dimension Permutation	This field was intended to allow an array field to have its dimensions permuted, but this was never implemented. This field should always be set to zero.
Dimension #n Size	This field is the size of a dimension of the array field as stored in the file. The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Member Type Message	This field is a datatype message describing the datatype of the member.

Layout: Compound Properties Description for Datatype Version 2

Byte	Byte	Byte	Byte
Name			
Byte Offset of Member			
Member Type Message			

Fields: Compound Properties Description for Datatype Version 2

Field Name	Description
Name	This NUL-terminated string provides a description for the opaque type. It is NUL-padded to a multiple of 8 bytes.
Byte Offset of Member	This is the byte offset of the member within the datatype.
Member Type Message	This field is a datatype message describing the datatype of the member.

Layout: Compound Properties Description for Datatype Version 3

Byte	Byte	Byte	Byte
Name			
Byte Offset of Member *(variable size)*			
Member Type Message			

Fields: Compound Properties Description for Datatype Version 3

Field Name	Description
Name	This NUL-terminated string provides a description for the opaque type. It is *not* NUL-padded to a multiple of 8 bytes.
Byte Offset of Member	This is the byte offset of the member within the datatype. The field size is the minimum number of bytes necessary, based on the size of the datatype element. For example, a datatype element size of less than 256 bytes uses a 1 byte length, a datatype element size of 256-65535 bytes uses a 2 byte length, and so on.
Member Type Message	This field is a datatype message describing the datatype of the member.

Class specific information for the Reference class (Class 7):

Bits: Reference Bit Field Description

Bits	Meaning
0-3	**Type.** This four-bit value contains the type of reference described. The values defined are: **Value** / **Description** 0 — Object Reference: A reference to another object in this HDF5 file. 1 — Dataset Region Reference: A reference to a region within a dataset in this HDF5 file. 2-15 — Reserved
4-23	Reserved (zero).

There are no properties defined for the reference class.

Class specific information for the Enumeration class (Class 8):

Bits: Enumeration Bit Field Description

Bits	Meaning
0-15	**Number of Members.** The number of name/value pairs defined for the enumeration type.
16-23	Reserved (zero).

Layout: Enumeration Property Description for Datatype Versions 1 and 2

Byte	Byte	Byte	Byte
Base Type			
Names			
Values			

Fields: Enumeration Property Description for Datatype Versions 1 and 2

Field Name	Description
Base Type	Each enumeration type is based on some parent type, usually an integer. The information for that parent type is described recursively by this field.
Names	The name for each name/value pair. Each name is stored as a null terminated ASCII string in a multiple of eight bytes. The names are in no particular order.
Values	The list of values in the same order as the names. The values are packed (no inter-value padding) and the size of each value is determined by the parent type.

Layout: Enumeration Property Description for Datatype Version 3

Byte	Byte	Byte	Byte
Base Type			
Names			
Values			

Fields: Enumeration Property Description for Datatype Version 3

Field Name	Description
Base Type	Each enumeration type is based on some parent type, usually an integer. The information for that parent type is described recursively by this field.
Names	The name for each name/value pair. Each name is stored as a null terminated ASCII string, *not* padded to a multiple of eight bytes. The names are in no particular order.
Values	The list of values in the same order as the names. The values are packed (no inter-value padding) and the size of each value is determined by the parent type.

Class specific information for the Variable-length class (Class 9):

Bits: Variable-length Bit Field Description

<table>
<tr><th>Bits</th><th>Meaning</th></tr>
<tr><td>0-3</td><td>Type. This four-bit value contains the type of variable-length datatype described. The values defined are:
<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Sequence: A variable-length sequence of any datatype. Variable-length sequences do not have padding or character set information.</td></tr><tr><td>1</td><td>String: A variable-length sequence of characters. Variable-length strings have padding and character set information.</td></tr><tr><td>2-15</td><td>Reserved</td></tr></table></td></tr>
<tr><td>4-7</td><td>Padding type. (variable-length string only) This four-bit value determines the type of padding used for variable-length strings. The values are the same as for the string padding type, as follows:
<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Null terminate: A zero byte marks the end of a string and is guaranteed to be present after converting a long string to a short string. When converting a short string to a long string, the value is padded with additional null characters as necessary.</td></tr><tr><td>1</td><td>Null pad: Null characters are added to the end of the value during conversion from a short string to a longer string. Conversion from a long string to a shorter string simply truncates the value.</td></tr><tr><td>2</td><td>Space pad: Space characters are added to the end of the value during conversion from a short string to a longer string. Conversion from a long string to a shorter string simply truncates the value. This is the Fortran representation of the string.</td></tr><tr><td>3-15</td><td>Reserved</td></tr></table>This value is set to zero for variable-length sequences.</td></tr>
<tr><td>8-11</td><td>Character Set. (variable-length string only) This four-bit value specifies the character set to be used for encoding the string:
<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>ASCII character set encoding</td></tr><tr><td>1</td><td>UTF-8 character set encoding</td></tr><tr><td>2-15</td><td>Reserved</td></tr></table>This value is set to zero for variable-length sequences.</td></tr>
<tr><td>12-23</td><td>Reserved (zero).</td></tr>
</table>

Layout: Variable-length Property Description

Byte	Byte	Byte	Byte
Base Type			

Fields: Variable-length Property Description

Field Name	Description
Base Type	Each variable-length type is based on some parent type. The information for that parent type is described recursively by this field.

Class specific information for the Array class (Class 10):

There are no bit fields defined for the array class.

Note that the dimension information defined in the property for this datatype class is independent of dataspace information for a dataset. The dimension information here describes the dimensionality of the information within a data element (or a component of an element, if the array datatype is nested within another datatype) and the dataspace for a dataset describes the size and locations of the elements in a dataset.

Layout: Array Property Description for Datatype Version 2

Byte	Byte	Byte	Byte
Dimensionality	Reserved (zero)		
Dimension #1 Size			
. . .			
Dimension #n Size			
Permutation Index #1			
. . .			
Permutation Index #n			
Base Type			

Fields: Array Property Description for Datatype Version 2

Field Name	Description
Dimensionality	This value is the number of dimensions that the array has.
Dimension #n Size	This value is the size of the dimension of the array as stored in the file. The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Permutation Index #n	This value is the index permutation used to map each dimension from the canonical representation to an alternate axis for each dimension. Currently, dimension permutations are not supported, and these indices should be set to the index position minus one. In other words, the first dimension should be set to 0, the second dimension should be set to 1, and so on.
Base Type	Each array type is based on some parent type. The information for that parent type is described recursively by this field.

Layout: Array Property Description for Datatype Version 3

<table>
<tr><th>Byte</th><th>Byte</th><th>Byte</th><th>Byte</th></tr>
<tr><td>Dimensionality</td><td colspan="3">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Dimension #1 Size</td></tr>
<tr><td colspan="4">.
.
.</td></tr>
<tr><td colspan="4">Dimension #n Size</td></tr>
<tr><td colspan="4">Base Type</td></tr>
</table>

Fields: Array Property Description for Datatype Version 3

Field Name	Description
Dimensionality	This value is the number of dimensions that the array has.
Dimension #n Size	This value is the size of the dimension of the array as stored in the file. The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Base Type	Each array type is based on some parent type. The information for that parent type is described recursively by this field.

IV.A.2.e. The Data Storage - Fill Value (Old) Message

Header Message Name: Fill Value (old)

Header Message Type: 0x0004

Length: Varies

Status: Optional; may not be repeated.

Description: The fill value message stores a single data value which is returned to the application when an uninitialized data element is read from a dataset. The fill value is interpreted with the same datatype as the dataset. If no fill value message is present then a fill value of all zero bytes is assumed.

This fill value message is deprecated in favor of the “new” fill value message (Message Type 0x0005) and is only written to the file for forward compatibility with versions of the HDF5 Library before the 1.6.0 version. Additionally, it only appears for datasets with a user-defined fill value (as opposed to the library default fill value or an explicitly set “undefined” fill value).

Format of Data: See the tables below.

Layout: Fill Value Message (Old)

byte	byte	byte	byte
Size			
Fill Value *(optional, variable size)*			

Fields: Fill Value Message (Old)

Field Name	Description
Size	This is the size of the Fill Value field in bytes.
Fill Value	The fill value. The bytes of the fill value are interpreted using the same datatype as for the dataset.

IV.A.2.f. The Data Storage - Fill Value Message

Header Message Name: Fill Value
Header Message Type: 0x0005
Length: Varies
Status: Required for dataset objects; may not be repeated.
Description: The fill value message stores a single data value which is returned to the application when an uninitialized data element is read from a dataset. The fill value is interpreted with the same datatype as the dataset.
Format of Data: See the tables below.

Layout: Fill Value Message - Versions 1 and 2

byte	byte	byte	byte
Version	Space Allocation Time	Fill Value Write Time	Fill Value Defined
Size *(optional)*			
Fill Value *(optional, variable size)*			

Fields: Fill Value Message - Versions 1 and 2

Field Name	Description
Version	The version number information is used for changes in the format of the fill value message and is described here: **Version** **Description** 0 Never used 1 Initial version of this message. 2 In this version, the Size and Fill Value fields are only present if the Fill Value Defined field is set to 1. 3 This version packs the other fields in the message more efficiently than version 2.

Layout: Fill Value Message - Version 3

byte	byte	byte	byte
Version	Flags	*This space inserted only to align table nicely*	
Size *(optional)*			
Fill Value *(optional, variable size)*			

Fields: Fill Value Message - Version 3

Field Name	Description
Version	The version number information is used for changes in the format of the fill value message and is described here: **Version** **Description** 0 Never used 1 Initial version of this message. 2 In this version, the Size and Fill Value fields are only present if the Fill Value Defined field is set to 1. 3 This version packs the other fields in the message more efficiently than version 2.
Flags	When the storage space for the dataset’s raw data will be allocated. The allowed values are: **Bits** **Description** 0-1 Space Allocation Time, with the same values as versions 1 and 2 of the message. 2-3 Fill Value Write Time, with the same values as versions 1 and 2 of the message. 4 Fill Value Undefined, indicating that the fill value has been marked as “undefined” for this dataset. Bits 4 and 5 cannot both be set. 5 Fill Value Defined, with the same values as versions 1 and 2 of the message. Bits 4 and 5 cannot both be set. 6-7 Reserved (zero).
Size	This is the size of the Fill Value field in bytes. This field is not present if the Version field is greater than 1, and the Fill Value Defined flag is set to 0.
Fill Value	The fill value. The bytes of the fill value are interpreted using the same datatype as for the dataset. This field is not present if the Version field is greater than 1, and the Fill Value Defined flag is set to 0.

