170-WP-003-001

The HDF-EOS Swath Concept

White Paper

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

December 1995

Prepared Under Contract NAS5-60000

RESPONSIBLE ENGINEER

David Wynne / B. Fortner \s\	12/20/95
Doug Ilg David Wynne Brand Fortner EOSDIS Core System Project	Date

SUBMITTED BY

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

Hughes Applied Information Systems
Upper Marlboro, Maryland

This page intentionally left blank.

Abstract

This document is a preliminary release of a design document for the storage of EOS Swath data in the hierarchical data format (HDF), for the Earth Observing System (EOS), Version 1 and later. We talk about what our thoughts are on using HDF to store swath type data, and concentrate on establishing the organization and structure of the geolocation information. We also show how we plan to use metadata to describe relationships between data and geolocation, and what instrument independent services we can expect to perform on these HDF-EOS Swaths.

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, Swath, Data Formats, PVL, ODL, Standard Data Products, Disk Formats, Browse, Arrays

This page intentionally left blank.

Contents

Abstract

1. Introduction

2. The HDF-EOS Swath Concept

Appendix A. Swath "Attributes"

Figures

Tables

1. Introduction

1.1 Purpose

This discussion paper introduces our current thoughts on storing *time-ordered* data in an HDF-EOS file. Below is a discussion about what is meant by ‘time-ordered’, and how this type of data can be stored and organized as an HDF-EOS structure. This structure is called a “swath” because most EOS swath data fits naturally into this structure. However, other time-ordered data, such as profiles, also fit into an HDF-EOS Swath structure, so it is important not to take the term too literally.

Important Notes:

1) This paper is oriented toward implementation in the “C” programming language. The Fortran-literate reader is cautioned that dimension ordering is row-major in C (last dimension varying fastest), whereas Fortran uses column-major ordering (first dimension varying fastest). Therefore, Fortran programmers should use dimensions in the reverse order to that shown in this document.

2) The main purpose of this document is to aid the HDF-EOS team in solidifying their ideas about the internal design of the swath structure and to provide the basis for the development of an API for swath data. It is not intended to be used as a product design tool. However, due to time constraints on some instrument teams, it is likely that some coding will need to begin well before the emergence of a convenient programming interface. We ask that Instrument teams with tight time constraints keep in close contact with the HDF-EOS team, to ensure design compatibility.

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. The HDF-EOS Swath Concept

2.1 What is an HDF-EOS Swath?

The Swath concept for HDF-EOS is based on a typical satellite swath, where an instrument takes a series of scans perpendicular to the ground track of the satellite as it moves along that ground track. Figure 2-1 below shows this traditional *physical* view of a prototypical swath.[1]

