RFC: New public functions to handle comparison

Peter Cao

Neil Fortner

Quincey Koziol

Vailin Choi

This RFC principally describes a new public API function, *H5Ocompare* that compares two HDF5 objects. This is performed according to the set of rules for comparing two HDF5 files or two HDF5 objects specified in the “HDF5 File and Object Comparison Specification[[1]](#footnote-1)”, which provides details of how two objects should be compared and guidelines for how *H5Ocompare* should be implemented.

This RFC also describes five new public functions: *H5Fcompare\_md*, which compares two files’ file metadata, *H5Pget/set\_compare*, which manipulate properties for the comparison, and *H5Pset/get\_compare\_epsilon*, which sets/gets the precision limit when comparing floating-point data values.

# Introduction

An HDF5 file appears to the user as a directed graph with three higher-level objects that are exposed by the HDF5 APIs: groups, datasets, and committed datatypes. The HDF5 model provides great flexibility with regard to how a file can be structured. At the same time, it creates challenges in determining how to compare two files or two objects.

A command line tool, *h5diff*, was developed to compare two HDF5 files or objects and report the differences. The tool is one of the most used tools. However, four major issues cannot be easily resolved with the current implementation of *h5diff*:

* **Performance:** *h5diff* has poor performance in some cases. Although the performance has greatly improved over time, some issues cannot be resolved given the current implementation. For example, uncompressing data to compare dataset values is a major bottleneck for compressed datasets; it can take over 80% of *h5diff’s* total time when comparing large datasets. It would be far better to perform the comparison within the library, directly on the compressed data when possible.
* **Reusability and Flexibility:** The code for *h5diff* incorporates all the logic and control flow for comparing objects and files. Users have no way to reuse this functionality in applications they write. Moving the object and file comparison operations within the HDF5 library and providing API routines to access them will allow application developers to use this functionality in their own applications.
* **Complexity:** The code for *h5diff* is very complex. Features and options have been added to it in an *ad hoc* way making the code complex and prone to errors. Over the years, a great deal of effort has been spent fixing problems, but it is becoming increasingly difficult to maintain. Directly moving this object and file comparison code into the library would be a daunting and failure prone process.
* **Completeness:** *h5diff* was implemented with insufficient design and definition. When two objects are compared, *h5diff* does not provide sufficient details of what should be examined. For example, when two datasets are compared, creation properties such as storage layout are not compared. There is no clear definition of what should be compared and what should be not. One of the main goals of this document is to provide a clear and complete set of definitions for the *H5Ocompare* function on what to compare. Although these defects could be corrected in *h5diff*, the resulting changes would only add to the complexity of its implementation.

The purpose of the main routine to add, *H5Ocompare,* is to address the problems above. Other advantages of having an *H5Ocompare* function include the following:

* Any tool built on *H5Ocompare* should have less code to maintain since the main work will be done by the function. Applications built on the function should be simple and specific.
* *H5Ocompare* provides function callbacks to application code thus allowing application developers the flexibility to choose how to react to object and file differences.
* The function can be used in high-level languages such as Java, Fortran, C++, and Python.

We will also develop a new tool that is based on *H5Ocompare*. The new tool is not targeted to replace the current *h5diff*; it is intended to address the problems of the current *h5diff*. The new tool will not try to mimic the output format of the current *h5diff*; however, the new tool will adopt some useful elements of the *h5diff* output format. This tool will be described in a separate, forthcoming, RFC.

This RFC presents a number of options regarding how objects are compared. Although not all options will be implemented in the first stage, the design of the new function should allow options to be added incrementally.

This RFC also proposes five new auxiliary public functions as follows:

* *H5Fcompare\_md*: This function compares file-level metadata. Separating the comparison of file metadata from the object comparison done by *H5Ocompare* provides a more coherent API to developers. This allows the root group of each file to be treated in the same way as other groups.
* *H5Pset\_compare*: This function provides options that allow users to modify the default comparison done by *H5Ocompare*.
* *H5Pget\_compare*: This function retrieves the user-modified properties set for the comparison.
* *H5Pset\_compare\_epsilon*: This function allows users to set the precision limit when comparing floating-point values.
* *H5Pget\_compare\_epsilon*: This function retrieves the precision limit used when comparing floating-point values.

# Comparing Objects

This section describes how *H5Ocompare* will compare two objects. Options, such as excluding certain metadata, can be applied to the comparison by calling *H5Pset\_compare*.

## Groups

When two groups are compared, these items will form the basis of the comparison: group creation properties, attributes, and the links contained within the two groups.

### What is Compared

The following is a list of the things to be compared for groups:

* Group creation properties, which include
	+ Link creation order
	+ Link storage layout (compact or dense)
* Attributes attached to the group (details in Attribute section)
* Links within the group (details in Link section)

### Options

The following options can be set when comparing groups:

* Exclude comparing creation properties. By default, creation properties will be compared.
* Exclude comparing attributes. By default, attributes will be compared (see Attribute section below).

## Links

A link is owned by a group and has a name, type, and value. The link’s value points to an existing object or a non-existing object (symbolic links only).

### What is Compared

Link characteristics to be compared include:

* creation properties (link name character encoding and link creation order)
* link name
* link type (hard, soft, external, or user-defined)
* link value (for symbolic link only)

### Options

Options for comparing links include the following:

* Compare links according to creation order. By default, links will be compared based on their name.
* Exclude comparing creation properties. By default, creation properties will be checked.
* Exclude comparing link type. By default, link types will be compared. Links of different types, e.g. one is an external link and the other is a soft link, will be reported as different.
* Exclude comparing link value (for symbolic links). By default link values will be compared for symbolic links.
* Exclude comparing links with same name or creation order. Links are determined to be the same based on the index currently used for the comparison: either by name or creation order.
* Exclude reporting unique links, i.e. links that exist in either one of the two groups. By default, unique links will be reported as different. Unique links are determined based on the index currently used for the comparison, either by name or creation order.

## Datasets

When two datasets are compared, the datasets’ metadata and attributes, as well as the data values of the two datasets will be compared.

### What is compared

Comparing datasets means comparing both the data values and metadata of the datasets, unless an option excluding a specific comparison is given. The following things will be compared:

* Data values (details in Data Values section)
* Dataset creation properties, which include:
	+ Layout of the raw data
	+ Size of chunks for the raw data of a chunked layout dataset
	+ Fill value if it is defined
	+ Filter(s) applied to the datasets (e.g. compression)
* Attributes attached to the dataset (details in Attribute section)
* Datatype (details in Datatypes section)
* Dataspace (details in Dataspace section)

### Options

Options for comparing datasets are listed below. Options for comparing data values, datatypes and dataspaces are presented in separate sections.

* Exclude comparing creation properties. By default, creation properties will be compared.
* Exclude comparing attributes. By default, attributes will be compared (see Attributes section).

## Attributes

An attribute is like a small dataset; it has a datatype, a dataspace, and data values. By default, attributes are matched by name, and comparing two attributes is the same as comparing two datasets except that attributes currently do not have creation and access properties. Unique attributes (attributes that exist only in one of the two objects) will be reported as different.

### What is compared

Comparing attributes means comparing both the data values and metadata of the attributes, unless an option excluding a specific comparison is given. The following things will be compared:

* Data values (see Data Values section)
* Creation properties (attribute name character encoding and attribute creation order)
* Datatype (details in Datatypes section)
* Dataspace (details in Dataspace section)

### Options

The following options can be set when comparing attributes:

* Compare attributes according to creation order. By default, attributes will be compared based on their name.
* Exclude comparing attributes with same name or creation order. Attributes are determined to be the same based on the index currently used for the comparison: either by name or creation order.
* Exclude reporting unique attributes, i.e. attributes that exist only in one of the two objects. By default, unique attributes will be reported as different. Unique attributes are determined based on the index currently used for the comparison, either by name or creation order.

## Datatypes

A datatype can describe an atomic type like a fixed- or floating-point type, or more complex types like a C structure (compound datatype), array (array datatype), or C++ vector (variable-length datatype). A datatype is defined by its class and class-specific properties.