IV.A.2.g. The Link Message

Header Message Name: Link
Header Message Type: 0x0006
Length: Varies
Status: Optional; may be repeated.

Description: This message encodes the information for a link in a group's object header, when the group is storing its links "compactly", or in the group's fractal heap, when the group is storing its links "densely".

A group is storing its links compactly when the fractal heap address in the *Link Info Message* is set to the "undefined address" value.

Format of Data: See the tables below.

Layout: Link Message

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td>Version</td><td>Flags</td><td>Link type (optional)</td><td>This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Creation Order (8 bytes, optional)</td></tr>
<tr><td>Link Name Character Set (optional)</td><td>Length of Link Name (variable size)</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Link Name (variable size)</td></tr>
<tr><td colspan="4">Link Information (variable size)</td></tr>
</table>

Fields: Link Message

Field Name	Description
Version	The version number for this message. This document describes version 1.
Flags	This field contains information about the link and controls the presence of other fields below. **Bits** / **Description** 0-1: Determines the size of the *Length of Link Name* field. **Value** / **Description** 0: The size of the *Length of Link Name* field is 1 byte. 1: The size of the *Length of Link Name* field is 2 bytes. 2: The size of the *Length of Link Name* field is 4 bytes. 3: The size of the *Length of Link Name* field is 8 bytes. 2: Creation Order Field Present: if set, the *Creation Order* field is present. If not set, creation order information is not stored for links in this group. 3: Link Type Field Present: if set, the link is not a hard link and the *Link Type* field is present. If not set, the link is a hard link. 4: Link Name Character Set Field Present: if set, the link name is not represented with the ASCII character set and the *Link Name Character Set* field is present. If not set, the link name is represented with the ASCII character set. 5-7: Reserved (zero).
Link type	This is the link class type and can be one of the following values: **Value** / **Description** 0: A hard link (should never be stored in the file) 1: A soft link. 2-63: Reserved for future HDF5 internal use. 64: An external link. 65-255: Reserved, but available for user-defined link types.

	This field is present if bit 3 of *Flags* is set.
Creation Order	This 64-bit value is an index of the link's creation time within the group. Values start at 0 when the group is created an increment by one for each link added to the group. Removing a link from a group does not change existing links' creation order field. This field is present if bit 2 of *Flags* is set.
Link Name Character Set	This is the character set for encoding the link's name: **Value** **Description** 0 ASCII character set encoding (this should never be stored in the file) 1 UTF-8 character set encoding This field is present if bit 4 of *Flags* is set.
Length of link name	This is the length of the link's name. The size of this field depends on bits 0 and 1 of *Flags*.
Link name	This is the name of the link, non-NULL terminated.
Link information	The format of this field depends on the *link type*. For **hard** links, the field is formatted as follows: *Size of Offsets* bytes: The address of the object header for the object that the link points to. For **soft** links, the field is formatted as follows: Bytes 1-2: Length of soft link value. *Length of soft link value* bytes: A non-NULL-terminated string storing the value of the soft link. For **external** links, the field is formatted as follows: Bytes 1-2: Length of external link value. *Length of external link value* bytes: The first byte contains the version number in the upper 4 bits and flags in the lower 4 bits for the external link. Both version and flags are defined to be zero in this document. The remaining bytes consist of two NULL-terminated strings, with no padding between them. The first string is the name of the HDF5 file containing the object linked to and the second string is the full path to the

	object linked to, within the HDF5 file's group hierarchy. For **user-defined** links, the field is formatted as follows: Bytes 1-2: Length of user-defined data. *Length of user-defined link value* bytes: The data supplied for the user-defined link type.

IV.A.2.h. The Data Storage - External Data Files Message

Header Message Name: External Data Files

Header Message Type: 0x0007

Length: Varies

Status: Optional; may not be repeated.

Description: The external data storage message indicates that the data for an object is stored outside the HDF5 file. The filename of the object is stored as a Universal Resource Location (URL) of the actual filename containing the data. An external file list record also contains the byte offset of the start of the data within the file and the amount of space reserved in the file for that data.

Format of Data: See the tables below.

Layout: External File List Message

byte	byte	byte	byte
Version	Reserved (zero)		
Allocated Slots		Used Slots	
Heap Address^{O}			
Slot Definitions...			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: External File List Message

Field Name	Description
Version	The version number information is used for changes in the format of External Data Storage Message and is described here: **Version** **Description** 0 Never used. 1 The current version used by the library.
Allocated Slots	The total number of slots allocated in the message. Its value must be at least as large as the value contained in the Used Slots field. (The current library simply uses the number of Used Slots for this message)
Used Slots	The number of initial slots which contains valid information.
Heap Address	This is the address of a local heap which contains the names for the external files (The local heap information can be found in Disk Format Level 1D in this document). The name at offset zero in the heap is always the empty string.
Slot Definitions	The slot definitions are stored in order according to the array addresses they represent.

Layout: External File List Slot

byte	byte	byte	byte
Name Offset in Local Heap^L			
Offset in External Data File^L			
Data Size in External File^L			

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: External File List Slot

Field Name	Description
Name Offset in Local Heap	The byte offset within the local name heap for the name of the file. File names are stored as a URL which has a protocol name, a host name, a port number, and a file name: *protocol*:*port*//*host*/*file*. If the protocol is omitted then “file:” is assumed. If the port number is omitted then a default port for that protocol is used. If both the protocol and the port number are omitted then the colon can also be omitted. If the double slash and host name are omitted then “localhost” is assumed. The file name is the only mandatory part, and if the leading slash is missing then it is relative to the application’s current working directory (the use of relative names is not recommended).
Offset in External Data File	This is the byte offset to the start of the data in the specified file. For files that contain data for a single dataset this will usually be zero.
Data Size in External File	This is the total number of bytes reserved in the specified file for raw data storage. For a file that contains exactly one complete dataset which is not extendable, the size will usually be the exact size of the dataset. However, by making the size larger one allows HDF5 to extend the dataset. The size can be set to a value larger than the entire file since HDF5 will read zeroes past the end of the file without failing.

IV.A.2.i. The Data Layout Message

Header Message Name: Data Layout
Header Message Type: 0x0008
Length: Varies
Status: Required for datasets; may not be repeated.
Description: The Data Layout message describes how the elements of a multi-dimensional array are stored in the HDF5 file. Four types of data layout are supported:

1. Contiguous: The array is stored in one contiguous area of the file. This layout requires that the size of the array be constant: data manipulations such as chunking, compression, checksums, or encryption are not permitted. The message stores the total storage size of the array. The offset of an element from the beginning

of the storage area is computed as in a C array.
2. Chunked: The array domain is regularly decomposed into chunks, and each chunk is allocated and stored separately. This layout supports arbitrary element traversals, compression, encryption, and checksums (these features are described in other messages). The message stores the size of a chunk instead of the size of the entire array; the storage size of the entire array can be calculated by traversing the chunk index that stores the chunk addresses.
3. Compact: The array is stored in one contiguous block as part of this object header message.
4. Virtual: This is only supported for version 4 of the Data Layout message. The message stores information that is used to locate the global heap collection containing the Virtual Dataset (VDS) mapping information. The mapping associates the VDS to the source dataset elements that are stored across a collection of HDF5 files.

Format of Data: See the tables below.

Layout: Data Layout Message (Versions 1 and 2)

byte	byte	byte	byte
Version	Dimensionality	Layout Class	Reserved *(zero)*
Reserved *(zero)*			
Data Address[O] *(optional)*			
Dimension 0 Size			
Dimension 1 Size			
...			
Dimension #n Size			
Dataset Element Size *(optional)*			
Compact Data Size *(optional)*			
Compact Data... *(variable size, optional)*			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Data Layout Message (Versions 1 and 2)

Field Name	Description
Version	The version number information is used for changes in the format of the data layout message and is described here: **Version** **Description** 0 Never used. 1 Used by version 1.4 and before of the library to encode layout information. Data space is always allocated when the data set is created. 2 Used by version 1.6.x of the library to encode layout information. Data space is allocated only when it is necessary.
Dimensionality	An array has a fixed dimensionality. This field specifies the number of dimension size fields later in the message. The value stored for chunked storage is 1 greater than the number of dimensions in the dataset's dataspace. For example, 2 is stored for a 1 dimensional dataset.
Layout Class	The layout class specifies the type of storage for the data and how the other fields of the layout message are to be interpreted. **Value** **Description** 0 Compact Storage 1 Contiguous Storage 2 Chunked Storage
Data Address	For contiguous storage, this is the address of the raw data in the file. For chunked storage this is the address of the v1 B-tree that is used to look up the addresses of the chunks. This field is not present for compact storage. If the version for this message is greater than 1, the address may have the "undefined address" value, to indicate that storage has not yet been allocated for this array.
Dimension #n Size	For contiguous and compact storage the dimensions define the entire size of the array while for chunked storage they define the size of a single chunk. In all cases, they are in units of array elements (not bytes). The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.

Dataset Element Size	The size of a dataset element, in bytes. This field is only present for chunked storage.
Compact Data Size	This field is only present for compact data storage. It contains the size of the raw data for the dataset array, in bytes.
Compact Data	This field is only present for compact data storage. It contains the raw data for the dataset array.

Version 3 of this message re-structured the format into specific properties that are required for each layout class.