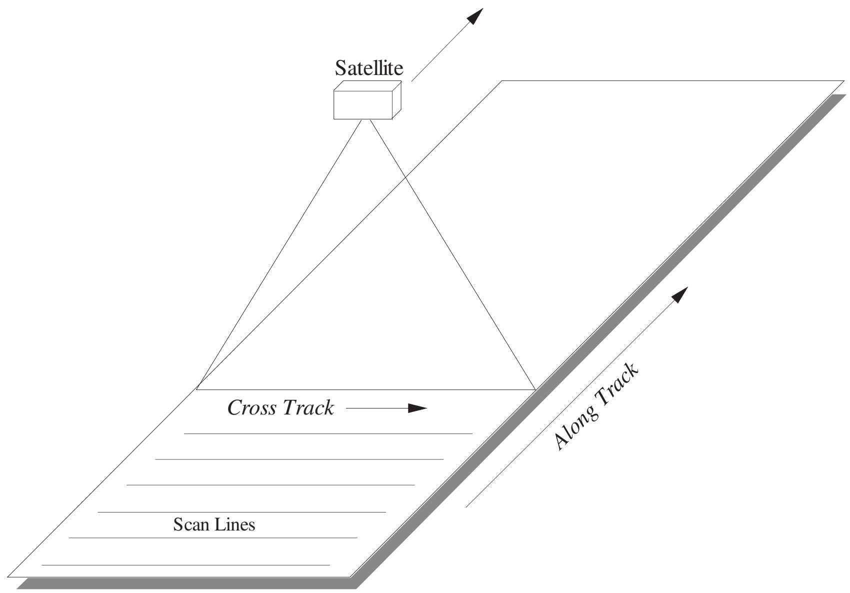

Figure 2-1. Physical view of a simple swath: a time-ordered series of scan lines

The HDF-EOS *data* view of a swath is one where the data is ordered by *time* or a *time-like variable* (e.g., scan line counter). The data stored for every time entry can consist of time, geolocation (latitude, longitude), scalar values, 1D arrays of values (scan lines or profiles), or 2D arrays of values (multiple channel scan lines). A conceptual example is shown in Figure 2-2.

[1] Please remember that the use of this view as a prototype does not preclude storing other types of swaths in this data model (such as profiles).

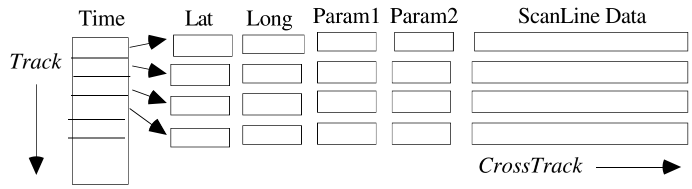

Figure 2-2. Data view of a swath: a time-ordered series of scalars and arrays

In this figure, each Time value has associated with it a Latitude and Longitude, two scalar values (Param1, Param2), and a 1D array containing ScanLine Data. Conceptually, each named item can be considered as a separate array. For example, in the figure above Time would be a 1D array, as would Lat, Long, Param1, and Param2. The ScanLine Data would be a 2D array. The final step in assembling the data view is to combine all the data objects that have the same dimensions.

The concepts of '*Track*' and '*Cross-Track*' dimensions are fundamental to this swath discussion. The 'Track' dimension (sometimes refered to as 'along track') is defined as the dimension where time or a time-like variable increases monotonically. The 'Cross-Track' dimension, if used, defines an on-ground dimension perpendicular to the Track dimension.

These two dimensions are fundamental because our geolocation services will depend on knowing which array dimensions correspond to 'Track' and 'Cross-Track'. We also define 'detector', 'band', 'channel', and 'parameter' conceptual dimensions, but since these dimensions are not used for geolocation services, it is not as important to standardize their usage.

Table 2-1 shows the available conceptual dimensions for *scanning* instruments. Table 2-2 shows the same information for *profiling* instruments, and Table 2-3 shows the same information for a *combination scanning-profiling* instrument, such as TRMM PR. A quick comparison of Tables 2-1 and 2-2 will demonstrate the remarkable similarity between scanning and profiling instruments in the context of the data view.

Table 2-1. Dimension definitions for a generic scanning instrument

Dimension	Description	Comments
Track	Parallel to the ground track of the satellite	Required; must be the first declared dimension*
Cross-Track	Perpendicular to the ground track of the satellite and parallel to the surface of the Earth	Required; ordering is unimportant
Detector	Number of foot prints per dwell	Optional; ordering is unimportant
Band or Channel	Generally used for lower level data that has not been processed into science parameters	Optional; ordering is unimportant; Band and Parameter are mutually exclusive
Parameter	No physical mapping; generally used for higher level data that has been processed into science parameters	Optional; ordering is unimportant; Band and Parameter are mutually exclusive

* "C" dimension order is assumed. In Fortran, the last declared dimension must be used.

Table 2-2. Dimension definitions for a generic profiling instrument

Dimension	Description	Comments
Track	Parallel to the ground track of the satellite	Required; must be the first declared dimension
Profile	Perpendicular to the ground track of the satellite and "vertical" with respect to the Earth	Required; ordering is unimportant; equivalent to atmospheric level
Detector	Number of foot prints per dwell	Optional; ordering is unimportant
Band or Channel	Generally used for lower level data that has not been processed into science parameters	Optional; ordering is unimportant; Band and Parameter are mutually exclusive
Parameter	No physical mapping; generally used for higher level data that has been processed into science parameters	Optional; ordering is unimportant; Band and Parameter are mutually exclusive

* "C" dimension order is assumed. In Fortran, the last declared dimension must be used.

