RFC: Special Values in HDF5

Scott Wegner
Kent Yang
Quincey Koziol

Introduction

HDF5 supports a single default or user-defined fill value during dataset creation. However, there are cases where a user may wish to specify more than one “special” value to describe non-standard data. This RFC will discuss the potential use-cases, implementations, and issues with multiple special values.

1 Motivation

During the 2008 NASA HDF annual briefing, The HDF Group received a user request to “support more than one type of special value inside HDF5.” In response, another user mentioned, “HDF5 already supports this. This may be a mask array.” We've contacted these users to gather more details about their specific needs prior to writing this RFC.

1.1 Current Fill Value Support

HDF5 currently supports a single fill value. A fill value is the value retrieved for a dataset element in HDF5 when no application data has been written to that element. In such cases, the fill value is returned for the user to interpret appropriately.

When a dataset is created, a user can define important details about how the fill value behaves. These details include:

- The actual value used (a user-defined value, or the default)
- When space is allocated for the dataset
- When the fill value is actually written

The default behavior for these options differs depending on other dataset options, such as whether chunked storage is used.

1.2 Going Further-- Special Values

The current fill value implementation isn't sufficient for the proposed special values feature. A fill value conveys to the user that the particular data point is missing, for one reason or another. The goal is to provide a system for storing more information about particular data points or regions, such as why the data is missing. Ideally, special values could even be applied to points where there is also valid data. **Special values essentially provide meta-data within a dataset, on the scale of individual data points or regions.**

1.3 Use Case for Special Values

In a NASA HDF-EOS5 dataset describing atmospheric variables observed by a NASA satellite, there may be regions left unsampled because they contain water, clouds, or ice. We cannot perform reliable calculations using any of these regions, so a fill value should be used. However, it is important to record specifically whether a region was blocked by clouds, is filled with water, or is covered in ice. We can define a special value for each of these regions and write it to dataset.

1.3.1 Non-Mutual Exclusion

There may also be cases when the fill regions overlap. Using the same above example, we may have a region which is both filled with water and blocked by clouds. These combinations should also be represented.

This is the only use case we have received from users so far, although there are probably others.

2 Relation to Fill Values

Although HDF5's current fill value feature is similar in some ways, it does not satisfy users' needs, such as the case provided above. The fill values feature provides a means for the user to skip over missing data points and let the library handle the points correctly. With special values, data needs to be written at the particular points, so they must be treated specifically. Moreover, “special values” won't be relevant for a large number of users, so it is best not to complicate the existing fill value interface. The effects of the two features are related, but the motivations, use-cases, and designs are completely different. For these reasons, it is best that the two features be completely decoupled.

3 Level of Support

We've heard from users that there is a need for special values. Some have already implemented a system of their own, while others prefer that the functionality be added to the HDF5 library. We should consider how we want to integrate this feature, and at what level.

3.1 Core Library Support

One option is to create API functions in the core HDF5 library. This would include adding special values definitions to the dataset creation property list. We could also include functions to read and write special values, and even do partial I/O on “real” data, based on regions containing a special value.

3.2 High-Level Library Support

We could also think about adding high level API functions to support special values. This approach would allow us to define a standard implementation without actually changing the file format. For example, the special values data could be attached to a dataset as an attribute. High-level API functions could read and write special values, either in the actual dataset, and attribute, or in a parallel dataset (see Section 4.3.3).

3.3 Special Value Specification without API Support

As an alternative, we could simply leave the library as it is, and document a standard implementation for users. We could write a short recommendation describing the implementation, similar to what we might include through high-level API functions. We could also write a short example application demonstrating its use. This way, we don't need to officially support multiple special values in the library or file format, but we can recommend a standard practice to help users who share files which use special values.

4 Design

4.1 Goals

In considering the design and implementation of special values, we should keep the following goals in mind:

4.1.1 Backwards Compatibility

Most importantly, whatever approach is chosen, the file format and API must be able to handle files created without support for multiple special values. Ideally, if the meta-data for special values isn't found in a dataset, all other operations should still work as expected.

4.1.2 Storage and Performance Considerations

Using special values should not significantly degrade I/O performance on a dataset, nor should it significantly increase its storage cost. Where possible, compression should efficiently handle large regions of special values. Also, it is important that we don't write fill values as well as special values in situations when we don't need to.

4.1.3 Usability

As with any feature introduced in the HDF5 library, special values should be relatively straight-forward and easy to use. Also, the API should be robust enough to sufficiently address users' needs. As an example, it may be beneficial to define new special values even after creating a dataset. Also, partial I/O for special values could take advantage of existing hyperslab selection techniques already available to datasets. It may be necessary to gather more input from users regarding the particular features that are most important.

4.2 Special Value Datatype

In particular use-cases, a datatype that is different from the original datatype of the dataset would be more appropriate for the special values. Depending on how special values are implemented, though, we may be constrained to the original datatype of the dataset. In most cases, this should be adaptable.

4.2.1 Mutually Exclusive Special Values

If special values are mutually exclusive--that is, any data point can contain at most one special value--then the special values can simply be enumerated. In an existing datatype, values should be chosen such that they are distinct from any real data. For example, in a dataset containing positive data, we could use -1, -2, -3...