Layout: Data Layout Message (Version 3)

byte	byte	byte	byte
Version	Layout Class	*This space inserted only to align table nicely*	
Properties *(variable size)*			

Fields: Data Layout Message (Version 3)

Field Name	Description
Version	The version number information is used for changes in the format of layout message and is described here: **Version** **Description** 3 Used by the version 1.6.3 and later of the library to store properties for each layout class.
Layout Class	The layout class specifies the type of storage for the data and how the other fields of the layout message are to be interpreted. **Value** **Description** 0 Compact Storage 1 Contiguous Storage 2 Chunked Storage
Properties	This variable-sized field encodes information specific to each layout class and is described below. If there is no property information specified for a layout class, the size of this field is zero bytes.

Class-specific information for compact storage (layout class 0): (Note: The dimensionality information is in the Dataspace message)

Layout: Compact Storage Property Description

byte	byte	byte	byte
Size		*This space inserted only to align table nicely*	
Raw Data... *(variable size)*			

Fields: Compact Storage Property Description

Field Name	Description
Size	This field contains the size of the raw data for the dataset array, in bytes.
Raw Data	This field contains the raw data for the dataset array.

Class-specific information for contiguous storage (layout class 1): (Note: The dimensionality information is in the Dataspace message)

Layout: Contiguous Storage Property Description

byte	byte	byte	byte
Address^{O}			
Size^{L}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Contiguous Storage Property Description

Field Name	Description
Address	This is the address of the raw data in the file. The address may have the “undefined address” value, to indicate that storage has not yet been allocated for this array.
Size	This field contains the size allocated to store the raw data, in bytes.

Class-specific information for chunked storage (layout class 2):

Layout: Chunked Storage Property Description

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td>Dimensionality</td><td colspan="3">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Address^O</td></tr>
<tr><td colspan="4">Dimension 0 Size</td></tr>
<tr><td colspan="4">Dimension 1 Size</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="4">Dimension #n Size</td></tr>
<tr><td colspan="4">Dataset Element Size</td></tr>
</table>

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Chunked Storage Property Description

Field Name	Description
Dimensionality	A chunk has a fixed dimensionality. This field specifies the number of dimension size fields later in the message.
Address	This is the address of the v1 B-tree that is used to look up the addresses of the chunks that actually store portions of the array data. The address may have the “undefined address” value, to indicate that storage has not yet been allocated for this array.
Dimension #n Size	These values define the dimension size of a single chunk, in units of array elements (not bytes). The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Dataset Element Size	The size of a dataset element, in bytes.

Version 4 of this message is similar to version 3 but has additional information for the virtual layout class as well as indexing information for the chunked layout class.

Layout: Data Layout Message (Version 4)

byte	byte	byte	byte
Version	Layout Class	*This space inserted only to align table nicely*	
Properties *(variable size)*			

Fields: Data Layout Message (Version 4)

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>Version</td><td>The value for this field is 4 and is used by version 1.10.0 and later of the library to store properties for each layout class and indexing information for the chunked layout.</td></tr>
<tr><td>Layout Class</td><td>The layout class specifies the type of storage for the data and how the other fields of the layout message are to be interpreted.

<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Compact Storage</td></tr><tr><td>1</td><td>Contiguous Storage</td></tr><tr><td>2</td><td>Chunked Storage</td></tr><tr><td>3</td><td>Virtual Storage</td></tr></table></td></tr>
<tr><td>Properties</td><td>This variable-sized field encodes information specific to a layout class as follows:

<table><tr><th>Layout Class</th><th>Description</th></tr><tr><td>Compact Storage</td><td>See Compact Storage Property Description for the version 3 Data Layout message.</td></tr><tr><td>Contiguous Storage</td><td>See Contiguous Storage Property Description for the version 3 Data Layout message.</td></tr><tr><td>Chunked Storage</td><td>See Chunked Storage Property Description below.</td></tr><tr><td>Virtual Storage</td><td>See Virtual Storage Property Description below.</td></tr></table></td></tr>
</table>

Class-specific information for chunked storage (layout class 2):

Layout: Chunked Storage Property Description

byte	byte	byte	byte
Flags	Dimensionality	Dimension Size Encoded Length	*This space inserted to align table nicely*
Dimension 0 Size *(variable size)*			
Dimension 1 Size *(variable size)*			
...			
Dimension #n Size *(variable size)*			
Chunk Indexing Type	*This space inserted only to align table nicely*		
Indexing Type Information *(variable size)*			
Address[O]			

Fields: Chunked Storage Property Description

Field Name	Description
Flags	This is the chunked layout feature flag: **Value** — **Description** DONT_FILTER_PARTIAL_BOUND_CHUNKS (bit 0) — Do not apply filter to a partial edge chunk. SINGLE_INDEX_WITH_FILTER (bit 1) — A filtered chunk for *Single Chunk* indexing.
Dimensionality	A chunk has fixed dimension. This field specifies the number of *Dimension Size* fields later in the message.
Dimension Size Encoded Length	This is the size in bytes used to encode *Dimension Size*.
Dimension #n Size	These values define the dimension size of a single chunk, in units of array elements (not bytes). The first dimension stored in the list of dimensions is the slowest changing dimension and the last dimension stored is the fastest changing dimension.
Chunk Indexing Type	There are five indexing types used to look up addresses of the chunks. For more information on each type, see “Appendix C: Types of Indexes for Dataset Chunks.” **Value** — **Description** 1 — *Single Chunk* indexing type. 2 — *Implicit* indexing type. 3 — *Fixed Array* indexing type. 4 — *Extensible Array* indexing type. 5 — *Version 2 B-tree* indexing type.
Indexing Type Information	This variable-sized field encodes information specific to an indexing type. More information on what is encoded with each type can be found below this table. • See *Single Chunk* below. • See *Implicit* below. • See *Fixed Array* below. • See *Extensible Array* below. • See *Version 2 B-tree* below.

Address	This is the address specific to an indexing type. The address may be undefined if the chunk or index storage is not allocated yet. **Value** / **Description** *Single Chunk index*: Address of the single chunk. *Implicit index*: Address of the array of dataset chunks. *Fixed Array index*: Address of the index. *Extensible Array index*: Address of the index. *Version 2 B-tree index*: Address of the index.

1. Index-specific information for *Single Chunk*:

   The following information exists only when the chunk is filtered. In other words, when DONT_FILTER_PARTIAL_BOUND_CHUNKS (bit 0) is enabled in the field *flags*.

Layout: Single Chunk Indexing Information

byte	byte	byte	byte
Size of filtered chunk[L]			
Filters for chunk			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Single Chunk Indexing Information

Field Name	Description
Size of filtered chunk	This field is the size of a filtered chunk.
Filters for chunk	This field contains filters for the chunk.

2. Index-specific information for *Implicit*:

Layout: Implicit Indexing Information

byte	byte	byte	byte
No specific indexing information			

3. Index-specific information for *Fixed Array*:

Layout: Fixed Array Indexing Information

byte	byte	byte	byte
Page Bits	*This space inserted only to align table nicely*		

Fields: Fixed Array Indexing Information

Field Name	Description
Page Bits	This field contains the number of bits needed to store the maximum number of elements in a data block page.

4. Index-specific information for *Extensible Array*:

Layout: Extensible Array Indexing Information

byte	byte	byte	byte
Max Bits	Index Elements	Min Pointers	Min Elements
Page Bits		*This space inserted only to align table nicely*	

Fields: Extensible Array Indexing Information

Field Name	Description
Max Bits	This field contains the number of bits needed to store the maximum number of elements in the array.
Index Elements	This field contains the number of elements to store in the index block.
Min Pointers	This field contains the minimum number of data block pointers for a superblock.
Min Elements	This field contains the minimum number of elements per data block.
Page Bits	This field contains the number of bits needed to store the maximum number of elements in a data block page.

5. Index-specific information for *Version 2 B-tree*:

Layout: Version 2 B-tree Indexing Information

byte	byte	byte	byte
Node Size			
Split Percent	Merge Percent	*This space inserted only to align table nicely*	

Fields: Version 2 B-tree Indexing Information

Field Name	Description
Node Size	This field is the size in bytes of a B-tree node.
Split Percent	This field is the percentage full of a B-tree node at which to split the node.
Merge Percent	This field is the percentage full of a B-tree node at which to merge the node.

Class-specific information for virtual storage (layout class 3):

Layout: Virtual Storage Property Description

byte	byte	byte	byte
Address[O]			
Index			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Virtual Storage Property Description

Field Name	Description
Address	This is the address of the global heap collection where the VDS mapping entries are stored. See “Disk Format: Level 1F - Global Heap Block for Virtual Datasets.”
Index	This is the index of the data object within the global heap collection.

IV.A.2.j. The Bogus Message

Header Message Name: Bogus

Header Message Type: 0x0009

Length: 4 bytes

Status: For testing only; should never be stored in a valid file.

Description: This message is used for testing the HDF5 Library's response to an "unknown" message type and should never be encountered in a valid HDF5 file.

Format of Data: See the tables below.

Layout: Bogus Message

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4">Bogus Value</td></tr>
</table>

Fields: Bogus Message

Field Name	Description
Bogus Value	This value should always be: 0xdeadbeef.

IV.A.2.k. The Group Info Message

Header Message Name: Group Info

Header Message Type: 0x000A

Length: Varies

Status: Optional; may not be repeated.

Description: This message stores information for the constants defining a "new style" group's behavior. Constant information will be stored in this message and variable information will be stored in the Link Info message.

Note: the "estimated entry" information below is used when determining the size of the object header for the group when it is created.

Format of Data: See the tables below.

