

170-WP-002-001

Thoughts on HDF-EOS Metadata

White Paper

White Paper—Not intended for
formal review or government approval.

December 1995

Prepared Under Contract NAS5-60000

RESPONSIBLE ENGINEER

<u>B. Fortner \s\</u>	<u>12/20/95</u>
Brand Fortner	Date
Doug Ilg	
EOSDIS Core System Project	

SUBMITTED BY

<u>Larry Klein \s\</u>	<u>12/20/95</u>
Larry Klein	Date
EOSDIS Core System Project	

Hughes Information Technology Corporation
Upper Marlboro, Maryland

This page intentionally left blank.

Abstract

This document is a discussion paper on the use of parameter=value metadata inside HDF-EOS datafiles. Here HDF refers to the scientific data format standard selected by NASA as the baseline standard for EOS, and HDF-EOS refers to EOS conventions for using HDF. In this document we discuss metadata in general, and the various ways that metadata should be stored in the HDF-EOS files. We also discuss how we can use metadata to describe the internal structure of the HDF-EOS file.

Note

This document is being made available even in preliminary form because of the high level of interest in HDF-EOS efforts, and to give people the chance to comment on and to change our direction of HDF-EOS, long before decisions are burned into silicon, so to speak.

Credits

This document was created with material from Doug Ilg (Hughes STX), Ted Meyer (National Aeronautics and Space Administration (NASA)), and Larry Klein, Brand Fortner, David Wynne (Applied Research Corporation). Comments and suggestions should be sent to:

Brand Fortner
Applied Research Corporation
1616A McCormick Dr.
Landover, MD 20785
USA

Email: bfortner@eos.hitc.com
Phone: (301) 925-0779
Fax: (301) 925-0321

Keywords: HDF-EOS, Metadata, Data Formats, PVL, Standard Data Products, Disk Formats, Browse, Arrays

This page intentionally left blank.

Contents

Abstract

1. Introduction

1.1	Purpose.....	1-1
1.2	Organization.....	1-1
1.3	Review and Approval.....	1-1

2. White Paper Format

2.1	A Need for Structural Metadata	2-1
2.2	Should Structural Metadata be Stored as Attributes or as Text?	2-1
2.3	What is PVL?	2-2
2.4	How Should the Structural Metadata Text be Stored?.....	2-2
2.5	Where Should Structural Metadata Text Attributes be Stored?.....	2-3
	2.5.1 What Information Should the Structural Metadata Contain?.....	2-4
2.6	HDF-EOS Configuration Files.....	2-9

Tables

2-1.	Metadata Definitions.....	2-1
2-2.	Structural Metadata Text.....	2-3

Abbreviations and Acronyms

This page intentionally left blank.

1. Introduction

1.1 Purpose

There is much discussion in EOS about metadata, or ‘information about data’. In this discussion, we will limit the term ‘metadata’ to refer to primarily scalar information, such as ‘observer=david’, that describes data. We will ignore the philosophical issue about one persons data being another persons metadata...

1.2 Review and Approval

This White Paper is an informal document approved at the Office Manager level. It does not require formal Government review or approval; however, it is submitted with the intent that review and comments will be forthcoming.

Questions regarding technical information contained within this Paper should be addressed to:

Brand Fortner, (email address: bfortner@eos.hitc.com).

Questions concerning the distribution should be addressed to:

Data Management Office
The ECS Project Office
Hughes Information Technology Corporation
1616 McCormick Drive
Upper Marlboro, Maryland 20774-5372

This page intentionally left blank.

2. White Paper Format

2.1 A Need for Structural Metadata

In the EOS system there is ‘core’ metadata, that must exist for every data product where applicable, and ‘product specific’ metadata, that may exist for only one particular product, or perhaps even individual granules. In either case, the EOS defined metadata deals with products and granules only. It typically does not deal with sub-granule information, such as what components are contained within a granule.

We therefore have a need for a third category of metadata that describes the contents of each granule, which for the most part will consist of single HDF-EOS files. We need this third component, which we will call ‘structural’ metadata, to describe each component contained in the granule in considerable detail, and in addition describe the relationships between the various components.