4.2.2 Non-Mutually Exclusive Special Values

If the values *aren't* mutually exclusive, or some special values may overlap, then we cannot use a simple enumeration. Rather, it would be best to use a bit-mask, such that special values could easily be OR'ed together. In an existing dataset, it might be appropriate to reserve the high-order bits of the datatype for special values, such as 0x100.., 0x010.., 0x001... Note that this has a high storage requirement for each additional special value, and is not as scalable as mutually exclusive values.

4.2.3 Scalability

There are inherent caps on the number of available special values that can be used in a dataset. For overlapping (non-mutually exclusive) special values in particular, each additional fill value will require an extra bit. This is more of an issue with the feature itself than with any implementation we choose. We should consider whether to define a strict bound on the number of special values, or alternatively how to handle cases where the number of values exceeds the size of the provided datatype.

4.3 Design Ideas

There are a number of different ways to handle special values in HDF5. We should consider the pros and cons of each, and evaluate how well they meet the above goals.

4.3.1 Dataset Header

We could consider adding special value information to the dataset header. An array containing each special value--with a string description and a value from the dataset's datatype--could be added to each dataset. This information would be set through the dataset creation property list, and requires changes to the HDF5 file format. The special values could then be written to data points or regions in the dataset, and interpreted selectively by the user.

This implementation provides the tightest integration with the library and could be supported with low-level API functions. However, it would require a change in the file format and extra API functions added to the core HDF5 library. This probably isn't the best choice because many users won't take advantage of this feature at all.

If the special values are added inside the dataset header and explicitly added to the HDF5 file format, then it may also be worthwhile to consider merging fill values and special values. In this regard, a fill value could be considered the simple case of one special value. As mentioned above, though, this view and implementation is overly-complicated for the general user.

4.3.2 Dataset Attribute

Rather than defining the special values in the dataset header, we could provide an attribute to the dataset. This attribute would include a detailed definition for each special value, similar to the implementation above. The special values would be written inside the dataset as above.

This option provides an advantage over the previous implementation because it doesn't require any changes to the file format. Also, there is no overhead for users who choose not to use special values. However, there are still many downsides. It is impossible to store both data values and special values for a data point. Also, we are constrained to defining our special values in the native datatype of the dataset. This can present unusual problems, such as trying to represent bit-mask special values in a float or compound datatype.

4.3.3 Parallel Special Values Dataset

Another idea, brought to our attention by a user, would be to write the special values data outside the original dataset by defining a new “parallel” dataset which contains all special value information. This new dataset would have the same rank and dimension as the original dataset, but only contain data for special values. The special values could be defined in the parallel dataset using either of the implementations above. An attribute could be added to the original dataset with an object pointer that “links” to the special values dataset.

This design has a few advantages. First of all, the parallel dataset is no longer constrained to using the datatype of the original dataset. Thus, we don't need to worry about reserving a potentially large number of values as “special” in the original data. Also, we have the opportunity to use a more space-efficient datatype, such as an enumeration or a bit-mask.

Another advantage is that many datasets can refer to the same parallel dataset. For example, if we have datasets that describe the temperature, pressure, and elevation over the same area, then we may define special values (such as “water-filled” and “ice-covered”) that are relevant and shared by each dataset.

This design requires additional physical space equal to the size of the original dataset. In most situations, this overhead would be greatly reduced by compression inside HDF5.

4.3.4 Attribute Triplet

Similar to the previous specification, “Dataset Attribute” (4.3.2), we can use an attribute, but use it to store all of special values information. This includes not only the definition for each special value but also the region information as well. Each special value definition would be a triplet of a string (for a name or description), a value (using any defined datatype), and a dataset region reference that contains the selection of elements in the dataset that the value applies to. The dataset would contain an array of these appropriately-defined compound datatype to store the triplet.

This particular implementation allows for non-mutual exclusion for special values. A user could even overlay special value regions where real data samples exist. This implementation would also benefit greatly if the HDF5 library adds support for simple operations on dataset region selections (i.e. union, intersect, complement).

5 Current Preference

In coming up with the list of possible designs and implementations, we have developed an opinion as to which will best satisfy the goals described above. Note that this preference comes prior to any outside comments, and is still under evaluation. We are open to consider other opinions.

It seems most suitable to store the special value definition in a dataset attribute along with the region information, as described in the “Attribute Triplet” (4.3.4) specification above. This provides the most flexibility for the datatype and values for each special value. It also has advantages over the “Parallel Special Values Dataset” (4.3.3) in that it tightly couples the special values data with the dataset it describes. Furthermore, a user could query the attribute and easily select all the data points for a particular special value. Existing datasets stored with HDF5 can easily be extended to include special values.

We also feel that it's best to add the feature simply as a specification, and to create an example application to demonstrate its use. This approach will allow us to further gage user interest and potentially move the code into the library at a later date. If many users benefit from the specification, then it will be relatively easy to "port" our example code to high-level API functions. Conversely, if not many users take advantage of special values, then it is better not to add extra bloat to the high-level or core HDF5 libraries.

Acknowledgments

We would like to thank Dan Kahn at SSAI (Science Systems and Applications, Inc) for sharing his application's special value implementation, as well as the example use-case included in this RFC.

Revision History

July 23, 2008:	Version 1 circulated for comment within The HDF Group.
August 2008:	Versions 2 and 3 incorporated comments from The HDF Group members.
August 21, 2008:	Version 4 incorporates final comments from The HDF Group members; posted for public comment. Comments should be sent to help@hdfgroup.org.
September 3, 2008:	Version 5 changes Section 4.4 "Current Design Preference" to Section 5 "Current Preference".