Table 2-3. Dimension definitions for a generic scanning-profiling instrument

Dimension	Description	Comments
Track	Parallel to the ground track of the satellite	Required; must be the first declared dimension*
CrossTrack	Perpendicular to the ground track of the satellite and parallel to the surface of the Earth	Required; ordering is unimportant
Profile	Perpendicular to the ground track of the satellite and "vertical" with respect to the Earth	Required; ordering is unimportant; equivalent to atmospheric level
Detector	Number of foot prints per dwell	Optional; ordering is unimportant
Band or Channel	Generally used for lower level data that has not been processed into science parameters	Optional; ordering is unimportant; Band and Parameter are mutually exclusive
Parameter	No physical mapping; generally used for higher level data that has been processed into science parameters	Optional; ordering is unimportant; Band and Parameter are mutually exclusive

* "C" dimension order is assumed. In Fortran, the last declared dimension must be used.

To apply these concepts to a particular instrument, the producer must determine the appropriate dimensions to use and arrange them in an acceptable order (see comments in Tables 2-1, 2-2 and 2-3). The names of the dimensions are meant only as points of reference. The actual names of the dimensions assigned within a swath structure are defined by the data producer.

For multidimensional arrays stored in the swath, the first dimension of the array must always be the 'Track' dimension. This limitation makes it possible to append data to swaths after file creation. This is because HDF allows the first dimension of an array to be appended to (the "unlimited" dimension, in HDF/netCDF speak). No case is known within EOS for which data must be appended along any other dimension.

2.2 The Components of a Swath

A single swath structure can contain any number of *Tables* (1D arrays stored as HDF Vdatas) and *Multidimensional Arrays* (stored as HDF SDSs). There is however one type of information that is special: *geolocation information*.

In a swath, geolocation information can be stored as a table, as a series of arrays, or as a combination of a table and arrays. For example, you may want to store Latitude and Longitude for every grid location (two 2D arrays, one for Latitude, one for Longitude), and also a Time value for every scanline (one 1D table). We require that every swath contain some geolocation component. The data itself can be stored in tables or multidimensional arrays in the swath. Again, the only limitation is that the first dimension (or the only dimension, in the case of a table) is the *Track* dimension.

These possible components are also summarized in Table 2-4. A particular swath may have multiple instances (or no instances) of any of these different components, except the geolocation objects: there cannot be more than one geolocation table and there should be a minimum of geolocation arrays in a single swath (having one table *and* a few arrays is okay).

This limitation is because the definition of a swath assumes that there is only a single geolocation structure (which may consist of a table, a set of arrays, or both) that is referenced by all of the data tables and arrays in the Swath structure. If a particular EOS granule requires supporting multiple geolocation objects, then the data should be stored as separate Swath structures, one per geolocation object.

Table 2-4. Possible Components of a Swath Structure

Component Type	Dim.	Comments	Storage
Geolocation Table	1D	Time, Scan Number, Track Counter	Table (Vdata)
Geolocation Array	2D	Latitude, Longitude, etc.	Array (SDS)
Data Table	1D	Scalar values per Track entry	Table (Vdata)
2D Data Array (scanner)	2D	Scan Lines per Track entry	2D Array (SDS)
2D Data Array (profiler)	2D	Profiles per Track entry	2D Array (SDS)
3D Data Array (scanner)	3D	Multiple Scan Lines per Track entry	3D Array (SDS)
3D Data Array (profiler)	3D	Multiple Profiles per Track entry	3D Array (SDS)

2.3 Defining a Swath

The reason for creating a swath structure is to be able to provide services on the swath that are *instrument independent*. For example, subsetting and subsampling by geolocation could be provided on data stored in a swath structure, independent of the instrument and product which that data represents.

Of course, the same services *could* be provided on the data even if it is not stored according to the swath conventions. However, the code to supply those services would have to be custom written for every instrument product.

These services all revolve around geolocation information (time, latitude, longitude, etc.). The primary considerations for the swath structure, therefore, concern these variables, what their format is, their dimensionality, and what the relationship is between other data arrays and the geolocation table and/or array(s).