### What is compared

The following datatype characteristics will be compared:

* Datatype class (e.g. integer (H5T\_INTEGER), float (H5T\_FLOAT), string (H5T\_STRING), etc.)
* Class-specific properties (e.g. size, signed or unsigned, byte order, etc.)
	+ H5T\_INTEGER – Size (bytes), precision (bits), offset (bits), padding, byte order, signed/unsigned
	+ H5T\_FLOAT -- Size (bytes), precision (bits), offset (bits), padding, byte order, and field information
	+ H5T\_TIME – Size (bytes), precision (bits), byte order
	+ H5T\_STRING – Size (fixed or variable), character set, pad/no pad, pad character
	+ H5T\_BITFIELD -- Size (bytes), precision (bits), offset (bits), padding, and byte order
	+ H5T\_OPAQUE -- Size (bytes), tag
	+ H5T\_COMPOUND – Size (bytes), number of members, member names, member datatypes, member offsets
	+ H5T\_REFERENCE -- Reference type (object or region)
	+ H5T\_ENUM -- Number of elements, element names, element values, base datatype
	+ H5T\_VLEN -- Base datatype
	+ H5T\_ARRAY -- Number of dimensions, dimension sizes, base datatype
* Attributes for committed datatypes (see Attributes section)

Note that when the datatype classes are not the same, *H5Ocompare* will report non-comparable via callback and will not continue comparison of class-specific properties.

### Options

Options for comparing committed datatypes are listed below:

* Exclude comparing attributes. By default, attributes on committed datatypes will be compared (see Attributes section).

## Dataspace

A dataspace describes the rank and the size of each dimension in the data object array.

### What is compared

Dataspace characteristics to be compared include:

* rank (the number of dimensions)
* current dimension sizes
* maximum dimension sizes

Note that when the ranks are not the same, *H5Ocompare* will report non-comparable via callback and will not continue comparison of current and maximum dimension sizes.

### Options

Options for comparing dataspace include:

* Exclude comparing maximum dimension sizes. By default, they will be compared.

## Data Values

The comparison of data values depends on the datatypes and dataspaces of the datasets or attributes the values belong to. Which data values will be compared depends on the dataspace of the dataset or attribute and how the data values are compared depends on the datatype of the dataset or attribute.

### Dataspace

The rank and dimensions of the dataspace for each of the objects being compared controls which data values are compared:

* If the ranks are not same, see description in Dataspace section.
* If the ranks are the same but the dataspaces’ dimension sizes are different, *H5Ocompare* will compare the overlapping regions starting from the origin. The shaded areas in the following examples are the compared regions. *H5Ocompare* will indicate the data values as different for the non-common regions and will report any differences found for the common regions via callback.

 Example1: space1[6x8]; space2[3x5] Example2: space1[2x8]; space2[4x1]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |
|  |

### Datatype

The datatype class (integer, floating-point, compound, etc) and properties for each of the objects being compared control how data values are compared:

* If the datatype classes are not the same, see description in Datatype section.
* If the datatype classes are the same, *H5Ocompare* will try to compare the data values. If the datatypes within the datatype class are not the same, *H5Ocompare* will try to convert data values if needed before the comparison. For example, when comparing a signed 8-bit integer and an unsigned 16-bit integer, *H5Ocompare* will convert both datasets’ values to signed 32-bit integers before comparing. If the signed and unsigned integers have the same precision but are the maximum precision supported in HDF5 (64-bits, currently), *H5Ocompare* will not do any conversion and will report non-comparable via callback.

Note that when comparing non-atomic datatype classes that are the same:

* enum, array, variable-length: these rules will apply to the base datatypes.
* compound: these rules will apply recursively through the nested fields.

###  Comparing Data Values

The comparison of data values will proceed when the dataspace ranks and datatype classes are the same as follows*:*

* If the datatypes within the datatype class are not the same:
	+ If conversion of the data values is not possible, *H5Ocompare* will report that the data values are non-uncomparable via callback; this will be the only callback made for the comparison of data values.
	+ If conversion of the data values is possible, *H5Ocompare* will convert the data values and then continue the comparison as described next for same datatypes.
* If the datatypes within the datatype class are the same:
	+ If the dimension sizes of the dataspaces are not the same, *H5Ocompare* will compare data values of the overlapping regions starting from the origin as described below and will report elements in non-common regions as different via callback.
	+ If the dimension sizes of the dataspaces are the same, *H5Ocompare* will compare all data values of the two regions as described below.

*H5Ocompare* will compare data values element by element with respect to the datatype class:

* Two integer (H5T\_INTEGER ) numbers of any precision and of the same type (signed/signed, unsigned/unsigned) can be directly compared. If the datatypes within the class are not the same, *H5Ocompare* will try to convert data values if needed before the comparison. See above rules described for datatypes.
* Comparing two float (H5T\_FLOAT) numbers is more complicated. To determine whether two floating-point values, float1 and float2, are different, one cannot use the simple comparison of (float1 == float2). Two floating-point values can be the same while (float1 == float2) may appear to differ because of floating-point precision. In HDF5, the precision limit can be set when comparing floating-point values; for details, see the “*Default EPSILON Values for Comparing Floating Point Data*[[2]](#footnote-2)” RFC. We propose two new public functions to set and retrieve the EPSILON value when comparing floating-point data values (*H5Pget/set\_epsilon*). Second, Not-a-Number (NaN) values need to be handled. By definition, two NaNs are always equal. A NaN and a regular number are always different. By default, *H5Ocompare* will check NaNs, however, checking for NaN is very expensive. If a user knows there are no NaNs in the datasets, they can skip checking NaN for better performance with an option.
* Strings (H5T\_STRING) data values are compared with the Standard C *strcmp()* function initially. When two strings are different, i.e. *strcmp*(str1, str2) is non-zero, the strings will be compared character by character and the differences will be reported.
* Data values with type class of H5T\_BITFIELD, H5T\_OPAQUE or H5T\_TIME will be compared byte by byte.
* Compound (H5T\_COMPOUND) data values will be compared by field according to the type of the field. For nested compound types, the comparison will recur through the nested fields. The comparison of the field values will follow the rules described above for datatypes.
* Currently HDF5 has two types of reference datatypes (H5T\_REFERENCE): object references and dataset region references. The following comparison is performed when comparing references:
	+ Object reference: compare the names of the object pointed to by the reference
	+ Dataset region reference: compare the names of the objects pointed to by the references and the selection of the referenced regions
* An enumerated (H5T\_ENUM) datatype is a set of [name, value] pairs with a base datatype. The comparison of the base datatype follows the same rules as described above for datatypes. The member names and their corresponding values are then compared.
* Each element of variable length datatype (H5T\_VLEN) is a one-dimensional sequence of data values, of a particular base datatype. The comparison of the base datatype follows the same rules as described above for datatypes. If the lengths are the same, each data value in the sequences will be compared. If the lengths are different, data values will be compared up to the shortest one. Extra values of the longer array will be reported as different.
* Values of array datatypes (H5T\_ARRAY) will be compared according to the above rules described for datatype. If the arrays’ ranks are the same but their dimension sizes are different, *H5Ocompare* will compare the overlapping regions common to both arrays and report the non-common regions as different. If the arrays’ ranks are not the same, *H5Ocompare* will report non-comparable via callback.

### Options

* Enable the comparison of data values only if the dataspaces have the same rank and the same dimension sizes. By default, *H5Ocompare* will compare data values according to the rules described previously for dataspaces.
* Enable the comparison of data values only if the datatypes have the same class and the same datatypes within the class. By default, *H5Ocompare* will compare data values according to the rules described previously for datatypes.
* Skip checking Not-a-Number (NaN) floating-point values. By default, *H5Ocompare* will check NaNs.

# Comparing File metadata

This section describes how the new public function, *H5Fcompare\_md*, compares the file metadata of two HDF5 files. Each HDF5 file contains file metadata such as file creation properties. When comparing the metadata, file creation properties will be checked. This includes:

* version number of super block,
* size of user block,
* size of addresses,
* size of lengths,
* sizes used to control symbol tables (B-tree rank and node size),
* tree rank used to control B-trees for indexing chunked datasets,
* number of shared message indexes,
* configuration settings for a shared message index (type and minimum size of messages),
* threshold values for storing shared messages: maximum number of messages to store in a compact list, minimum number of messages to store in a B-tree).

The result is determined from comparing the above information.

# File comparison

Each HDF5 file contains file metadata, a root group, and zero or more links to other objects. A hard link’s value points to an existing object such as a group, dataset or named datatype. A symbolic link’s value which might or might not point to an existing object.