Layout: Group Info Message

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td>Version</td><td>Flags</td><td colspan="2">Link Phase Change: Maximum Compact Value (optional)</td></tr>
<tr><td colspan="2">Link Phase Change: Minimum Dense Value (optional)</td><td colspan="2">Estimated Number of Entries (optional)</td></tr>
<tr><td colspan="2">Estimated Link Name Length of Entries (optional)</td><td colspan="2">This space inserted only to align table nicely</td></tr>
</table>

Fields: Group Info Message

Field Name	Description
Version	The version number for this message. This document describes version 0.
Flags	This is the group information flag with the following definition: **Bit** — **Description** 0 — If set, link phase change values are stored. 1 — If set, the estimated entry information is non-default and is stored. 2-7 — Reserved
Link Phase Change: Maximum Compact Value	The is the maximum number of links to store “compactly” (in the group’s object header). This field is present if bit 0 of *Flags* is set.
Link Phase Change: Minimum Dense Value	This is the minimum number of links to store “densely” (in the group’s fractal heap). The fractal heap’s address is located in the Link Info message. This field is present if bit 0 of *Flags* is set.
Estimated Number of Entries	This is the estimated number of entries in groups. If this field is not present, the default value of 4 will be used for the estimated number of group entries. This field is present if bit 1 of *Flags* is set.
Estimated Link Name Length of Entries	This is the estimated length of entry name. If this field is not present, the default value of 8 will be used for the estimated link name length of group entries. This field is present if bit 1 of *Flags* is set.

IV.A.2.I. The Data Storage - Filter Pipeline Message

Header Message Name: Data Storage - Filter Pipeline
Header Message Type: 0x000B
Length: Varies
Status: Optional; may not be repeated.

Description: This message describes the filter pipeline which should be applied to the data stream by providing filter identification numbers, flags, a name, and client data.

This message may be present in the object headers of both dataset and group objects. For datasets, it specifies the filters to apply to raw data. For groups, it specifies the filters to apply to the group’s fractal heap. Currently, only datasets using chunked data storage use the filter pipeline on their raw data.

Format of Data: See the tables below.

Layout: Filter Pipeline Message - Version 1

byte	byte	byte	byte
Version	Number of Filters	Reserved (zero)	
Reserved (zero)			
Filter Description List *(variable size)*			

Fields: Filter Pipeline Message - Version 1

Field Name	Description
Version	The version number for this message. This table describes version 1.
Number of Filters	The total number of filters described in this message. The maximum possible number of filters in a message is 32.
Filter Description List	A description of each filter. A filter description appears in the next table.

Layout: Filter Description - Version 1

byte	byte	byte	byte
Filter Identification Value		Name Length	
Flags		Number Client Data Values	
Name *(variable size, optional)*			
Client Data *(variable size, optional)*			
Padding *(variable size, optional)*			

Fields: Filter Description - Version 1

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>Filter Identification Value</td><td>

This value, often referred to as a filter identifier, is designed to be a unique identifier for the filter. Values from zero through 32,767 are reserved for filters supported by The HDF Group in the HDF5 Library and for filters requested and supported by third parties. Filters supported by The HDF Group are documented immediately below. Information on 3rd-party filters can be found at The HDF Group's Contributions page.

To request a filter identifier, please contact The HDF Group's Help Desk at . You will be asked to provide the following information:

1. Contact information for the developer requesting the new identifier
2. A short description of the new filter
3. Links to any relevant information, including licensing information

Values from 32768 to 65535 are reserved for non-distributed uses (for example, internal company usage) or for application usage when testing a feature. The HDF Group does not track or document the use of the filters with identifiers from this range.

The filters currently in library version 1.8.0 are listed below:

<table>
<tr><th>Identification</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>N/A</td><td>Reserved</td></tr>
<tr><td>1</td><td>deflate</td><td>GZIP deflate compression</td></tr>
<tr><td>2</td><td>shuffle</td><td>Data element shuffling</td></tr>
<tr><td>3</td><td>fletcher32</td><td>Fletcher32 checksum</td></tr>
<tr><td>4</td><td>szip</td><td>SZIP compression</td></tr>
<tr><td>5</td><td>nbit</td><td>N-bit packing</td></tr>
<tr><td>6</td><td>scaleoffset</td><td>Scale and offset encoded values</td></tr>
</table>

</td></tr>
<tr><td>Name Length</td><td>Each filter has an optional null-terminated ASCII name and this field holds the length of the name including the null termination padded with nulls to be a multiple of eight. If the filter has no name then a value of zero is stored in this field.</td></tr>
</table>

Flags	The flags indicate certain properties for a filter. The bit values defined so far are: **Bit** **Description** 0 If set then the filter is an optional filter. During output, if an optional filter fails it will be silently skipped in the pipeline. 1-15 Reserved (zero)
Number of Client Data Values	Each filter can store integer values to control how the filter operates. The number of entries in the *Client Data* array is stored in this field.
Name	If the *Name Length* field is non-zero then it will contain the size of this field, padded to a multiple of eight. This field contains a null-terminated, ASCII character string to serve as a comment/name for the filter.
Client Data	This is an array of four-byte integers which will be passed to the filter function. The *Client Data Number* of Values determines the number of elements in the array.
Padding	Four bytes of zeroes are added to the message at this point if the Client Data Number of Values field contains an odd number.

Layout: Filter Pipeline Message - Version 2

byte	byte	byte	byte
Version	Number of Filters	*This space inserted only to align table nicely*	
Filter Description List *(variable size)*			

Fields: Filter Pipeline Message - Version 2

Field Name	Description
Version	The version number for this message. This table describes version 2.
Number of Filters	The total number of filters described in this message. The maximum possible number of filters in a message is 32.
Filter Description List	A description of each filter. A filter description appears in the next table.

Layout: Filter Description - Version 2

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="2">Filter Identification Value</td><td colspan="2">Name Length (optional)</td></tr>
<tr><td colspan="2">Flags</td><td colspan="2">Number Client Data Values</td></tr>
<tr><td colspan="4">Name (variable size, optional)</td></tr>
<tr><td colspan="4">Client Data (variable size, optional)</td></tr>
</table>

Fields: Filter Description - Version 2

Field Name	Description
Filter Identification Value	This value, often referred to as a filter identifier, is designed to be a unique identifier for the filter. Values from zero through 32,767 are reserved for filters supported by The HDF Group in the HDF5 Library and for filters requested and supported by third parties. Filters supported by The HDF Group are documented immediately below. Information on 3rd-party filters can be found at The HDF Group's Contributions page. To request a filter identifier, please contact The HDF Group's Help Desk at . You will be asked to provide the following information: 1. Contact information for the developer requesting the new identifier 2. A short description of the new filter 3. Links to any relevant information, including licensing information Values from 32768 to 65535 are reserved for non-distributed uses (for example, internal company usage) or for application usage when testing a feature. The HDF Group does not track or document the use of the filters with identifiers from this range. The filters currently in library version 1.8.0 are listed below: **Identification** / **Name** / **Description** 0 / N/A / Reserved 1 / deflate / GZIP deflate compression 2 / shuffle / Data element shuffling 3 / fletcher32 / Fletcher32 checksum 4 / szip / SZIP compression 5 / nbit / N-bit packing 6 / scaleoffset / Scale and offset encoded values
Name Length	Each filter has an optional null-terminated ASCII name and this field holds the length of the name including the null termination padded with nulls to be a multiple of eight. If the filter has no name then a value of zero is stored in this field. Filters with IDs less than 256 (in other words, filters

	that are defined in this format documentation) do not store the *Name Length* or *Name* fields.
Flags	The flags indicate certain properties for a filter. The bit values defined so far are: **Bit** — **Description** 0 — If set then the filter is an optional filter. During output, if an optional filter fails it will be silently skipped in the pipeline. 1-15 — Reserved (zero)
Number of Client Data Values	Each filter can store integer values to control how the filter operates. The number of entries in the *Client Data* array is stored in this field.
Name	If the *Name Length* field is non-zero, then it will contain the size of this field, *not* padded to a multiple of eight. This field contains a *non*-null-terminated, ASCII character string to serve as a comment/name for the filter. Filters that are defined in this format documentation such as deflate and shuffle do not store the *Name Length* or *Name* fields.
Client Data	This is an array of four-byte integers which will be passed to the filter function. The Client Data Number of Values determines the number of elements in the array.

IV.A.2.m. The Attribute Message

Header Message Name: Attribute
Header Message Type: 0x000C
Length: Varies
Status: Optional; may be repeated.

Description: The *Attribute* message is used to store objects in the HDF5 file which are used as attributes, or “metadata” about the current object. An attribute is a small dataset; it has a name, a datatype, a dataspace, and raw data. Since attributes are stored in the object header, they should be relatively small (in other words, less than 64KB). They can be associated with any type of object which has an object header (groups, datasets, or committed (named) datatypes).

In 1.8.x versions of the library, attributes can be larger than 64KB. See the “Special Issues” section of the Attributes chapter in the *HDF5 User’s Guide* for more information.

Note: Attributes on an object must have unique names: the HDF5 Library currently enforces this by causing the creation of an attribute with a duplicate name to fail. Attributes on different objects may have the same name, however.

Format of Data: See the tables below.

Layout: Attribute Message (Version 1)

byte	byte	byte	byte
Version	Reserved (zero)	Name Size	
Datatype Size		Dataspace Size	
Name *(variable size)*			
Datatype *(variable size)*			
Dataspace *(variable size)*			
Data *(variable size)*			

Fields: Attribute Message (Version 1)

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>Version</td><td>The version number information is used for changes in the format of the attribute message and is described here:
<table><tr><th><u>Version</u></th><th><u>Description</u></th></tr><tr><td>0</td><td>Never used.</td></tr><tr><td>1</td><td>Used by the library before version 1.6 to encode attribute message. This version does not support shared datatypes.</td></tr></table></td></tr>
<tr><td>Name Size</td><td>The length of the attribute name in bytes including the null terminator. Note that the Name field below may contain additional padding not represented by this field.</td></tr>
<tr><td>Datatype Size</td><td>The length of the datatype description in the Datatype field below. Note that the Datatype field may contain additional padding not represented by this field.</td></tr>
<tr><td>Dataspace Size</td><td>The length of the dataspace description in the Dataspace field below. Note that the Dataspace field may contain additional padding not represented by this field.</td></tr>
<tr><td>Name</td><td>The null-terminated attribute name. This field is padded with additional null characters to make it a multiple of eight bytes.</td></tr>
<tr><td>Datatype</td><td>The datatype description follows the same format as described for the datatype object header message. This field is padded with additional zero bytes to make it a multiple of eight bytes.</td></tr>
<tr><td>Dataspace</td><td>The dataspace description follows the same format as described for the dataspace object header message. This field is padded with additional zero bytes to make it a multiple of eight bytes.</td></tr>
<tr><td>Data</td><td>The raw data for the attribute. The size is determined from the datatype and dataspace descriptions. This field is not padded with additional bytes.</td></tr>
</table>