There are two components to establishing a swath structure:

1. defining the geolocation information, and
2. defining the relationships between data and geolocation.

Each of these is discussed in turn below.

2.3.1 Defining Geolocation

There are four specific geolocation parameters which will be recognized and supported by the swath data interface: *Time*, *Latitude*, *Longitude*, and *Colatitude*. Some combination of these parameters must exist in a swath structure for the interface to be able to perform services on the data, such as geolocation-based subsetting and subsampling. A minimally serviceable swath structure will contain only Time, whereas a fully serviceable swath will have Time, Longitude, and either Latitude or Colatitude.

In Figure 2-3 we show a geolocation table (1D Vdata) that contains Time, Latitude, and Longitude columns for geolocation information per scan line. Note how the separate columns are combined into the same Vdata.

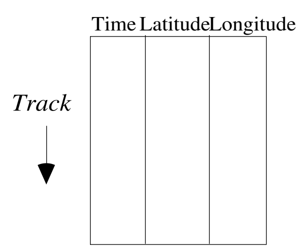

Figure 2-3. Geolocation Table with 1D Geolocation Information Included

For cases where geolocation exists for every Track, Cross-Track location, then the Time, Latitude, and Longitude arrays should be two-dimensional. In Figure 2-4 we show two 2D arrays of Latitude and Longitude, which for convenience have been combined into a single 3D array. We will discuss this combining of 2D arrays in more detail in a later section. For now, consider it something that we do behind the scenes, which should have no effect on the conceptual understanding of the geolocation information.

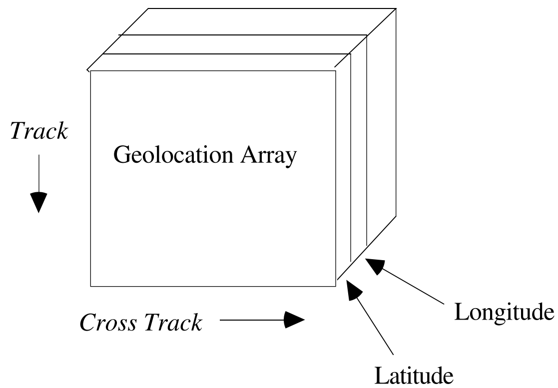

Figure 2-4. Geolocation Array containing Latitude and Longitude planes

2.3.3 Defining the Relationship to Geolocation

The next step is to match up the data elements with the geolocation parameters. This is done by mapping dimensions of the data elements to the 'Track' and 'Cross-Track' dimensions of the geolocation parameters. The mapping of data to geolocation is described in a block of PVL text in an attribute named "SwathStructure"[2] that is associated with the swath data structure itself. We will describe the contents of the "SwathStructure" attribute in more detail in the next section on Swath Configuration files.

For example, suppose you had two 2D arrays, one a geolocation array (containing say Latitude), the other a data array (containing say Temperature). The following PVL segment defines these two arrays and their dimensions (again, we will discuss the details of this code in the next section):

PVL Code Example:

```
group = "Dimension";
   object = "GeoTrack";
      Size = 1200;
   end_object = "GeoTrack";

   object = "GeoCrossTrack";
      Size = 200;
   end_object = "GeoCrossTrack";

   object = "DataX";
      Size = 600;
   end_object = "DataX";
```

---

[2] Actually, this is only half right. HDF only supports attributes on SDSs and whole files, so swath attributes need to be simulated. The method for constructing a swath attribute will be discussed later in this document.

```
   object = "DataY";
      Size = 200;
   end_object = DataY;
end_group = "Dimension";

group = "GeoParameter";
   object = "Latitude";
      DataType = float32;
      DimList = "GeoTrack, GeoCrossTrack";
   end_object = "Latitude";

   object = "Temperature";
      DataType = float32;
      DimList = "DataX, DataY";
   end_object = "Temperature";
end_group = "GeoParameter";
```

Now the next step is to define the relationships between the data array and the geolocation array. We do that by another PVL entry of DimensionMap. The PVL code below shows a template for this entry.