To compare two files, applications can use *H5Fcompare\_md* to compare the two files’ file metadata, then iterate through the links and objects in the two files, and compare each pair of objects via *H5Ocompare.*

The following special cases of file comparison are handled as described below:

1. A file is compared to itself. There should be no difference if a file is compared to itself. *H5Fcompare\_md* and *H5Ocompare* should quickly confirm that it is the same file -- without going through the actual comparison.
2. Two identical files are compared. There should be no difference if two identical files are compared. This case is the same as (A) except that the two files are two separate physical files with exactly the same contents. Applications will need to use both *H5compare\_md* and *H5Ocompare* to verify that the contents are exactly the same.
3. Two empty files are compared. A file is empty if its root group does not have any links or attributes. If two empty files are compared, applications can compare the file metadata via *H5Fcompare\_md* and then compare the root group of the two files via *H5Ocompare*. If the file metadata and group’s creation properties are ignored, there should be no difference between two empty files. Otherwise, the result depends on the comparison of the file metadata and the root group’s creation properties.
4. An empty file is compared with a non-empty file. The result varies according to the comparison options. By default, an empty file and a non-empty file should be different. If file metadata, group creation properties and attributes, and unique objects are excluded, there will be no difference.

# New public functions to handle comparison

This section describes the following six new public routines:

* *H5Ocompare*
* *H5Fcompare\_md*
* *H5Pset\_compare, H5Pget\_compare*
* *H5Pset\_compare\_epsilon, H5Pget\_compare\_epsilon*

## New public function for comparing objects

**Name:**

*H5Ocompare*

**Signature:**

htri\_t *H5Ocompare(hid\_t loc1\_id, const char \*name1, hid\_t lapl1, hid\_t loc2\_id, const char \*name2, hid\_t lapl2, hid\_t cmppl\_id, H5O\_cmp\_cb\_t \*cb\_info)*

**Purpose:**

Compare two objects in the same or different files.

**Description:**

H5Ocompare compares the object specified by name1 in the file or group specified by loc1\_id to the object specified by name2 in the file or group specified by loc2\_id.

The parameters *lapl1* and *lapl2* are link access property lists associated with the links *name1* and *name2* respectively.

Several properties are available to govern the behavior of H5Ocompare. These properties are set in the comparison property list, cmppl\_id with the new public function, H5Pset\_compare.

Differences in the two objects are reported via function callbacks, which are grouped together in a structure *H5O\_cmp\_cb\_*t as defined below. This structure is passed in as the *cb\_info* parameter to this routine along with a pointer to user’s data:

typedef struct H5O\_cmp\_cb\_t {

 H5O\_cmp\_link\_cb\_t link;

 H5O\_cmp\_obj\_md\_cb\_t obj\_md;

 H5O\_cmp\_attr\_md\_cb\_t attr\_md;

 H5O\_cmp\_dset\_data\_cb\_t dset\_data;

 H5O\_cmp\_attr\_data\_cb\_t attr\_data;

 void \*udata;

} H5O\_cmp\_cb\_t;

Details of these callbacks are described in the next sections.

On entry to *H5Ocompare*, the routine will try to resolve *name1* with respect to *loc1\_id* and *name2* with respect to *loc2\_id* to objects, using *lapl1* and *lapl2*, respectively. If not successful, *H5Ocompare* will return an error and exit. If successful, *H5Ocompare* will compare the two specified objects as follows:

1. Compares the metadata of the two objects, and invokes *obj\_md* callback for each difference found.
2. Compares the two objects’ common attributes, and reports any differences found in metadata via the *attr\_md* callback or raw data via the *data* callback. Unique attributes (attributes that exist only in one of the two objects) will be reported via callback also.
3. Compares the two objects:
	1. Datasets: *H5Ocompare* will compare the raw data and report all the differences found via the *data* callback.
	2. Named datatypes: *H5Ocompare* has already compared the class and properties of the datatypes in step (a) above.
	3. Groups: *H5Ocompare* will compare all the links in the two groups and report any differences found via the *link* callback. If recursive comparison is desired, applications will need to iterate links in the groups with another function and then perform object comparisons with further calls to *H5Ocompare*.

**Parameters:**

|  |  |
| --- | --- |
| *hid\_t* loc1\_id  | IN: Location identifier of the first object to be compared |
| *const char \**name1 *hid\_t* lapl1 | IN: Name of the first object to be comparedIN: Link access property list associated with the first object |
| *hid\_t* loc2\_id  | IN: Location identifier of the second object to be compared |
| *const char \**name2*hid\_t* lapl2      | IN: Name of the second object to be compared IN: Link access property list associated with the second object |
| *hid\_t* cmppl\_id  | IN: Comparison property list identifier  |
| *H5O\_cmp\_cb\_t \*cb\_info*  | IN/OUT: A callback structure that contains a list of function callbacks and a pointer to user’s data for reporting the results of comparison.  |

**Returns:**

The return value indicates the result of the comparison:

* True (1) if the two objects are equivalent (strict or loose[[3]](#footnote-3))
* False (0) if the two objects are not equivalent (strict or loose)
* Negative value if there is error

Note that two objects are strictly equivalent/not equivalent when default properties in the comparison property list are used when comparing; two objects are loosely equivalent/not equivalent when properties set in the comparison property list are used when comparing.

### Function Callbacks

*H5Ocompare* will invoke function callback when encountering differences from comparing:

* links
* object metadata
* attribute’s metadata
* data values in datasets
* data values in attributes

The definitions of the five function callbacks—*link, obj\_md, attr\_md, dset\_data, attr\_data*—are described in the following sections. *H5Ocompare* may invoke the corresponding callback repeatedly for each type of difference found. The return values from each function callback can be:

* A zero value causes the callback to continue reporting the remaining differences found.
* A non-zero value causes the callback to discontinue reporting the remaining differences found.

Each callback uses the following data structure to indicate the comparison result:

typedef enum H5\_cmp\_status\_t {

 H5\_STATUS\_DIFFERENT,

 H5\_STATUS\_EXIST\_ONLY\_O1,

 H5\_STATUS\_EXIST\_ONLY\_O2,

 H5\_STATUS\_UNCOMPARABLE

} H5\_cmp\_status\_t;

* H5\_STATUS\_DIFFERENT
	+ when the two values are different
* H5\_STATUS\_EXIST\_ONLY\_O1
	+ when the value exists only in the first object
* H5\_STATUS\_EXIST\_ONLY\_O2
	+ when the value exists only in the second object
* H5\_STATUS\_UNCOMPARABLE
	+ when the ranks of the dataspaces are not the same
	+ when the datatypes are not the same class
	+ when the datatypes are the same class but conversion of the data values are not possible
	+ when the ranks of array datatypes are not the same

### The function callback: link

*herr\_t (\*H5O\_cmp\_link\_cb\_t)(H5O\_cmp\_index\_t index, H5\_cmp\_status\_t status, H5O\_cmp\_link\_type\_t type, H5O\_cmp\_link\_values\_t \*values, void \*udata)*

The parameters have the following values or meanings:

*index*

* A union type, *H5O\_cmp\_index\_t* is defined below.
* Indicates which link is being compared:
	+ When compared according to name, *name* is valid and is the link name
	+ When compared according to creation order, *corder* is valid and is the link’s creation order

*status*

* An enumerated type, *H5\_cmp\_status\_t* is described previously.
* Reports the result of the comparison for *type*.

*type*

* An enumerated type, *H5O\_cmp\_link\_type\_t* is defined below.
* Reports the type of difference found.

*values*

* A union type, *H5O\_cmp\_link\_values\_t* is defined below.
* Reports the values of the difference found for *type*.
* A union of structures with each *struct* corresponding to each value defined for *type*. There are two fields of the same type in each structure. If *status* is *H5\_STATUS\_EXIST\_ONLY\_O1*, the value of the second field in the structure is undefined. Likewise, if *status* is *H5\_STATUS\_EXIST\_ONLY\_O2*, the value of the first field in the structure is undefined.

*udata*

* Equals to the *udata* field in the parameter *cb\_info* that is passed to *H5Ocompare*.
* Used to share any application-defined data between the application and the callbacks.