Layout: Attribute Message (Version 2)

byte	byte	byte	byte
Version	Flags	Name Size	
Datatype Size		Dataspace Size	
Name *(variable size)*			
Datatype *(variable size)*			
Dataspace *(variable size)*			
Data *(variable size)*			

Fields: Attribute Message (Version 2)

Field Name	Description
Version	The version number information is used for changes in the format of the attribute message and is described here: **Version** / **Description** 2 — Used by the library of version 1.6.x and after to encode attribute messages. This version supports shared datatypes. The fields of name, datatype, and dataspace are not padded with additional bytes of zero.
Flags	This bit field contains extra information about interpreting the attribute message: **Bit** / **Description** 0 — If set, datatype is shared. 1 — If set, dataspace is shared.
Name Size	The length of the attribute name in bytes including the null terminator.
Datatype Size	The length of the datatype description in the *Datatype* field below.
Dataspace Size	The length of the dataspace description in the *Dataspace* field below.
Name	The null-terminated attribute name. This field is *not* padded with additional bytes.
Datatype	The datatype description follows the same format as described for the datatype object header message. If the *Flag* field indicates this attribute's datatype is shared, this field will contain a "shared message" encoding instead of the datatype encoding. This field is *not* padded with additional bytes.
Dataspace	The dataspace description follows the same format as described for the dataspace object header message. If the *Flag* field indicates this attribute's dataspace is shared, this field will contain a "shared message" encoding instead of the dataspace encoding. This field is *not* padded with additional bytes.

Data	The raw data for the attribute. The size is determined from the datatype and dataspace descriptions. This field is *not* padded with additional zero bytes.

Layout: Attribute Message (Version 3)

byte	byte	byte	byte
Version	Flags	Name Size	
Datatype Size		Dataspace Size	
Name Character Set Encoding	*This space inserted only to align table nicely*		
Name *(variable size)*			
Datatype *(variable size)*			
Dataspace *(variable size)*			
Data *(variable size)*			

Fields: Attribute Message (Version 3)

Field Name	Description
Version	The version number information is used for changes in the format of the attribute message and is described here: **Version** / **Description** 3 — Used by the library of version 1.8.x and after to encode attribute messages. This version supports attributes with non-ASCII names.
Flags	This bit field contains extra information about interpreting the attribute message: **Bit** / **Description** 0 — If set, datatype is shared. 1 — If set, dataspace is shared.
Name Size	The length of the attribute name in bytes including the null terminator.
Datatype Size	The length of the datatype description in the *Datatype* field below.
Dataspace Size	The length of the dataspace description in the *Dataspace* field below.
Name Character Set Encoding	The character set encoding for the attribute's name: **Value** / **Description** 0 — ASCII character set encoding 1 — UTF-8 character set encoding
Name	The null-terminated attribute name. This field is *not* padded with additional bytes.
Datatype	The datatype description follows the same format as described for the datatype object header message. If the *Flag* field indicates this attribute's datatype is shared, this field will contain a "shared message" encoding instead of the datatype encoding. This field is *not* padded with additional bytes.
Dataspace	The dataspace description follows the same format as described for the dataspace object header message.

	If the *Flag* field indicates this attribute's dataspace is shared, this field will contain a "shared message" encoding instead of the dataspace encoding. This field is *not* padded with additional bytes.
Data	The raw data for the attribute. The size is determined from the datatype and dataspace descriptions. This field is *not* padded with additional zero bytes.

IV.A.2.n. The Object Comment Message

Header Message Name: Object Comment

Header Message Type: 0x000D

Length: Varies

Status: Optional; may not be repeated.

Description: The object comment is designed to be a short description of an object. An object comment is a sequence of non-zero (\0) ASCII characters with no other formatting included by the library.

Format of Data: See the tables below.

Layout: Object Comment Message

byte	byte	byte	byte
Comment *(variable size)*			

Fields: Object Comment Message

Field Name	Description
Name	A null terminated ASCII character string.

IV.A.2.o. The Object Modification Time (Old) Message

Header Message Name: Object Modification Time (Old)

Header Message Type: 0x000E

Length: Fixed

Status: Optional; may not be repeated.

Description: The object modification date and time is a timestamp which indicates (using ISO-8601 date and time format) the last modification of an object. The time is updated when any object header message changes according to the system clock where the change was posted. All fields of this message should be interpreted as coordinated universal time

(UTC).

This modification time message is deprecated in favor of the "new" Object Modification Time message and is no longer written to the file in versions of the HDF5 Library after the 1.6.0 version.

Format of Data: See the tables below.

Layout: Modification Time Message (Old)

byte	byte	byte	byte
Year			
Month		Day of Month	
Hour		Minute	
Second		Reserved	

Fields: Modification Time Message (Old)

Field Name	Description
Year	The four-digit year as an ASCII string. For example, 1998.
Month	The month number as a two digit ASCII string where January is 01 and December is 12.
Day of Month	The day number within the month as a two digit ASCII string. The first day of the month is 01.
Hour	The hour of the day as a two digit ASCII string where midnight is 00 and 11:00pm is 23.
Minute	The minute of the hour as a two digit ASCII string where the first minute of the hour is 00 and the last is 59.
Second	The second of the minute as a two digit ASCII string where the first second of the minute is 00 and the last is 59.
Reserved	This field is reserved and should always be zero.

IV.A.2.p. The Shared Message Table Message

Header Message Name: Shared Message Table
Header Message Type: 0x000F
Length: Fixed
Status: Optional; may not be repeated.

Description: This message is used to locate the table of shared object header message (SOHM) indexes. Each index consists of information to find the shared messages from either the heap or object header. This message is *only* found in the superblock extension.

Format of Data: See the tables below.

Layout: Shared Message Table Message

byte	byte	byte	byte
Version	*This space inserted only to align table nicely*		
Shared Object Header Message Table Address[O]			
Number of Indices	*This space inserted only to align table nicely*		

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Shared Message Table Message

Field Name	Description
Version	The version number for this message. This document describes version 0.
Shared Object Header Message Table Address	This field is the address of the master table for shared object header message indexes.
Number of Indices	This field is the number of indices in the master table.

IV.A.2.q. The Object Header Continuation Message

Header Message Name: Object Header Continuation

Header Message Type: 0x0010

Length: Fixed

Status: Optional; may be repeated.

Description: The object header continuation is the location in the file of a block containing more header messages for the current data object. This can be used when header blocks become too large or are likely to change over time.

Format of Data: See the tables below.

Layout: Object Header Continuation Message

byte	byte	byte	byte
Offset^{O}			
Length^{L}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: Object Header Continuation Message

Field Name	Description
Offset	This value is the address in the file where the header continuation block is located.
Length	This value is the length in bytes of the header continuation block in the file.

The format of the header continuation block that this message points to depends on the version of the object header that the message is contained within.

Continuation blocks for version 1 object headers have no special formatting information; they are merely a list of object header message info sequences (type, size, flags, reserved bytes and data for each message sequence). See the description of Version 1 Data Object Header Prefix.

Continuation blocks for version 2 object headers *do* have special formatting information as described here (see also the description of Version 2 Data Object Header Prefix.):

Layout: Version 2 Object Header Continuation Block

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td colspan="4">Signature</td></tr>
<tr><td>Header Message Type #1</td><td colspan="2">Size of Header Message Data #1</td><td>Header Message #1 Flags</td></tr>
<tr><td colspan="2">Header Message #1 Creation Order (optional)</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Header Message Data #1</td></tr>
<tr><td colspan="4">.
.
.</td></tr>
<tr><td>Header Message Type #n</td><td colspan="2">Size of Header Message Data #n</td><td>Header Message #n Flags</td></tr>
<tr><td colspan="2">Header Message #n Creation Order (optional)</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">Header Message Data #n</td></tr>
<tr><td colspan="4">Gap (optional, variable size)</td></tr>
<tr><td colspan="4">Checksum</td></tr>
</table>

Fields: Version 2 Object Header Continuation Block

Field Name	Description
Signature	The ASCII character string “OCHK” is used to indicate the beginning of an object header continuation block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Header Message #n Type	Same format as version 1 of the object header, described above.
Size of Header Message #n Data	Same format as version 1 of the object header, described above.
Header Message #n Flags	Same format as version 1 of the object header, described above.
Header Message #n Creation Order	This field stores the order that a message of a given type was created in. This field is present if bit 2 of *flags* is set.

IV.A.2.r. The Symbol Table Message

Header Message Name: Symbol Table Message

Header Message Type: 0x0011

Length: Fixed

Status: Required for "old style" groups; may not be repeated.

Description: Each "old style" group has a v1 B-tree and a local heap for storing symbol table entries, which are located with this message.

Format of data: See the tables below.

Layout: Symbol Table Message

byte	byte	byte	byte
v1 B-tree Address[O]			
Local Heap Address[O]			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Symbol Table Message

Field Name	Description
v1 B-tree Address	This value is the address of the v1 B-tree containing the symbol table entries for the group.
Local Heap Address	This value is the address of the local heap containing the link names for the symbol table entries for the group.

IV.A.2.s. The Object Modification Time Message

Header Message Name: Object Modification Time

Header Message Type: 0x0012

Length: Fixed

Status: Optional; may not be repeated.

Description: The object modification time is a timestamp which indicates the time of the last modification of an object. The time is updated when any object header message changes according to the system clock where the change was posted.

Format of Data: See the tables below.

Layout: Modification Time Message

byte	byte	byte	byte
Version	Reserved *(zero)*		
Seconds After UNIX Epoch			

Fields: Modification Time Message

Field Name	Description
Version	The version number is used for changes in the format of Object Modification Time and is described here: **Version** **Description** 0 Never used. 1 Used by Version 1.6.1 and after of the library to encode time. In this version, the time is the seconds after Epoch.
Seconds After UNIX Epoch	A 32-bit unsigned integer value that stores the number of seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

IV.A.2.t. The B-tree ‘K’ Values Message

Header Message Name: B-tree ‘K’ Values

Header Message Type: 0x0013

Length: Fixed

Status: Optional; may not be repeated.