Table 2-1. Metadata Definitions

Type of Metadata	Description
Core Metadata	Info about data product. Fields are common across data products.
Product-Specific Metadata	Info about data product. Fields are specific to that data product.
Structural Metadata	Info about the structure and contents of a particular granule.

With this third metadata type (which we assume would be contained within the granule files), we or anyone else can use the contained information to provide general services such as listing the components, subsetting and subsampling the components by parameter and geolocation, and so on.

2.2 Should Structural Metadata be Stored as Attributes or as Text?

How should we provide this information? Some of the needed structural metadata is already inherent in the makeup of the HDF file itself. For example, standard HDF calls make it easy to list the contained components by name, or to do subsetting and subsampling by parameter or row/column. But other information, such as geolocation relationships, is just not normally there.

One way to provide this information is through the HDF feature of Attributes. For example, an array may have an associated HDF Attribute that describes its relationship to another geolocation array. We have a couple problems with this approach. The first problem is that only SDS arrays and the file itself can have associated Attributes. HDF data objects such as Vdatas and Vgroups cannot have Attributes. The second problem is that the use of attributes could greatly increase the number of data elements in the HDF directory, something that we are trying to minimize.

Our proposed solution is to store structural metadata as a text block that contains PVL metadata statements. This is an attractive option for a several reasons. First, PVL will be used for EOS core and product-specific metadata, so all three kinds of EOS metadata will now be in the same format. The second reason is that it potentially greatly decreases the number of data element entries needed in the HDF file directory. Yet another reason is that it makes it possible for *humans* to directly read and verify the structural metadata. This last feature will be especially important for data producers that will use configuration files, described later in this document.

One problem with using PVL text blocks is the need to parse the text to find the values of particular parameters. This problem will be mitigated somewhat by the need to develop efficient parsers for the core and product-specific metadata: we will just leverage off of their efforts. We expect that parsing, although relatively slow when compared to accessing information via attributes, will turn out to be a minor part of the computational burden of reading data from HDF-EOS files (although it increases the complexity of our HDF-EOS library).

2.3 What is PVL?

PVL stands for Parameter Value Language. It is a text standard for assigning names to values that was developed by the Consultative Committee for Space Data Systems (CCSDS). It is directly derived from ODL, used in the Planetary Data System (PDS) at JPL. For our purposes, it is merely a syntax for naming values, and for grouping those names. For example, consider the following brief example:

PVL Code Example: `group = DataParameter;`

```
    object = "Temperature";  
        DataType = "float32";  
    end_object = "Temperature";  
end_group = Dataparameter;
```

Here an object of type `DataParameter` contains the following attributes: an object name (`Temperature`), and a `DataType` (`float32`). PVL defines the syntax, such as ending every line with a semicolon. This example, although short, illustrates the two major features of PVL that we use: the naming of values (alphanumeric, numeric, etc.) and the grouping of those values. That's it. For more information, see the PVL Tutorial and PVL Language Specification, documents CCSDS641.0-G-1 and CCSDS641.0-B-1 respectively. Note that quotes around the values are optional. They are only required if you want to include spaces or special characters in the value string.

2.4 How Should the Structural Metadata Text be Stored?

Now we have decided to store structural metadata as a PVL text block. Where, exactly, should we store this text block? We cannot use HDF Annotations, as that feature will be phased out. We can instead store the text block as a single HDF Attribute, with a number type of character string (`DFNT_CHAR8`). The size of this attribute would be the size, in characters, of the text block.

There is a problem with character attributes, however: they are limited to 32K bytes for a single attribute. We would rarely encounter this limit, but when we do, we need a plan. Our current plan is to store the additional metadata in additional attributes. These attributes will be linked together by a naming convention.

For example, the first metadata attribute for core metadata will always be named CoreMetadata.0. If this attribute is not big enough, a second attribute will be created, of name CoreMetadata.1, and so on up to CoreMetadata.9. Similarly, product specific metadata will be stored as a text attribute of name ProductMetadata.0 through ProductMetadata.9, and structural Metadata will be named StructMetadata.0 thru StructMetadata.9.