#### H5O\_cmp\_index\_t

typedef union H5O\_cmp\_index\_t {

 const char \*name;

int64\_t corder;

} H5O\_cmp\_index\_t;

#### H5O\_cmp\_link\_type\_t

|  |  |
| --- | --- |
| *H5O\_LINK\_EXIST* | Used to indicate that the link only exists in one group. The object it exists in is indicated by the *status* parameter and the *values* parameter will be set to NULL for the callback. This will be the only callback made for this link. |
| *H5O\_LINK\_CSET**H5O\_LINK\_CORDER**H5O\_LINK\_TYPE H5O\_LINK\_NAME**H5O\_LINK\_VALUE* | Character set encoding of link name (set with *H5Pset\_char\_encoding*) Creation order of link (set with *H5Pset\_link\_creation\_order*)Type of link (hard, soft, external or user-defined)Difference found for link names when compared according to creation order.Difference found for link values when comparing symbolic links (soft link or external link) |

#### H5O\_cmp\_link\_values\_t

typedef union H5O\_cmp\_link\_values\_t {

 struct {

 H5T\_cset\_t val1;

 H5T\_cset\_t val2;

 } cset;

 struct {

 int64\_t val1;

 int64\_t val2;

 } corder;

 struct {

 H5L\_type\_t val1;

 H5L\_type\_t val2;

 } link\_type;

 struct {

 const char \*val1;

 const char \*val2;

 } link\_name;

 struct {

 H5O\_cmp\_link\_val\_t1 val1;

 H5O\_cmp\_link\_val\_t val2;

 } link\_val;

} H5O\_cmp\_link\_values\_t;

1The structure *H5O\_cmp\_link\_val\_t* is defined below.

#### H5O\_cmp\_link\_val\_t

Typedef struct H5O\_cmp\_link\_val\_t {

 H5L\_type\_t ltype;

 union {

const char \*soft\_link;

 struct {

 const char \*filename;

 const char \*obj\_path;

 } ext\_link;

 } lval;

} H5O\_cmp\_link\_val\_t;

### The function callback: obj\_md

*herr\_t (\*H5O\_cmp\_obj\_md\_cb\_t)(H5\_cmp\_status\_t status, H5O\_cmp\_obj\_md\_type\_t type, H5O\_cmp\_obj\_md\_values\_t \*values, void \*udata)*

The parameters have the following values or meanings:

*status*

* An enumerated type, *H5\_cmp\_status\_t* is described previously.
* Reports the result of the comparison for *type*.

*type*

* An enumerated type, *H5O\_cmp\_obj\_md\_type\_t* is defined below.
* Reports the type of difference found.

*values*

* A union type, *H5O\_cmp\_obj\_md\_values\_t* is defined below.
* Reports the values of the difference found for *type*.
* A union of structures with each *struct* corresponding to a value defined for *type*. There are two fields of the same type in each structure. If *status* is *H5\_STATUS\_EXIST\_ONLY\_O1*, the value of the second field in the structure is undefined. Likewise, if *status* is *H5\_STATUS\_EXIST\_ONLY\_O2*, the value of the first field in the structure is undefined.

 *udata*

* Equals to the *udata* field in the parameter *cb\_info* that is passed to *H5Ocompare*.
* Used to share any application-defined data between the application and the callbacks.

#### H5O\_cmp\_obj\_md\_type\_t

|  |  |
| --- | --- |
| *H5O\_OBJ\_MD\_TYPE* | Type of object |
| *H5O\_OBJ\_MD\_RC*  | Reference count of object |
| *H5O\_OBJ\_MD\_ATIME* | Access time (set with *H5Pset\_obj\_track\_times*) |
| *H5O\_OBJ\_MD\_MTIME* | Modification time (set with *H5Pset\_obj\_track\_times*) |
| *H5O\_OBJ\_MD\_CTIME* | Change time (set with *H5Pset\_obj\_track\_times*) |
| *H5O\_OBJ\_MD\_BTIME* | Birth time (set with *H5Pset\_obj\_track\_times*) |
| *H5O\_OBJ\_MD\_NUM\_ATTRS* | Number of attributes attached to object |
| *H5O\_OBJ\_MD\_GRP\_CRT\_ORDER\_FLAGS**H5O\_OBJ\_MD\_GRP\_MAX\_COMPACT**H5O\_OBJ\_MD\_GRP\_MIN\_DENSE* | Creation order flags for a group (set with *H5Pset\_link\_creation\_order*). This comparison will be made only if both objects are groupsMaximum number of entries to store for a compact group (set with *H5Pset\_link\_phase\_change*). This comparison will be made only if both objects are groupsMinimum number of entries to store in a dense group (set with *H5Pset\_link\_phase\_change*). This comparison will be made only if both objects are groups |
| *H5O\_OBJ\_MD\_ATTR\_CRT\_ORDER\_FLAGS* | Creation order flags for attributes (set with *H5Pset\_attr\_creation\_order*) |
| *H5\_OBJ\_MD\_ATTR\_MAX\_COMPACT* | Maximum number of attributes to store in the object header (set with *H5Pset\_attr\_phase\_change*) |
| *H5O\_OBJ\_MD\_ATTR\_MIN\_DENSE* | Minimum number of attributes to store in dense storage (set with *H5Pset\_attr\_phase\_change*) |
| *H5O\_OBJ\_MD\_COMMENT* | Object comment (*H5Oset\_comment*) |
| *H5O\_OBJ\_MD\_DTYPE* | Datatype (for datasets and named datatypes). This comparison will not be made if one object is a group. |
| *H5O\_OBJ\_MD\_FILTER\_PIPELINE* | An object creation property list containing a copy of the object’s filter pipeline (for datasets and groups; set with *H5Pset\_filter*, etc.).  |
| *H5O\_OBJ\_MD\_SPACE* | Dataspace (for datasets). This comparison will not be made if one object is not a dataset. |
| *H5O\_OBJ\_MD\_LAYOUT* | Layout type (for datasets; set with *H5Pset\_layout*). This comparison will not be made if one object is not a dataset. |
| *H5O\_OBJ\_MD\_CHUNK* | Chunked layout information (for chunked datasets; set with *H5Pset\_chunk*). This comparison will not be made if one object is not a chunked dataset. |
| *H5O\_OBJ\_MD\_EXTERNAL**H5O\_OBJ\_MD\_FILL\_DTYPE*  | External layout information (for external datasets; set with *H5Pset\_external*). This comparison will not be made if one object is not an external dataset.Datatype for fill value (for datasets; set with *H5Pset\_fill\_value*). This datatype may differ from that of the dataset, but the HDF5 library must be able to convert value to the dataset datatype when the dataset is created. This comparison will not be made if one object is not a dataset. |
| *H5O\_OBJ\_MD\_FILL\_VALUE* *H5O\_OBJ\_MD\_FILL\_TIME* | Fill value (for datasets; set with *H5Pset\_fill\_value*). This comparison will not be made if one object is not a dataset.Fill time (for datasets; set with *H5Pset\_fill\_time*). This comparison will not be made if one object is not a dataset. |
| *H5O\_OBJ\_MD\_ALLOC\_TIME* | Allocation time (for datasets; set with *H5Pset\_alloc\_time*). This comparison will not be made if one object is not a dataset. |

#### H5O\_cmp\_obj\_md\_values\_t

typedef union H5O\_cmp\_obj\_md\_values\_t {

struct {

 H5O\_type\_t val1;

 H5O\_type\_t val2;

} type;

struct {

 unsigned val1;

 unsigned val2;

} rc;

struct {

 time\_t val1;

 time\_t val2;

} atime;

struct {

 time\_t val1;

 time\_t val2;

} mtime;

struct {

 time\_t val1;

 time\_t val2;

} ctime;

struct {

 time\_t val1;

 time\_t val2;

} btime;

struct {

 unsigned val1;

 unsigned val2;

} num\_attrs;

struct {

 unsigned val1;

 ussigned val2;

} grp\_crt\_order\_flags;

struct {

 unsigned val1;

 unsigned val2;

} grp\_max\_compact;

struct {

 unsigned val1;

 unsigned val2;

} grp\_min\_dense;

struct {

 unsigned val1;

 ussigned val2;

} attr\_crt\_order\_flags;

struct {

 unsigned val1;

 unsigned val2;

} attr\_max\_compact;

struct {

 unsigned val1;

 unsigned val2;

} attr\_min\_dense;

struct {

 const char \*val1;

 const char \*val2;

} comment;

struct {

 H5O\_cmp\_dtype\_t1 val1;

 H5O\_cmp\_dtype\_t val2;

} dtype;

struct {

 hid\_t val1;

 hid\_t val2;

} filter\_pipeline;

struct {

 H5O\_cmp\_space\_t2 val1;

 H5O\_cmp\_space\_t val2;

} space;

struct {

 H5D\_layout\_t val1;

 H5D\_layout\_t val2;

} layout;

struct {

 struct {

int val1;

 int val2;

 } ndims;

 struct {

 const hsize\_t \*val1;

 const hsize\_t \*val2;

 } dim;

} chunk3;

struct {

 struct {

 int val1;

 int val2;

 } count;

 struct {

 H5O\_cmp\_external\_t4 val1;

 H5O\_cmp\_external\_t val2;

 } external;

} external5;

struct {

 hid\_t val1;

 hid\_t val2;

} fill\_dtype;

struct {

 struct {

 hid\_t val1;

 hid\_t val2;

 } dtype;

 struct {

 const void \*val1;

 const void \*val2;

 } value;

} fill\_value6;

struct {

 H5D\_fill\_time\_t val1;

 H5D\_fill\_time\_t val2;

 } fill\_time;

 struct {

 H5D\_alloc\_time\_t val1;

 H5D\_alloc\_time\_t val2;

 } alloc\_time;

 } H5O\_cmp\_obj\_md\_values\_t;

*1*The structure *H5O\_cmp\_dtype\_t* is defined below.