Description: This message retrieves non-default ‘K’ values for internal and leaf nodes of a group or indexed storage v1 B-trees. This message is *only* found in the superblock extension.

Format of Data: See the tables below.

Layout: B-tree ‘K’ Values Message

byte	byte	byte	byte
Version	Indexed Storage Internal Node K		*This space inserted only to align table nicely*
Group Internal Node K		Group Leaf Node K	

Fields: B-tree ‘K’ Values Message

Field Name	Description
Version	The version number for this message. This document describes version 0.
Indexed Storage Internal Node K	This is the node ‘K’ value for each internal node of an indexed storage v1 B-tree. See the description of this field in version 0 and 1 of the superblock as well the section on v1 B-trees.
Group Internal Node K	This is the node ‘K’ value for each internal node of a group v1 B-tree. See the description of this field in version 0 and 1 of the superblock as well as the section on v1 B-trees.
Group Leaf Node K	This is the node ‘K’ value for each leaf node of a group v1 B-tree. See the description of this field in version 0 and 1 of the superblock as well as the section on v1 B-trees.

IV.A.2.u. The Driver Info Message

Header Message Name: Driver Info

Header Message Type: 0x0014

Length: Varies

Status: Optional; may not be repeated.

Description: This message contains information needed by the file driver to reopen a file. This message is *only* found in the superblock extension: see the “Disk Format: Level 0C - Superblock Extension” section for more information. For more information on the fields in the driver info message, see the “Disk Format: Level 0B - File Driver Info” section; those who use the multi and family file drivers will find this section particularly helpful.

Format of Data: See the tables below.

Layout: Driver Info Message

<table>
<tr><th>byte</th><th>byte</th><th>byte</th><th>byte</th></tr>
<tr><td>Version</td><td colspan="3">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">
Driver Identification</td></tr>
<tr><td colspan="2">Driver Information Size</td><td colspan="2">This space inserted only to align table nicely</td></tr>
<tr><td colspan="4">
Driver Information (variable size)

</td></tr>
</table>

Fields: Driver Info Message

Field Name	Description
Version	The version number for this message. This document describes version 0.
Driver Identification	This is an eight-byte ASCII string without null termination which identifies the driver.
Driver Information Size	The size in bytes of the *Driver Information* field of this message.
Driver Information	Driver information is stored in a format defined by the file driver.

IV.A.2.v. The Attribute Info Message

Header Message Name: Attribute Info

Header Message Type: 0x0015

Length: Varies

Status: Optional; may not be repeated.

Description: This message stores information about the attributes on an object, such as the maximum creation index for the attributes created and the location of the attribute storage when the attributes are stored “densely”.

Format of Data: See the tables below.

Layout: Attribute Info Message

byte	byte	byte	byte
Version	Flags	Maximum Creation Index *(optional)*	
Fractal Heap Address^O			
Attribute Name v2 B-tree Address^O			
Attribute Creation Order v2 B-tree Address^O *(optional)*			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Attribute Info Message

Field Name	Description
Version	The version number for this message. This document describes version 0.
Flags	This is the attribute index information flag with the following definition: **Bit** **Description** 0 If set, creation order for attributes is tracked. 1 If set, creation order for attributes is indexed. 2-7 Reserved
Maximum Creation Index	The is the maximum creation order index value for the attributes on the object. This field is present if bit 0 of *Flags* is set.
Fractal Heap Address	This is the address of the fractal heap to store dense attributes.
Attribute Name v2 B-tree Address	This is the address of the version 2 B-tree to index the names of densely stored attributes.
Attribute Creation Order v2 B-tree Address	This is the address of the version 2 B-tree to index the creation order of densely stored attributes. This field is present if bit 1 of *Flags* is set.

IV.A.2.w. The Object Reference Count Message

Header Message Name: Object Reference Count

Header Message Type: 0x0016

Length: Fixed

Status: Optional; may not be repeated.

Description: This message stores the number of hard links (in groups or objects) pointing to an object: in other words, its *reference count*.

Format of Data: See the tables below.

Layout: Object Reference Count

byte	byte	byte	byte
Version	*This space inserted only to align table nicely*		
Reference count			

Fields: Object Reference Count

Field Name	Description
Version	The version number for this message. This document describes version 0.
Reference Count	The unsigned 32-bit integer is the reference count for the object. This message is only present in "version 2" (or later) object headers, and if not present those object header versions, the reference count for the object is assumed to be 1.

IV.A.2.x. The File Space Info Message

Header Message Name: File Space Info
Header Message Type: 0x0017
Length: Fixed
Status: Optional; may not be repeated.
Description: This message stores the file space management information that the library uses in handling file space requests for the file. Version 0 of the message is used for release 1.10.0 only. Version 1 of the message is used for release 1.10.1+. There is no File Space Info message before release 1.10 as the library does not track file space across multiple file opens.

Note that version 0 is deprecated starting release 1.10.1. That means when the 1.10.1+ library opens an HDF5 file with a version 0 message, the library will decode and map the message to version 1. On file close, it will encode the message as a version 1 message.

The library uses the following three mechanisms to manage file space in an HDF5 file:

- Free-space managers
  They track free-space sections of various sizes in the file that are not currently allocated. Each free-space manager corresponds to a file space type. There are two main groups of file space types: metadata and raw data. Metadata is further divided into five types: superblock, B-tree, global heap, local heap, and object header. See the description of Free-space Manager as well the description of file space allocation types in Appendix B
- Aggregators
  The library manages two aggregators, one for metadata and one for raw data. Aggregator is a contiguous block of free-space in the file. The size of each aggregator is tunable via public routines H5Pset_meta_block_size and H5Pset_small_data_block_size respectively.

- Virtual file drivers
  The library's virtual file driver interface dispatches requests for additional space to the allocation routine of the file driver associated with the file. For example, if the sec2 file driver is being used, its allocation routine will increase the size of the file to service the requests.

For release 1.10.0, the library derives the following four file space strategies based on the mechanisms:

- H5F_FILE_SPACE_ALL
  - Mechanisms used: free-space managers, aggregators, and virtual file drivers
  - Does not persist free-space across file opens
  - This strategy is the library default
- H5F_FILE_SPACE_ALL_PERSIST
  - Mechanisms used: free-space managers, aggregators, and virtual file drivers
  - Persist free-space across file opens
- H5F_FILE_SPACE_AGGR_VFD
  - Mechanisms used: aggregators and virtual file drivers
  - Does not persist free-space across file opens
- H5F_FILE_SPACE_VFD
  - Mechanisms used: virtual file drivers
  - Does not persist free-space across file opens

For release 1.10.1+, the free-space manager mechanism is modified to handle paged aggregation which aggregates small metadata and raw data allocations into constant-sized well-aligned pages to allow efficient I/O accesses. With the support of this feature, the library derives the following four file space strategies:

- H5F_FSPACE_STRATEGY_FSM_AGGR
  - Mechanisms used: free-space managers, aggregators, and virtual file drivers
  - This strategy is the library default
- H5F_FSPACE_STRATEGY_PAGE
  - Mechanisms used: free-space managers with embedded paged aggregation and virtual file drivers
- H5F_FSPACE_STRATEGY_AGGR
  - Mechanisms used: aggregators and virtual file drivers
- H5F_FSPACE_STRATEGY_NONE
  - Mechanisms used: virtual file drivers

The default is not persisting free-space across file opens for the above four strategies. User can use the public routine H5Pset_file_space_strategy to request persisting free-space.

Format of Data: See the tables below.

Layout: File Space Info - Version 0

byte	byte	byte	byte
Version	Strategy	Threshold^{L}	
Free-space manager address^{O} for H5FD_MEM_SUPER			
Free-space manager address^{0} for H5FD_MEM_BTREE			
Free-space manager address^{0} for H5FD_MEM_DRAW			
Free-space manager address^{0} for H5FD_MEM_GHEAP			
Free-space manager address^{0} for H5FD_MEM_LHEAP			
Free-space manager address^{0} for H5FD_MEM_OHDR			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: File Space Info

Field Name	Description
Version	This is version 0 of this message.
Strategy	This is the file space strategy used to manage file space. There are four types: **Value** **Description** 1 H5F_FILE_SPACE_ALL_PERSIST 2 H5F_FILE_SPACE_ALL 3 H5F_FILE_SPACE_AGGR_VFD 4 H5F_FILE_SPACE_VFD
Threshold	This is the smallest free-space section size that the free-space manager will track.
Free-space manager addresses	These are the six free-space manager addresses for the six file space allocation types: • H5FD_MEM_SUPER • H5FD_MEM_BTREE • H5FD_MEM_DRAW • H5FD_MEM_GHEAP • H5FD_MEM_LHEAP • H5FD_MEM_OHDR Note that these six fields exist only if the value for the field “*Strategy*” is H5F_FILE_SPACE_ALL_PERSIST.