Table 2-2. Structural Metadata Text

Type of Metadata	Name
Core	CoreMetadata.0
Product Specific	ProductMetadata.0
Structural	StructMetadata.0

2.5 Where Should Structural Metadata Text Attributes be Stored?

HDF Attributes can be attached to one of only two things: to the file itself (global attributes), or to an SDS array. This is not usually a problem for core and product specific metadata, since typically this information relates to the entire file as opposed to a component of the file.

This may however be a problem for structural metadata. The natural place for this information is inside the structure that it describes. For example, suppose an HDF file (granule) consists of 4 swath structures. You would naturally expect the structural metadata that describes each swath to be stored inside the swath itself.

That sounds good, but there is a problem. Not all swaths have SDSs. Therefore, there is no way to place an attribute in structures that do not contain SDSs. We therefore propose to keep all structural metadata at the global level, in a single text block (possibly spread over multiple character attributes, as described above).

For example, even if an HDF-EOS file contains 3 objects such as swaths, the structural metadata for all 3 swaths would be stored in a single global text attribute. One advantage of this scheme is the there is an almost exact, one-to-one mapping of this single structural metadata text block, and the PVL configuration file that may have been used to create the HDF-EOS file in the first place. These configuration files will be described in a later section.

A disadvantage to this scheme is that the metadata that relates to a particular object, such as a swath, is not stored as part of the swath. We can hide this detail through our HDF-EOS library in the following way: when you request metadata for a particular swath within the HDF-EOS file, our subroutines will scan the global metadata text block, and pull out only the metadata that relates to the displayed object.

2.5.1 What Information Should the Structural Metadata Contain?

Both the Core and Product Specific Metadata contain information that relates either to the Standard Data Product, or to the Granule itself (usually, a single HDF file). That leaves the structural metadata to describe information about particular components within the granule, what is sometimes called sub-granule information. We can think of three kinds of information that the structural metadata should contain:

- *Dimension Information*—Material that describes the dimensions used to size arrays and tables.
- *Component Information*—Material that relates to a specific component. An example would be say the name, units, scaling factors, etc. of a particular data array.
- *Geolocation Information*—Material that describes the relationship of data arrays to geolocation tables or arrays, used for subsetting or subsampling by geolocation.

We will discuss each of these in turn.

2.5.1.1 Dimension Information

Every array and table in the HDF file must have dimensions associated with it. For our purposes, these dimensions are not typically data objects in the file, but are just shorthand ways of describing the dimensions¹. Usually, a single dimension specification will be used many times for many different tables and arrays. An example of using PVL to specify a dimension is shown below (this example is for illustration only, as the details may change before it is released):

Dimension Example: group = Dimension;

```
object = "Track";  
    size = 600;  
end_object = "Track";
```

```
object = "Xtrack";  
    size = 25;  
end_object = "Xtrack";
```

```
object = "Dim3";  
    size = 100;  
end_object = "Dim3";
```

¹Actually, the HDF routines do create dimension data objects, but they serve a different purpose.

```
end_group = Dimension;
```

In this example, a dimension called `Track` is declared as size 600, another dimension called `Xtrack` as size 25, another called `Dim3` as size 100. Other items such as units, etc. can also be specified inside the dimension specification block. The `group=` and `end_group` lines are used to group all of the dimension definitions.

Note how we have two levels of grouping: one of the parameters inside `objects`, and another of the objects inside the `group`. In the PVL world, the keyword `object` and the keyword `group` are synonyms: as a convention we use `group` to denote larger collections, to improve readability.

2.5.1.2 Component Information

The `DataParameter` group shown below is used to list information about the data components of an HDF-EOS structure. These components can be data arrays or tables. Information such as the number type and the associated dimensions are listed here, as shown in the example below.