2The structure *H5O\_cmp\_space\_t* is defined below.

3*chunk*: values stored in the field *chunk.ndims* determine the length of the arrays pointed to by *chunk.dim*.

 4The structure *H5O\_cmp\_external\_t* is defined below.

5*external*: the values stored in the field *external.count* determine the length of the arrays pointed to by *external.external*.

 6*fill\_value*: the field *fill\_value.dtype* contains the datatypes for the fill value, and therefore indicates how to interpret the data stored in *fill\_value.value*, as well as the size of the data. The fill values are compared according to the same rules as data value comparison.

#### H5O\_cmp\_dtype\_t

typedef struct H5O\_cmp\_dtype\_t {

H5T\_class\_t tclass;

union {

struct atomic {

 H5T\_order\_t order;

 size\_t prec;

 size\_t offset;

 H5T\_pad\_t lsb\_pad;

 H5T\_pad\_t msb\_pad;

} atomic;

struct cmpd {

hid\_t dtype;

unsigned nmembs;

} cmpd;

struct enumer {

 hid\_t base\_dtype;

unsigned nmembs;

} enumer;

struct vlen {

 hid\_t base\_dtype;

} vlen;

struct opaque {

 const char \*tag;

} opaque;

struct array {

 hid\_t base\_dtype;

size\_t nelem;

 unsigned ndims;

 size\_t dim[H5S\_MAX\_RANK];

} array;

} type;

} H5O\_cmp\_dtype\_t;

#### H5O\_cmp\_space\_t

typedef struct H5O\_cmp\_space\_t {

unsigned rank;

hsize\_t size[H5S\_MAX\_RANK];

hsize\_t max[H5S\_MAX\_RANK];

} H5O\_cmp\_space\_t;

#### H5O\_cmp\_external\_t

typedef struct H5O\_cmp\_external\_t {

const char \*name;

off\_t offset;

hsize\_t size;

} H5O\_cmp\_external\_t;

### The function callback: attr\_md

*herr\_t (\*H5O\_cmp\_attr\_md\_cb\_t)(H5O\_cmp\_index\_t index, H5\_cmp\_status\_t status, H5O\_cmp\_attr\_md\_type\_t type, H5O\_cmp\_attr\_md\_values\_t \*values, void \*udata)*

The parameters have the following values or meanings:

*index*

* A union type, *H5O\_cmp\_index\_t* is defined previously.
* Indicates which attribute is being compared:
	+ When compared according to name, *name* is valid and is the attribute name
	+ When compared according to creation order, *corder* is valid and is the attribute’s creation order

*status*

* An enumerated type, *H5\_cmp\_status\_t* is described previously.
* Reports the result of the comparison for *type*.

*type*

* An enumerated type, *H5O\_cmp\_attr\_md\_type\_t* is defined below.
* Reports the type of difference found.

*values*

* A union type, *H5O\_cmp\_attr\_md\_values\_t* is defined below.
* Reports the values of the difference found for *type*.
* A union of structures with each *struct* corresponding to each value defined for *type*. There are two fields of the same type in each structure. If *status* is *H5\_STATUS\_EXIST\_ONLY\_O1*, the value of the second field in the structure is undefined. Likewise, if *status* is *H5\_STATUS\_EXIST\_ONLY\_O2*, the value of the first field in the structure is undefined.

*udata*

* Equals to the *udata* field in the parameter *cb\_info* that is passed to *H5Ocompare*.
* Used to share any application-defined data between the application and the callbacks.

#### H5O\_cmp\_attr\_md\_type\_t

|  |  |
| --- | --- |
| *H5O\_ATTR\_EXIST**H5O\_ATTR\_CSET**H5O\_ATTR\_CORDER**H5O\_ATTR\_NAME**H5O\_ATTR\_DTYPE*  | Used to indicate that the attribute only exists on one object. The object it exists in is indicated by the *status* parameter and the *values* parameter will be set to NULL for the callback. This will be the only callback made for this attribute.Character set encoding of attribute name (set with *H5Pset\_char\_encoding*) Creation order of attributes (set with *H5Pset\_attr\_creation\_order*)The difference found in names of attributes when compared according to creation order.Datatype of attribute |
| *H5O\_ATTR\_SPACE* | Dataspace of attribute |

#### H5O\_cmp\_attr\_md\_values\_t

typedef union H5O\_cmp\_attr\_md\_values\_t {

struct {

 H5T\_cset\_t val1;

 H5T\_cset\_t val2;

 } cset;

 struct {

 H5O\_msg\_crt\_idx\_t val1;

 H5O\_msg\_crt\_idx\_t val2;

 } corder;

 struct {

 const char \*val1;

 const char \*val2;

 } name;

 struct {

H5O\_cmp\_dtype\_t1 val1;

H5O\_cmp\_dtype\_t val2;

 } dtype;

 struct {

H5O\_cmp\_space\_t2 val1;

 H5O\_cmp\_space\_t val2;

 } space;

} H5O\_cmp\_attr\_md\_values\_t;

*1*The structure *H5O\_cmp\_dtype\_t* is defined previously.

2The structure *H5O\_cmp\_space\_t* is defined previously.

### The function callback: dset\_data

*herr\_t (\*H5O\_cmp\_dset\_data\_cb\_t)(H5\_cmp\_status\_t status, H5O\_cmp\_data\_values\_t \*values, void \*udata)*

The parameters have the following values or meanings:

*status*

* An enumerated type, *H5\_cmp\_status\_t* is described previously.
* Reports the result of the comparison.

*values*

* A structure, *H5O\_cmp\_data\_values\_t* is defined below.
* Reports the differences found for the raw data values.

*udata*

* Equals to the *udata* field in the parameter *cb\_info* that is passed to *H5Ocompare*.
* Used to share any application-defined data between the application and the callbacks.

#### H5O\_cmp\_data\_values\_t

typedef struct H5O\_cmp\_data\_values\_t {

 unsigned ndiffs;

 unsigned rank;

 struct {

 hid\_t val1;

 hid\_t val2;

 } dtype;

 struct {

 const H5O\_cmp\_value\_diff\_t \*diff1;

 const H5O\_cmp­\_value\_diff\_t \*diff2;

 } diffs;

} H5O\_cmp\_data\_values\_t;

The four fields in *H5O\_cmp\_data\_values\_t* are described as follows:

*ndiffs*

* Contains the number of differences reported by this call which may be less than the total number of differences.
* Indicates the length of the arrays pointed to by *diffs.diff1, diffs.diff2*.

*rank*

* The number of dimension sizes for the dataspaces

*dtype*

* The datatype IDs of the two objects or attributes

*diffs*

* A structure of two fields with the same type *H5O\_cmp\_value\_diff\_t* as defined below.
* Points to two arrays of matched elements that are different between the two objects.
* For example, *diff1[0]* is matched against *diff2[0]*, and its inclusion in this list implies that the data stored in *diff1[0].value* is different from that in *diff2[0].value*. The *offset* field denotes the logical offset of the element within the dataset—*diff1[x].offset* will always be equal to *diff2[x].offset*. The length of *offset* is determined by the number of dimensions in the dataspace of the dataset or attribute.