Layout: File Space Info - Version 1

byte	byte	byte	byte
Version	Strategy	Persisting free-space	*This space inserted only to align table nicely*
Free-space Section Threshold[L]			
File Space Page Size			
Page-end Metadata threshold		*This space inserted only to align table nicely*	
EOA[0]			
Address[O] of small-sized free-space manager for H5FD_MEM_SUPER			
Address[O] of small-sized free-space manager for H5FD_MEM_BTREE			
Address[O] of small-sized free-space manager for H5FM_MEM_DRAW			
Address[O] of small-sized free-space manager for H5FD_MEM_GHEAP			
Address[O] of small-sized free-space manager for H5FD_MEM_LHEAP			
Address[O] of small-sized free-space manager for H5FD_MEM_OHDR			
Address[O] of large-sized free-space manager for H5FD_MEM_SUPER			
Address[O] of large-sized free-space manager for H5FD_MEM_BTREE			
Address[O] of large-sized free-space manager for H5FM_MEM_DRAW			
Address[O] of large-sized free-space manager for H5FD_MEM_GHEAP			

Address^O of large-sized free-space manager for H5FD_MEM_LHEAP
Address^O of large-sized free-space manager for H5FD_MEM_OHDR

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

Fields: File Space Info

Field Name	Description
Version	This is version 1 of this message.
Strategy	This is the file space strategy used to manage file space. There are four types: **Value** **Description** 0 H5F_FSPACE_STRATEGY_FSM_AGGR 1 H5F_FSPACE_STRATEGY_PAGE 2 H5F_FSPACE_STRATEGY_AGGR 3 H5F_FSPACE_STRATEGY_NONE
Persisting free-space	True or false in persisting free-space.
Free-space Section Threshold	This is the smallest free-space section size that the free-space manager will track.
File space page size	This is the file space page size, which is used when the paged aggregation feature is enabled.
Page-end metadata threshold	This is the smallest free-space section size at the end of a page that the free-space manager will track. This is used when the paged aggregation feature is enabled.
EOA	The EOA before the allocation of free-space manager header and section info for the self-referential free-space managers when persisting free-space. Note that self-referential free-space managers are managers that involve file space allocation for the managers' free-space header and section info.
Addresses of small-sized free-space managers	These are the addresses of the six small-sized free-space managers for the six file space allocation types: • H5FD_MEM_SUPER • H5FD_MEM_BTREE • H5FD_MEM_DRAW • H5FD_MEM_GHEAP • H5FD_MEM_LHEAP • H5FD_MEM_OHDR Note that these six fields exist only if the value for the field “*Persisting free-space*” is true.

Addresses of large-sized free-space managers	These are the addresses of the six large-sized free-space managers for the six file space allocation types: • H5FD_MEM_SUPER • H5FD_MEM_BTREE • H5FD_MEM_DRAW • H5FD_MEM_GHEAP • H5FD_MEM_LHEAP • H5FD_MEM_OHDR Note that these six fields exist only if the value for the field "*Persisting free-space*" is true.

IV.B. Disk Format: Level 2B - Data Object Data Storage

The data for an object is stored separately from its header information in the file and may not actually be located in the HDF5 file itself if the header indicates that the data is stored externally. The information for each record in the object is stored according to the dimensionality of the object (indicated in the dataspace header message). Multi-dimensional array data is stored in C order; in other words, the "last" dimension changes fastest.

Data whose elements are composed of atomic datatypes are stored in IEEE format, unless they are specifically defined as being stored in a different machine format with the architecture-type information from the datatype header message. This means that each architecture will need to [potentially] byte-swap data values into the internal representation for that particular machine.

Data with a variable-length datatype is stored in the global heap of the HDF5 file. Global heap identifiers are stored in the data object storage.

Data whose elements are composed of reference datatypes are stored in several different ways depending on the particular reference type involved. Object pointers are just stored as the offset of the object header being pointed to with the size of the pointer being the same number of bytes as offsets in the file.

Dataset region references are stored as a heap-ID which points to the following information within the file-heap: an offset of the object pointed to, number-type information (same format as header message), dimensionality information (same format as header message), sub-set start and end information (in other words, a coordinate location for each), and field start and end names (in other words, a [pointer to the] string indicating the first field included and a [pointer to the] string name for the last field).

Data of a compound datatype is stored as a contiguous stream of the items in the structure, with each item formatted according to its datatype.

V. Appendix A: Definitions

Definitions of various terms used in this document are included in this section.

Term	**Definition**
Undefined Address	The undefined address for a file is a file address with all bits set: in other words, 0xffff...ff.
Unlimited Size	The unlimited size for a size is a value with all bits set: in other words, 0xffff...ff.

VI. Appendix B: File Space Allocation Types

There are six basic types of file space allocation as follows:

Basic Allocation Type	Description
H5FD_MEM_SUPER	File space allocated for *Superblock.*
H5FD_MEM_BTREE	File space allocated for *B-tree.*
H5FD_MEM_DRAW	File space allocated for *raw data*.
H5FD_MEM_GHEAP	File space allocated for *Global Heap.*
H5FD_MEM_LHEAP	File space allocated for *Local Heap.*
H5FD_MEM_OHDR	File space allocated for *Object Header.*

There are other file space allocation types that are mapped to the above six basic types because they are similar in nature. The mapping and the corresponding description are listed in the following two tables:

Basic Allocation Type	Mapping of Allocation Types to Basic Allocation Types
H5FD_MEM_SUPER	*none*
H5FD_MEM_BTREE	H5FD_MEM_SOHM_INDEX
H5FD_MEM_DRAW	H5FD_MEM_FHEAP_HUGE_OBJ
H5FD_MEM_GHEAP	*none*
H5FD_MEM_LHEAP	H5FD_MEM_FHEAP_DBLOCK, H5FD_MEM_FSPACE_SINFO
H5FD_MEM_OHDR	H5FD_MEM_FHEAP_HDR, H5FD_MEM_FHEAP_IBLOCK, H5FD_MEM_FSPACE_HDR, H5FD_MEM_SOHM_TABLE

Allocation Type	Description
H5FD_MEM_FHEAP_HDR	File space allocated for *Fractal Heap Header.*
H5FD_MEM_FHEAP_DBLOCK	File space allocated for *Fractal Heap Direct Blocks.*
H5FD_MEM_FHEAP_IBLOCK	File space allocated for *Fractal Heap Indirect Blocks.*
H5FD_MEM_FHEAP_HUGE_OBJ	File space allocated for huge objects in the fractal heap.
H5FD_MEM_FSPACE_HDR	File space allocated for *Free-space Manager Header.*
H5FD_MEM_FSPACE_SINFO	File space allocated for *Free-space Section List* of the free-space manager.
H5FD_MEM_SOHM_TABLE	File space allocated for *Shared Object Header Message Table.*
H5FD_MEM_SOHM_INDEX	File space allocated for *Shared Message Record List.*

VII. Appendix C: Types of Indexes for Dataset Chunks

For an HDF5 file without the latest format enabled, the library uses the Version 1 B-tree to index dataset chunks.

For an HDF5 file with the latest format enabled, the library uses one of the following five indexing types depending on a chunked dataset's dimension specification and the way it is extended.

VII.A. The Single Chunk Index

The *Single Chunk* index can be used when the dataset fulfills the following condition:

- the current, maximum, and chunk dimension sizes are all the same

The dataset has only one chunk, and the address of the single chunk is stored in the version 4 *Data Layout* message. See the Chunked Storage Property Description layout and field description tables.

VII.B. The Implicit Index

The *Implicit* index can be used when the dataset fulfills the following conditions:

- fixed maximum dimension sizes
- no filter applied to the dataset
- the timing for the space allocation of the dataset chunks is H5P_ALLOC_TIME_EARLY

Since the dataset's dimension sizes are known and storage space is to be allocated early, an array of dataset chunks are allocated based on the maximum dimension sizes when the dataset is created. The base address of the array is stored in the version 4 *Data Layout* message. See the Chunked Storage Property Description layout and field description tables.

When accessing a dataset chunk with a specified offset, the address of the chunk in the array is computed as below:

> base address + (size of a chunk in bytes * chunk index associated with the offset)

A chunk index starts at 0 and increases according to the fastest changing dimension, then the next fastest, and so on. The chunk index for a dataset chunk offset is computed as below:

1. Calculate the scaled offset for each dimension in scaled_offset:

   ```
   scaled_offset = chunk_offset/chunk_dims
   ```

2. Calculate the # of chunks for each dimension in nchunks:

   ```
   nchunks = (curr_dims + chunk_dims - 1)/chunk_dims
   ```

3. Calculate the down chunks for each dimension in down_chunks:

   ```
   /* n is the # of dimensions */
   for(i = (int)(n-1), acc = 1; i >= 0; i--) {
      down_chunks[i] = acc;
      acc *= nchunks[i];
   }
   ```

4. Calculate the chunk index in chunk_index:

   ```
   /* n is the # of dimensions */
   for(u = 0, chunk_index = 0; u < n; u++)
      chunk_index += down_chunks[u] * scaled_offset[u];
   ```

For example, for a 2-dimensional dataset with curr_dims[4,5] and chunk_dims[3,2], there will be a total of 6 chunks, with 3 chunks in the fastest changing dimension and 2 chunks in the slowest changing dimension. See the figure below. The chunk index for the chunk offset [3,4] is computed as below:

1. scaled_offset[0] = 1, scaled_offset[1] = 2
2. nchunks[0] = 2, nchunks[1] = 3
3. down_chunks[0] = 3, down_chunks[1] = 1
4. chunk_index = 5

Figure 3. Implicit index chunk diagram

VII.C. The Fixed Array Index

The *Fixed Array* index can be used when the dataset fulfills the following condition:

- fixed maximum dimension sizes

Since the maximum number of chunks is known, an array of in-file-on-disk addresses based on the maximum number of chunks is allocated when data is written to the dataset. To access a dataset chunk with a specified offset, the chunk index associated with the offset is calculated. The index is mapped into the array to locate the disk address for the chunk.

The Fixed Array (FA) index structure provides space and speed improvements in locating chunks over index structures that handle more dynamic data accesses like a Version 2 B-tree index. The entry into the Fixed Array is the Fixed Array header which contains metadata about the entries stored in the array. The header contains a pointer to a data block which stores the array of entries that describe the dataset chunks. For greater efficiency, the array will be divided into multiple pages if the number of entries exceeds a threshold value. The space for the data block and possibly data block pages are allocated as a single contiguous block of space.

The content of the data block depends on whether paging is activated or not. When paging is not used, elements that describe the chunks are stored in the data block. If paging is turned on, the data block contains a bitmap indicating which pages are initialized. Then subsequent data block pages will contain the entries that describe the chunks.

An entry describes either a filtered or non-filtered dataset chunk. The formats for both element types are described below.

Layout: Fixed Array Header

byte	byte	byte	byte
Signature			
Version	Client ID	Entry Size	Page Bits
Max Num Entries^{L}			
Data Block Address^{O}			
Checksum			

(Items marked with an ‘L’ in the above table are of the size specified in the Size of Lengths field in the superblock.)