Component Example: `group = DataParameter;`

```
object = "Band_3";
    DataType = int16;
    DimList = ("Track", "Xtrack");
end_object = "Band_3";

object = "Temperature";
    DataType = float32;
    DimList = ("Track", "Dim3");
end_object = "Temperature";
```

```
end_group = DataParameter;
```

The example above shows the specification of two 2D arrays. The first array, `Band_3`, is of number type `int16`, and is of the size specified by the two dimensions: `Track` and `Xtrack` (we know it is 2D, because it only has two dimensions associated with it). The second array, `Temperature`, also 2D, is of number type `float32`, and is of size `Track` by `Dim3`.

These dimensions and the order of the dimensions are specified by the `DimList` parameter. Note the use of parentheses to designate an ordered list of objects. Note also that even though these two arrays share a dimension, that sharing does not necessarily imply any relationship: they just both happen to be the same size in that dimension.

The ordering of the dimensions is important. The first dimension listed is always going to be taken as the slowest varying dimension. This first dimension can if desired be declared as the UNLIMITED dimension of netCDF and HDF arrays. What UNLIMITED means in is that after the file is written, it will be possible to increase the size of the array in this first, UNLIMITED, dimension, without having to rewrite the entire HDF file. This is not possible with any other dimension.

For arrays and tables that contain geolocation information, we have established the convention of calling the group corresponding to those components GeoParameter. Other than the name difference, everything else about the GeoParameter group is identical to the DataParameter group.

Some of this information will *also* be available using standard HDF calls. For example, you can easily find out the dimensionality and number type of any array in an HDF file by using standard HDF calls. This means that some information will be stored in two places: in the standard HDF places (dimensions, scales) and in the StructMetadata metadata block. We think this minor duplication of information is more than offset by the additional functionality provided by the HDF-EOS conventions.

2.5.1.3 Geolocation Information

Much of the justification for HDF-EOS comes from the ability to provide services on the data using generic (non product-specific) routines. Most of these services revolve around using geolocation information to subset and subsample the data. To do this, we need the geolocation information stored in a known format and organization.

We can think of three ways of providing this geolocation information: by relating data arrays to corresponding arrays containing geolocation information (latitude/longitude), by using metadata to describe the mathematical relationship of locations in a data array to geolocation, and finally by specifying geolocation values for every data value in a table. These three geolocation methods are used in the three geolocated HDF-EOS structures of *Swath*, *Grid*, and *Point* respectively.

- *Swath*—Consists of data tables and data arrays that are organized along a track dimension. Can also be used for profile information. Geolocation (latitude, longitude, time) will be available as a table of values (one time value per scan line, for example), or as an array of values (with dimensions of the track dimension by the Xtrack dimension). Geolocation consists of describing the relationship of the data tables and arrays to the geolocation table or arrays.
- *Grid*—Consists of data arrays that were created using a projection (such as Mercator), with a known mathematical procedure for calculating the geolocation for every data element in the array.
- *Point*—Consists of a table of individual data values which have geolocation values associated with them.

The role of metadata in each case is different. In the Swath case, metadata is used to establish relationships between the various components. In the Grid case, metadata is used to establish the parameters for a mathematical transformation. In the Point case, metadata essentially has no role, as every data value has geolocation directly connected to it.

2.5.1.4 Swath Example

An example of swath metadata is shown below.

Dimension Example: group = Swath;

```
group = Dimension;
    object = "Track";
        size = 600;
    end_object;

    object = "Xtrack";
        size = 25;
    end_object;

    object = "Dim3";
        size = 100;
    end_object;
end_group = Dimension;

group = DataParameter;
    object = "Band_3";
        DataType = int16;
        DimList = ("Track","Xtrack");
    end_object;

    object = Temperature;
        DataType = float32;
        DimList = ("Track","Dim3");
    end_object;
end_group = DataParameter;

group = GeoParameter;
    object = "Latitude";
```

```

        DataType = float32;
        DimList = ("Track","Xtrack");
    end_object;

    object = "Longitude";
        DataType = float32;
        DimList = ("Track","Xtrack");
    end_object;
end_group = GeoParameter;

group = DimensionMap;
    object = Map1;
        DataDimension = "Dim3";
        GeoDimension = "Xtrack";
        Offset = 0;
        Increment = 4;
    end_object;
end_group=DimensionMap;

end_group=Swath;

```

The Dimension and DataParameter groups of this example have been discussed above. The structure of the GeoParameter group is identical to the DataParameter group, so its meaning should be clear. The last group, DimensionMap, requires some explanation.