#### H5O\_cmp\_value\_diff\_t

typedef struct H5O\_cmp\_value\_diff\_t {

hsize\_t \*offset;

void \*value;

} H5O\_cmp\_value\_diff\_t;

### The Function callback: attr\_data

*herr\_t (\*H5O\_cmp\_attr\_data\_cb\_t)(H5O\_cmp\_index\_t index, H5\_cmp\_status\_t status, H5O\_cmp\_data\_values\_t \*values, void \*udata)*

The parameters have the following values or meanings:

*index*

* A union type, *H5O\_cmp\_index\_t* is defined previously.
* Indicates which attribute’s data is being compared:
	+ When compared according to names, *name* is valid and is the attribute’s name.
	+ When compared according to creation order, *corder* is valid and is the attribute’s creation order.

*status*

* An enumerated type, *H5\_cmp\_status\_t* is described previously.
* Reports the result of the comparison.

*values*

* A structure, *H5O\_cmp\_data\_values\_t* is described previously.
* Reports the differences found for the raw data values.

*udata*

* Equals to the *udata* field in the parameter *cb\_info* that is passed to *H5Ocompare*.
* Used to share any application-defined data between the application and the callbacks.

## New public function for comparing file metadata

**Name:**

*H5Fcompare\_md*

**Signature:**

htri\_t *H5Fcompare\_md (hid\_t loc1\_id, hid\_t loc2\_id, hid\_t cmppl\_id, H5F\_cmp\_file\_md\_cb\_t \*file\_md, void \*udata)*

**Purpose:**

Compare file metadata of two files.

**Description:**

H5Fcompare\_md compares file metadata of the file specified by loc1\_id with the file metadata of the file specified by loc2\_id. File metadata is information the library uses to describe the HDF5 file and to identify its associated objects.

The parameter *cmppl\_id* is the comparison property list (and is currently unused).

Differences in the metadata are reported via the callback function, *file\_md*. This is passed in as a parameter to this routine and is described below. The parameter *udata* points to user data and is passed as a parameter to the callback function.

**Parameters:**

|  |  |
| --- | --- |
| *hid\_t* loc1\_id*hid\_t* loc2\_id*hid\_t cmppl\_id**H5F\_cmp\_file\_md\_cb\_t* \*file\_md*void \**udata | IN: Location identifier of the first file to be comparedIN: Location identifier of the second file to be comparedIN: The comparison property listIN/OUT: A callback functionIN/OUT: Pointer to user data |
|  |  |

**Returns:**

The return value indicates the result of the comparison:

* True (1) if all the metadata of the two files are equivalent (strict or loose)
* False (0) if one or more of the metadata are not equivalent (strict or loose)
* Negative value if there is error

### Callback function: *file\_md*

*herr\_t (\*H5F\_cmp\_file\_md\_cb\_t)(H5\_cmp\_status\_t status, H5F\_cmp\_file\_md\_type\_t type, H5F\_cmp\_file\_md\_values\_t \*values, void \*udata)*

The callback function is called repeatedly for each difference found while comparing the two file’s metadata. The return values from the callback are the same as described previously in *H5Ocompare*.

The parameters of this callback function have the following values or meanings:

*status*

* An enumerated type, *H5\_cmp\_status\_t* is described previously in *H5Ocompare*.
* Reports the result of the comparison for *type*.

*type*

* An enumerated type, *H5F\_cmp\_file\_md\_type\_t* is defined below.
* Reports the type of difference found.

*values*

* A union type, *H5F\_cmp\_file\_md\_values\_t* is defined below.
* Reports the values of the difference found for *type*.
* A union of structures with each *struct* corresponds to each value defined for *type*. There are two fields of the same datatype in each structure. If *status* is *H5\_STATUS\_EXIST\_ONLY\_O1*, the value of the second field in the structure is undefined. Likewise, if *status* is *H5\_STATUS\_EXIST\_ONLY\_O2*, the value of the first field in the structure is undefined.

*udata*

* Equals to the *udata* field in the parameter *cb\_info* that is passed to *H5Fcompare\_md*.
* Used to share any application-defined data between the application and the callbacks.

#### H5F\_cmp\_file\_md\_type\_t

|  |  |
| --- | --- |
| *H5F\_FILE\_MD\_USERBLOCK\_SIZE* | Size of the user block (set with *H5Pset\_userblock*) |
| *H5F\_FILE\_MD\_SIZEOF\_ADDR*  | Size of addresses stored in the file (*H5Pset\_sizes*) |
| *H5F\_FILE\_MD\_SIZEOF\_SIZE* | Size of lengths stored in the file (*H5Pset\_sizes*) |
| *H5F\_FILE\_MD\_SYM\_IK* | “K” value of group B-tree internal nodes (*H5Pset\_sym\_k*) |
| *H5F\_FILE\_MD\_SYM\_LK* | “K” value of group B-tree leaf nodes (set with *H5Pset\_sym\_k*) |
| *H5F\_FILE\_MD\_ISTORE\_K* | “K” value of data chunk B-trees (*H5Pset\_istore\_k*) |
| *H5F\_FILE\_MD\_SHARED\_MESG\_INDEXES* | Number and type of shared message indexes (set with *H5Pset\_shared\_mesg\_nindexes*, *H5Pset\_shared\_mesg\_index*) |
| *H5F\_FILE\_MD\_SHARED\_MESG\_MAX\_LIST* | Maximum number of shared messages to store in a list (set with *H5Pset\_shared\_mesg\_phase\_change*) |
| *H5F\_FILE\_MD\_SHARED\_MESG\_MIN\_BTREE* | Minimum number of shared messages to store in a B-tree (set with *H5Pset\_shared\_mesg\_phase\_change*) |

#### H5F\_cmp\_file\_md\_values\_t

typedef union H5F\_cmp\_file\_md\_values\_t {

 struct {

 haddr\_t val1;

 haddr\_t val2;

 } userblock\_size;

 struct {

 unsigned val1;

 unsigned val2;

 } sizeof\_addr;

 struct {

 unsigned val1;

 unsigned val2;

 } sizeof\_size;

 struct {

 unsigned val1;

 unsigned val2;

 } sym\_ik;

 struct {

 unsigned val1;

 unsigned val2;

 } sym\_lk;

 struct {

 unsigned val1;

 unsigned val2;

 } istore\_k;

 struct {

 struct {

 unsigned val1;

 unsigned val2;

 } nindexes;

 struct {

 H5F\_cmp\_shared\_mesg\_index\_t1 val1;

 H5F\_cmp\_shared\_mesg\_index\_t val2;

 } indexes;

 } shared\_mesg\_indexes2;

 struct {

 unsigned val1;

 unsigned val2;

 } shared\_mesg\_max\_list;

 struct {

 unsigned val1;

 unsigned val2;

 } shared\_mesg\_min\_btree;

} H5F\_cmp\_file\_md\_values\_t;

1The structure *H5F\_cmp\_shared\_mesg\_index\_t* is defined below.

2The values stored in the field *shared\_mesg\_indexes.nindexes* determine the length of the arrays pointed to by *shared\_mesg\_indexes.indexes*.

#### H5F\_cmp\_shared\_mesg\_index\_t

typedef struct H5F\_cmp\_shared\_mesg\_index\_t {

 unsigned mesg\_type\_flags;

 unsigned min\_mesg\_size;

} H5F\_cmp\_shared\_mesg\_index\_t;

## New public functions for handling comparison properties

There will be a new property list class (H5P\_OBJ\_COMPARE) for comparing objects. Two new public functions are available to set and get the properties in the comparison property list when comparing objects.

### H5Pset\_compare

**Name:**

*H5Pset\_compare*

**Signature:**

herr\_t  *H5Pset\_compare (hid\_t cmppl\_id, H5\_flags\_t compare\_options)*

**Purpose:**

Set properties to be used when comparing two objects.

**Description:**

H5Pset\_compare sets properties in the comparison property list *cmppl\_id* that will be invoked when comparing two objects.

cmppl\_id is the comparison property list and specifies the properties governing the comparison of two objects.