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Fixed Array Header

Field Name	Description
Signature	The ASCII character string “FAHD” is used to indicate the beginning of a Fixed Array header. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The ID for identifying the client of the Fixed Array: **ID** **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks 2+ Reserved
Entry Size	The size in bytes of an entry in the Fixed Array.
Page Bits	The number of bits needed to store the maximum number of entries in a data block page.
Max Num Entries	The maximum number of entries in the Fixed Array.
Data Block Address	The address of the data block in the Fixed Array.
Checksum	The checksum for the header.

Layout: Fixed Array Data Block

byte	byte	byte	byte
Signature			
Version	Client ID	*This space inserted only to align table nicely*	
Header Address^O			
Page Bitmap *(variable size and optional)*			
Elements *(variable size and optional)*			
Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Fixed Array Data Block

Field Name	Description
Signature	The ASCII character string "FADB" is used to indicate the beginning of a Fixed Array data block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The ID for identifying the client of the Fixed Array: **ID** **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks 2+ Reserved.
Header Address	The address of the Fixed Array header. Principally used for file integrity checking.
Page Bitmap	A bitmap indicating which data block pages are initialized. Exists only if the data block is paged.
Elements	Contains the elements stored in the data block and exists only if the data block is not paged. There are two element types: **ID** **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks
Checksum	The checksum for the Fixed Array data block.

Layout: Fixed Array Data Block Page

byte	byte	byte	byte
Elements *(variable size)*			
Checksum			

Fields: Fixed Array Data Block Page

Field Name	Description
Elements	Contains the elements stored in the data block page. There are two element types: **ID** / **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks
Checksum	The checksum for a Fixed Array data block page.

Layout: Data Block Element for Non-filtered Dataset Chunk

byte	byte	byte	byte
Address^{O}			

(Items marked with an ‘O’ in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Data Block Element for Non-filtered Dataset Chunk

Field Name	Description
Address	The address of the dataset chunk in the file.

Layout: Data Block Element for Filtered Dataset Chunk

byte	byte	byte	byte
Address^O			
Chunk Size *(variable size; at most 8 bytes)*			
Filter Mask			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Data Block Element for Filtered Dataset Chunk

Field Name	Description
Address	The address of the dataset chunk in the file.
Chunk Size	The size of the dataset chunk in bytes.
Filter Mask	Indicates the filter to skip for the dataset chunk. Each filter has an index number in the pipeline; if that filter is skipped, the bit corresponding to its index is set.

VII.D. The Extensible Array Index

The *Extensible Array* index can be used when the dataset fulfills the following condition:

- only one dimension of unlimited extent

The Extensible Array (EA) is a data structure that is used as a chunk index in datasets where the dataspace has a single unlimited dimension. In other words, one dimension is set to H5S_UNLIMITED, and the other dimensions are any number of fixed-size dimensions. The idea behind the extensible array is that a particular data object can be located via a lightweight indexing structure of fixed depth for a given address space. This indexing structure requires only a few (2-3) file operations per element lookup and gives good cache performance. Unlike the B-tree structure, the extensible array is optimized for appends. Where a B-tree would always add at the rightmost node under these circumstances, either creating a deep tree (version 1) or requiring expensive rebalances to correct (version 2), the extensible array has already mapped out a pre-balanced internal structure. This optimized internal structure is instantiated as needed when chunk records are inserted into the structure.

An Extensible Array consists of a header, an index block, secondary blocks, data blocks, and (optional) data block pages. The general scheme is that the index block is used to reference a secondary block, which is, in turn, used to reference the data block page where the chunk information is stored. The data blocks will be paged for efficiency when their size passes a threshold value. These pages are laid out contiguously on the disk after the data block, are initialized as needed, and are tracked via bitmaps stored in the secondary block. The number of secondary and data blocks/pages in a chunk index varies as they are allocated as needed and the first few are (conceptually) stored in parent elements as an optimization.

Layout: Extensible Array Header

byte	byte	byte	byte
Signature			
Version	Client ID	Element Size	Max Nelmts Bits
Index Blk Elmts	Data Blk Min Elmts	Secondary Blk Min Data Ptrs	Max Data Blk Page Nelmts Bits
Num Secondary Blks^L			
Secondary Blk Size^L			
Num Data Blks^L			
Data Blk Size^L			
Max Index Set^L			
Num Elements^L			
Index Block Address^O			
Checksum			

(Items marked with an 'L' in the above table are of the size specified in the Size of Lengths field in the superblock.)

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Extensible Array Header

Field Name	Description
Signature	The ASCII character string “EAHD” is used to indicate the beginning of an Extensible Array header. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The ID for identifying the client of the Fixed Array: **ID** **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks 2+ Reserved.
Element Size	The size in bytes of an element in the Extensible Array.
Max Nelmts Bits	The number of bits needed to store the maximum number of elements in the Extensible Array.
Index Blk Elmts	The number of elements to store in the index block.
Data Blk Min Elmts	The minimum number of elements per data block.
Secondary Blk Min Data Ptrs	The minimum number of data block pointers for a secondary block.
Max Dblk Page Nelmts Bits	The number of bits needed to store the maximum number of elements in a data block page.
Num Secondary Blks	The number of secondary blocks created.
Secondary Blk Size	The size of the secondary blocks created.
Num Data Blks	The number of data blocks created.
Data Blk Size	The size of the data blocks created.
Max Index Set	The maximum index set.

Num Elmts	The number of elements realized.
Index Block Address	The address of the index block.
Checksum	The checksum for the header.

Layout: Extensible Array Index Block

byte	byte	byte	byte
Signature			
Version	Client ID	*This space inserted only to align table nicely*	
Header Address[O]			
Elements *(variable size and optional)*			
Data Block Addresses *(variable size and optional)*			
Secondary Block Addresses *(variable size and optional)*			
Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Extensible Array Index Block

Field Name	Description
Signature	The ASCII character string “EAIB” is used to indicate the beginning of an Extensible Array Index Block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The client ID for identifying the user of the Extensible Array: **ID** — **Description** 0 — Non-filtered dataset chunks 1 — Filtered dataset chunks 2+ — Reserved.
Header Address	The address of the Extensible Array header. Principally used for file integrity checking.
Elements	Contains the elements that are stored directly in the index block. An optimization to avoid unnecessary secondary blocks. There are two element types: **ID** — **Description** 0 — Non-filtered dataset chunks 1 — Filtered dataset chunks
Data Block Addresses	Contains the addresses of the data blocks that are stored directly in the Index Block. An optimization to avoid unnecessary secondary blocks.
Secondary Block Addresses	Contains the addresses of the secondary blocks.
Checksum	The checksum for the Extensible Array Index Block.

Layout: Extensible Array Secondary Block

byte	byte	byte	byte
Signature			
Version	Client ID	*This space inserted only to align table nicely*	
Header Address^{O}			
Block Offset *(variable size)*			
Page Bitmap *(variable size and optional)*			
Data Block Addresses *(variable size and optional)*			
Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Extensible Array Secondary Block

Field Name	Description
Signature	The ASCII character string “EASB” is used to indicate the beginning of an Extensible Array Secondary Block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The ID for identifying the client of the Extensible Array: **ID** **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks 2+ Reserved.
Header Address	The address of the Extensible Array header. Principally used for file integrity checking.
Block Offset	Stores the offset of the block in the array.
Page Bitmap	A bitmap indicating which data block pages are initialized. Exists only if the data block is paged.
Data Block Addresses	Contains the addresses of the data blocks referenced by this secondary block.
Checksum	The checksum for the Extensible Array Secondary Block.

Layout: Extensible Array Data Block

byte	byte	byte	byte
Signature			
Version	Client ID	*This space inserted only to align table nicely*	
Header Address^{O}			
Block Offset *(variable size)*			
Elements *(variable size and optional)*			
Checksum			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Extensible Array Data Block

Field Name	Description
Signature	The ASCII character string “EADB” is used to indicate the beginning of an Extensible Array data block. This gives file consistency checking utilities a better chance of reconstructing a damaged file.
Version	This document describes version 0.
Client ID	The ID for identifying the client of the Extensible Array: **ID** — **Description** 0 — Non-filtered dataset chunks 1 — Filtered dataset chunks 2+ — Reserved.
Header Address	The address of the Extensible Array header. Principally used for file integrity checking.
Block Offset	The offset of the block in the array.
Elements	Contains the elements stored in the data block and exists only if the data block is not paged. There are two element types: **ID** — **Description** 0 — Non-filtered dataset chunks 1 — Filtered dataset chunks
Checksum	The checksum for the Extensible Array data block.

Layout: Extensible Array Data Block Page

byte	byte	byte	byte
Elements *(variable size)*			
Checksum			

Fields: Extensible Array Data Block Page

Field Name	Description
Elements	Contains the elements stored in the data block page. There are two element types: **ID** **Description** 0 Non-filtered dataset chunks 1 Filtered dataset chunks
Checksum	The checksum for an Extensible Array data block page.

Layout: Data Block Element for Non-filtered Dataset Chunk

byte	byte	byte	byte
Address^{O}			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Data Block Element for Non-filtered Dataset Chunk

Field Name	Description
Address	The address of the dataset chunk in the file.

Layout: Data Block Element for Filtered Dataset Chunk

byte	byte	byte	byte
Address^O			
Chunk Size *(variable size; at most 8 bytes)*			
Filter Mask			

(Items marked with an 'O' in the above table are of the size specified in the Size of Offsets field in the superblock.)

Fields: Data Block Element for Filtered Dataset Chunk

Field Name	Description
Address	The address of the dataset chunk in the file.
Chunk Size	The size of the dataset chunk in bytes.
Filter Mask	Indicates the filter to skip for the dataset chunk. Each filter has an index number in the pipeline; if that filter is skipped, the bit corresponding to its index is set.

VII.E. The Version 2 B-trees Index

The *Version 2 B-trees* index can be used when the dataset fulfills the following condition:

- more than one dimension of unlimited extent

Version 2 B-trees can be used to index various objects in the library. See "Version 2 B-trees" for more information. The B-tree types 10 and 11 record layouts are for indexing dataset chunks.