What we need to do is establish a relationship between data arrays and geolocation arrays. If all of these arrays are always of the same size, say Track by Xtrack, then the relationship is pretty straightforward. In that case, there would be a one-to-one mapping of locations in the data arrays to locations in the geolocation arrays.

But suppose that the geolocation array is a different size than the data array. How do we establish the mapping of the two arrays? We do that by using the DimensionMap grouping. This grouping establishes a link not between the arrays themselves, but between the dimensions that they use. That way, if there are many data arrays (or geolocation arrays) of the same size, then we only need to establish one relationship, rather than one for every pairing of data arrays to geolocation arrays.

In the DimensionMap group above, we established a relationship between the Xtrack dimension, used for Latitude and Longitude geolocation arrays and for the Band_3 data array, and the Dim3

dimension, used for Temperature. This relationship says that every element in the Xtrack dimension matches up with every 4th element (Increment=4) in the Dim3 array, with an initial offset of zero (offset=0).

2.6 HDF-EOS Configuration Files

One of the things not addressed in the PVL examples is where exactly these components are stored. We use dimensions to describe the sizes, then we use DataParameter and GeoParameter to describe the arrays and tables, but are they stored in HDF SDSs or Vdatas? So far we have not addressed that issue.

One option would be to ignore the issue, and ask people to always access their data through our library. That way, we would control how and where each data object is stored. This is not practical, because of the wide range of software out there that uses just HDF, and does not know or want to know about our software.

We therefore propose that another component of the metadata, specified either by the creator of the datafile or automatically by our software, is a description of the configuration of the actual storage of the data arrays. An example is shown below.

```
Configuration Ex.   group = SwathObject;
                   object = "Band_3";
                   end_object = "Band_3";

                   object = "Temperature";
                   end_object = "Temperature";

                   object = Sds_1;
                   Planes = ("Latitude", "Longitude");
                   end_object;

                   end_group = SwathObject;
```

In this example, the HDF file consists of three SDSs, named Band_3, Temperature, and Sds_1. The first two SDSs are 2D, and both contain a single 2D array. Metadata parameters associated with each of these two SDSs would be sandwiched inside the object, end_object entries for those two arrays.

The third SDS is 3D, and contains two 2D arrays (Latitude and Longitude). In general, when dealing with a series of 2D arrays of the same size and number type, it is more efficient to combine them in a single 3D array. In this example, the 3D array Sds_1 is of size 2 (number of 2D arrays) by 600 (Track) by 25 (Xtrack).

Why is it useful to explicitly list the locations of the HDF-EOS arrays in the HDF file, instead of making their locations hidden inside the subroutine library? We can think of a few reasons. The first is that for people using straight HDF, this configuration information describes, in a human-readable form, where every data array is stored in HDF terms. The second is that if someone wanted to, they could write a routine to parse this PVL description and read the arrays directly, without using our subroutine library. The third is that the data producer may want to organize the data arrays in very particular way, to for example make subsetting in a particular way more efficient.

Where is this configuration information created? We propose the following: The information is always stored in the HDF-EOS file, as part of the metadata for a particular object, be it a swath, point, or grid. The information can either be specified by the data producer, if they want the arrays placed in particular places, or it can be generated automatically by our HDF-EOS file creation routines.

We welcome your comments on this brief discussion paper on Metadata.

Brand Fortner (bfortner@eos.hitc.com)

Doug Ilg (dilg@ulabsgi.gsfc.nasa.gov)

David Wynne (davidw@eos.hitc.com)

Abbreviations and Acronyms

ECS EOSDIS Core System