The parameter compare\_options is of type H5\_flags\_t, which is defined as:

 typedef uint32\_t H5\_flags\_t

The following flags are available for inclusion in the comparison property list and they can be set in compare\_options:

When comparing objects or links:

1. H5O\_COMPARE\_SKIP\_CRT\_PROP

Do not compare creation properties of groups, datasets, attributes or links

1. H5O\_COMPARE\_SKIP\_ATTRS

Do not compare attributes attached to objects (groups, datasets, named datatypes)

When comparing links in groups:

1. H5O\_COMPARE\_SKIP\_UNIQUE\_LINKS

Do not report unique links (links that exist only in one of the groups)

1. H5O\_COMPARE\_SKIP\_COMMON\_LINKS

Do not compare links with names or creation order common in both groups

1. H5O\_COMPARE\_LINKS\_CRT\_ORDER

Compare links according to creation order

1. H5O\_COMPARE\_SKIP\_LINK\_TYPE

Do not compare link type

1. H5O\_COMPARE\_SKIP\_LINK\_VALUE

Do not compare link value for symbolic links

When comparing attributes attached to objects (groups, datasets, named datatypes):

1. H5O\_COMPARE\_SKIP\_UNIQUE\_ATTRS

Do not report unique attributes (attributes that exist only in one of the objects)

1. H5O\_COMPARE\_SKIP\_COMMON\_ATTRS

Do not compare common attributes (attributes with names common in both objects)

1. H5O\_COMPARE\_ATTRS\_CRT\_ORDER

Compare attributes according to creation order

When comparing data values:

1. H5O\_COMPARE\_VALUES\_SAME\_DTYPES

Compare data values only if the datatypes are the same class and same types within the class

1. H5O\_COMPARE\_VALUES\_SAME\_DSPACES

Compare data values only if the datatypes and dataspaces are the same

1. H5O\_COMPARE\_SKIP\_NAN

Do not compare Not-a-Number

When comparing dataspaces:

1. H5O\_COMPARE\_SKIP\_DSPACE\_MAX\_DIM

Do not compare the maximum dimension sizes of dataspaces

**Parameters:**

|  |  |
| --- | --- |
| *hid\_t* loc1\_id*H5\_flags\_t* compare\_options | IN: Location identifier of the first file to be comparedIN: Flag(s) to be set for comparison |

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Pget\_compare

**Name:**

*H5Pget\_compare*

**Signature:**

herr\_t  *H5Pget\_compare (hid\_t cmppl\_id, H5\_flags\_t \*compare\_options)*

**Purpose:**

Retrieves properties to be used when comparing two objects.

**Description:**

H5Pget\_compare retrieves the properties currently specified in the comparison property list *cmppl\_id*, which will be invoked when comparing two objects.

compare\_options is a bit map indicating the flags or properties governing the comparison of two objects that are set in the comparison property list cmppl\_id.

**Parameters:**

|  |  |
| --- | --- |
| *hid\_t* loc1\_id *H5\_flags\_t*  \*compare\_options | IN: Location identifier of the first file to be comparedOUT: Flag(s) set in the comparison property list |

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

## New public functions for handling floating-point epsilon value

Two new public routines are available to set and get the epsilon value (precision limit) in the comparison property list when comparing floating-point data values. The default to use will be the epsilon values defined by the system, FLT\_EPSILON, DBL\_EPSILON, and LDBL\_EPSILON. If the system epsilon values are not defined, use constants that are close to most epsilon values as:

#define FLT\_EPSILON 1.19209E-07

#define DBL\_EPSILON 2.22045E-16

#define LDBL\_EPSILON 1.0842E-19

### H5Pset\_compare\_epsilon

**Name:**

*H5Pset\_compare\_epsilon*

**Signature:**

herr\_t  *H5Pset\_compare\_epsilon (hid\_t cmppl\_id, H5\_cmp\_epsilon\_t epsilon)*

**Purpose:**

Set the epsilon value to be used when comparing two objects’ floating-point data values.

**Description:**

H5Pset\_compare\_epsilon sets the precision limits, *epsilon*, in the comparison property list *cmppl\_id* that will be used when comparing floating-point data values.

**Parameters:**

|  |  |
| --- | --- |
| *hid\_t* loc1\_id *H5\_cmp\_epsilon\_t* epsilon | IN: Location identifier of the first file to be comparedIN: The epsilon value to be set*H5\_cmp\_epsilon\_t* is defined as:typedef union H5\_cmp\_epsilon\_t { float f\_epsilon; /\* float \*/ double d\_epsilon; /\* double \*/ long double l\_epsilon; /\* long double \*/} H5\_cmp\_epsilon\_t; |

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Pget\_compare\_epsilon

**Name:**

*H5Pget\_compare\_epsilon*

**Signature:**

herr\_t  *H5Pget\_compare\_epsilon (hid\_t cmppl\_id, H5\_cmp\_epsilon\_t \*epsilon)*

**Purpose:**

Retrieves the epsilon value used when comparing two objects’ floating-point data values.

**Description:**

H5Pget\_compare\_epsilon retrieves the epsilon values currently specified in the comparison property list *cmppl\_id* when comparing two objects’ floating-point data values.

**Parameters:**

|  |  |
| --- | --- |
| *hid\_t* loc1\_id *H5\_cmp\_epsilon\_t* \*epsilon | IN: Location identifier of the first file to be comparedOUT: The epsilon that is set in the comparison property list |

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

# Examples

This section presents a few examples to help you understand how the *H5Ocompare()* function can be used for different purposes.

**NOTE: These examples are out of date and will be revised to reflect API above – please ignore them for now!**

## Example 1: Check File Metadata

In this example, we will compare the differences in the file metadata for two files and print the results to the standard output stream, stdout.

### Step 1: implement the callback function for reporting results, *H5O\_cmp\_file\_md\_cb\_t*,

herr\_t compare\_file\_superblock\_cb (H5O\_cmp\_file\_md\_type\_t type, H5O\_cmp\_status\_t status, const H5O\_cmp\_file\_md\_info\_t \*cmp\_info, UNUSED void \*udata)

{

 if (status == H5O\_STATUS\_UNEQUAL && cmp\_info)

 {

 printf("\n");

 switch (type) {

 case H5O\_FILE\_MD\_SUPERBLOCK\_VERSION:

 printf("Superblock version (file1, file2): \t%d\t\%d\n",

 cmp\_info->superblock\_version.o1, cmp\_info->superblock\_version.o2);

 break;

 case H5O\_FILE\_MD\_SIZEOF\_ADDR:

 printf("Size of addresses(file1, file2):: \t%d\t%d\n",

 cmp\_info->sizeof\_addr.o1, cmp\_info->sizeof\_addr.o2);

 break;

 case H5O\_FILE\_MD\_SIZEOF\_SIZE:

 printf("Size of lengths(file1, file2): \t%d\t%d\n",

 cmp\_info->sizeof\_size.o1, cmp\_info->sizeof\_size.o2);

 break;

 case H5O\_FILE\_MD\_ISTORE\_K:

 printf("\"K\" value of data chunk b-trees in file one: \t%d\t%d\n",

 cmp\_info->istore\_k.o1, cmp\_info->istore\_ke.o2);

 break;

 case H5O\_FILE\_MD\_SYM\_IK:

 printf(""\"K\" value of group b-tree internal nodes (file1, file2): \t%d\t%d\n",

 cmp\_info->sym\_ik.o1, cmp\_info->sym\_ik.o2);

 break;

 case H5O\_FILE\_MD\_SYM\_LK:

 printf("\"K\" value of group b-tree leaf nodes (file1, file2): \t%d\t%d\n",

 cmp\_info->sym\_ik.o1, cmp\_info->sym\_ik.o2);

 break;

 case H5O\_FILE\_MD\_USERBLOCK\_SIZE:

 printf("Size of the user block(file1, file2): \t%d\t%d\n",

 cmp\_info->userblock\_size.o1, cmp\_info->userblock\_size.o2);

 break;

 case H5O\_FILE\_MD\_SHARED\_MESG\_INDEXES:

 printf("Number of shared messages(file1, file2): \t%d\t%d\n",

 cmp\_info->shared\_mesg\_indexes.nindexes.o1, cmp\_info->shared\_mesg\_indexes.nindexes.o2);

 printf("Shared message indexes in file1 (mesg type, mesg size): ");

 for (i=0; i<cmp\_info->shared\_mesg\_indexes.nindexes.o1; i++)

 prinft("(%d, %d), ", cmp\_info->shared\_mesg\_indexes.indexes[i].o1.mesg\_type\_flags,

 cmp\_info->shared\_mesg\_indexes.indexes[i].o1.min\_mesg\_size);

 printf("Shared message indexes in file2 (mesg type, mesg size): ");

 for (i=0; i<cmp\_info->shared\_mesg\_indexes.nindexes.o2; i++)

 prinft("(%d, %d), ", cmp\_info->shared\_mesg\_indexes.indexes[i].o2.mesg\_type\_flags,

 cmp\_info->shared\_mesg\_indexes.indexes[i].o2.min\_mesg\_size);

 break;

 case H5O\_FILE\_MD\_SHARED\_MESG\_MAX\_LIST:

 printf("Maximum number of shared messages to store in a list(file1, file2): \t%d\t%d\n",

 cmp\_info->shared\_mesg\_max\_list.o1, cmp\_info->shared\_mesg\_max\_list.o2);

 break;

 case H5O\_FILE\_MD\_SHARED\_MESG\_MIN\_BTREE:

 printf("Minimum number of shared messages to store in a b-tree(file1, file2): \t%d\t%d\n",

 cmp\_info->shared\_mesg\_min\_btree.o1, cmp\_info->shared\_mesg\_min\_btree.o2);

 break;

 }

 }

 return SUCCEED;

}

### Step2: call H5Compare() to check the file metadata

#include "hdf5.h"

/\* usage: PROG\_NAME src\_file src\_obj dst\_file \*/

int main(int argc, char \*argv[]) {

 hid\_t file1=-1, file2=-1;

 H5O\_cmp\_cb\_t cb\_info;

 if (argc < 3) {

 puts("USAGE: PROG\_NAME file1 file2\n");

 return 1;

 }

 file1 = H5Fopen(argv[1], H5F\_ACC\_RDONLY, H5P\_DEFAULT);

 file2 = H5Fopen(argv[2], H5F\_ACC\_RDONLY, H5P\_DEFAULT);;

 memset(&cb\_info, 0, sizeof(H5O\_cmp\_cb\_t);

 file\_md.file\_md = compare\_file\_superblock\_cb;

 H5Ocompare(file1, ".", file2, ".", H5P\_DEFAULT, &cb\_info)

 H5Fclose(file1);

 H5Fclose(file2);

 return 0;

}

## Example 2: Check Data Values of Two Datasets

This example shows how to compare the values of two datasets. Instead of printing the results, the callback function returns the differences of the dataset values to the caller.

### Step 1: implement the callback function

typedef struct compare\_dset\_cb\_info\_t {

 size\_t buf\_size[2]; /\* buf\_size[0]=original buf size, buf\_size[1]=actual buf size \*/

 hsize\_t \*offset\_dset1; /\* OUT \*/

 void \*value\_dset1; /\* OUT \*/

 hsize\_t \*offset\_dset2; /\* OUT \*/

 void \*value\_dset2; /\* OUT \*/

} compare\_dset\_cb\_info\_t;

herr\_t compare\_dset\_data\_cb (char \*name, H5O\_cmp\_status\_t status,

 const H5O\_cmp\_dset\_data\_info\_t \*cmp\_info, void \*udata)

{

 compare\_dset\_cb\_info\_t \*info = (compare\_dset\_cb\_info\_t \*)udata;

 if (status == H5O\_STATUS\_UNEQUAL && cmp\_info)

 {

 info->buf\_size[1] = cmp\_info->ndiffs;

 if (info->buf\_size[0]<cmp\_info->ndiffs) {

 offset\_dset1 = (hsize\_t\*) realloc (offset\_dset1, cmp\_info->ndiffs \* sizeof(hsize\_t));

 offset\_dset2 = (hsize\_t\*) realloc (offset\_dset2, cmp\_info->ndiffs \* sizeof(hsize\_t));

 value\_dset1 = realloc (value\_dset1, cmp\_info->ndiffs \* H5Tget\_size(cmp\_info->dtype.o1));

 value\_dset2 = realloc (value\_dset2, cmp\_info->ndiffs \* H5Tget\_size(cmp\_info->dtype.o2));

 }

 memcpy(offset\_dset1, info->diff->o1.offset, cmp\_info->ndiffs \* sizeof(hsize\_t));

 memcpy(offset\_dset2, info->diff->o2.offset, cmp\_info->ndiffs \* sizeof(hsize\_t));

 memcpy(value\_dset1, info->diff->o1.value, cmp\_info->ndiffs \* H5Tget\_size(cmp\_info->dtype.o1));

 memcpy(value\_dset2, info->diff->o2.value, cmp\_info->ndiffs \* H5Tget\_size(cmp\_info->dtype.o2));

 }

 return SUCCEED;

};

### Step2: call the H5Ocompare() to get the differences of the values

#include "hdf5.h"

/\* usage: PROG\_NAME src\_file dst\_file src\_obj dst\_obj \*/

int main(int argc, char \*argv[]) {

 hid\_t file1=-1, file2=-1, did=-1;

 H5O\_cmp\_cb\_t cb\_info;

 compare\_dset\_cb\_info\_t udata;

 if (argc < 5) {

 puts("USAGE: PROG\_NAME file1 file2 dset1 dset2\n");

 return -1;

 }

 file1 = H5Fopen(argv[1], H5F\_ACC\_RDONLY, H5P\_DEFAULT);

 file2 = H5Fopen(argv[2], H5F\_ACC\_RDONLY, H5P\_DEFAULT);

 /\* set up the user data that is passed to the callback \*/

 udata.buf\_size[0] = 10;

 offset\_dset1 = (hsize\_t \*)calloc(udata.buf\_size[0], sizeof(hsize\_t));

 offset\_dset2 = (hsize\_t \*)calloc(udata.buf\_size[0], sizeof(hsize\_t));

 did = H5Dopen2(file1, argv[3], H5P\_DEFAULT);

 tid = H5Dget\_type(did);

 value\_dset1 = calloc(udata.buf\_size[0], H5Tget\_size(tid)));

 H5Tclose(tid);

 H5Dclose(did);

 did = H5Dopen2(file1, argv[4], H5P\_DEFAULT);

 tid = H5Dget\_type(did);

 value\_dset2 = calloc(udata.buf\_size[0], H5Tget\_size(tid)));

 H5Tclose(tid);

 H5Dclose(did);

 memset(&cb\_info, 0, sizeof(H5O\_cmp\_cb\_t);

 cb\_info.dset\_data = compare\_dset\_data\_cb;

 cb\_info.udata = &udata;

 /\* call H5Ocompare() to check the data values \*/

 H5Ocompare(fil1, argv[3], file2, argv[4], H5P\_DEFAULT, &cb\_info)

 H5Fclose(file1);

 H5Fclose(file2);

 if (udata.buf\_size[1] == 0)

 puts("No difference in data values.");

 else {

 /\* do something with the results \*/

 }

….

 return 0;

}

Revision History

|  |  |
| --- | --- |
| *January 12, 2011:* | Version 1 circulated for comment within The HDF Group.  |
| *January 20, 2011:* | Version 2 revised with Quincey’s and Neil’s feedback.  |
| *February 4, 2011:* | Version 3 added more details on how to compare objects.  |
| *March 16, 2011:* | Version 4 added details for H5Ocompare() function and examples.  |
| *January 18, 2012:* | Version 5 completely revised, removing recursive operation and revamping interface.  |
| Future Extensions |  |

* Allow user to specify the maximum number of differences reported per dataset when comparing its data values. By default, all the differences will be reported.
* Allow user to specify the maximum number of differences reported per callback.
* Options may be added in the future to strengthen compatibility requirements for datatypes (for example to require all fields in a compound be present in both datatypes) or relax compatibility requirements for dataspaces (for example to allow comparison as long as the total number of elements is the same).
* Public routines H5Tcompare and H5Scompare for users to compare datatypes and dataspaces.
1. <https://www.hdfgroup.uiuc.edu/RFC/HDF5/tools/h5diff/h5diff_spec.pdf> [↑](#footnote-ref-1)
2. https://www.hdfgroup.uiuc.edu/RFC/HDF5/tools/h5diff/RFC\_h5diff\_default\_epsilon.pdf [↑](#footnote-ref-2)
3. See the definition and description of equivalence (strict or loose) in the document

“*HDF5 File and Object Comparison Specification*” [*https://www.hdfgroup.uiuc.edu/RFC/HDF5/tools/h5diff/h5diff\_spec.pdf*](https://www.hdfgroup.uiuc.edu/RFC/HDF5/tools/h5diff/h5diff_spec.pdf) [↑](#footnote-ref-3)