PVL Template:

```
group = "DimensionMap";
   object = <map name>;
      DataDimension = <dimension name>;
      GeoDimension = <dimension name>;
      Offset = <value>;
      Increment = <value>;
   end_object = <map name>;
         .
         .
         .
end_group = "DimensionMap";
```

A DimensionMap entry is interpreted as follows:

- DataDimension is the name of the dimension of the data object being mapped.
- GeoDimension is the name of the dimension the geolocation object being mapped to.
- Offset is the offset into the data array along DataDimension where the first geolocation value applies. A negative value indicates that the offset is applied to GeoDimension instead, which is useful in cases where the geolocation object is larger than the data object.
- Increment is the increment along DataDimension for which there is geolocation data in the Geolocation object. A negative value indicates that the increment is applied along GeoDimension[3] , which is useful in cases where the geolocation object is larger than the data object.

[3]A value of `-n` is actually meant to indicate a stride of $^1/_n$. The negative value is used in order to avoid the rounding error and interpretation issues inherent in floating point values.

So for example of two arrays, we would need the following two `DimensionMap` entries:

PVL Code Example:
```
group = "DimensionMap";
   object = "Map1";
      DataDimension = "DataX";
      GeoDimension = "GeoTrack";
      Offset = 0;
      Increment = -2;
   end_object = "Map1";

   object = "Map2";
      DataDimension = "DataY";
      GeoDimension = "GeoCrossTrack";
      Offset = 0;
      Increment = 1;
   end_object = "Map2";
end_group = "DimensionMap";
```

In the example PVL code above, a Increment of -2 in the first DimensionMap means that the Latitude array advances two entries (along the GeoTrack dimension) for every one entry of the Temperature array. Note that the negative is used to designate that the geolocation array is larger than the data array. In the second DimensonMap, a Increment of 1 means that the DataY and GeoCrossTrack dimensions are the same size, and map one-to-one.

In cases where a data object and a geolocation object share the same dimension, the relationship can be assumed to be a one-to-one mapping, and there is no need to explicitly define it with a DimensionMap entry.

2.4 Swath Configuration File

To simplify the creation and definition of swath structures, we have developed a text-based configuration language. The language is based on PVL, with some special-purpose keywords added to handle the specifics of defining a swath structure. The Data Producer needs to create this configuration file as part of the process of defining the HDF-EOS file.

Defining a swath configuration file consists of the following steps:

- defining the necessary dimensions,
- defining the individual data and geolocation parameters,
- defining the relationships between the dimensions of the data and the dimensions of the geolocation parameters,
- and defining the metadata for the swath.

The configuration script for a swath begins with the statement object=swath; and ends with a corresponding end_object; statement. Note that most of the contents of the “SwathStructure” discussed above will be inherited from this Swath Configuration File[4].

2.4.1 Defining Dimensions

Dimensions are defined using the following syntax:

PVL Code:

```
group = "Dimension";
   object = <dimension name>;
      Size = <dimension size>;
      Type = <dimension type>;
   end_object = <dimension name>;
end_group = "Dimension";
```

where <dimension name> is the name chosen for the dimension, and <dimension size> is the number of entries in that dimension. The Type field is optional and can be any of the following: int8, int16, int32, uint8, uint16, uint32, float32, or float64. If the Type field is omitted, the type is assumed to be int32.

2.4.2 Defining Parameters

There are two different types of parameters which the swath will accommodate: data parameters (DataParameter) and geolocation parameters (GeoParameter). Data parameters are defined using the following syntax:

PVL Code:

```
group = "DataParameter";
   object = <parameter name>;
      DataType = <parameter data type>;
      TrackDim = <dimension name>;
      CrossTrackDim = <dimension name>;
      DimList = <dimension names>;
   end_object = <parameter name>;
      .
      .
      .
end_group = "DataParameter";
```

and, geolocation parameters are defined using the following syntax:

PVL Code:

```
group = "GeoParameter";
   object = <parameter name>;
      DataType = <parameter data type>;
      TrackDim = <dimension name>;
      CrossTrackDim = <dimension name>;
```

---

[4]However, some of the entries in “SwathStructure” will be created by the HDF-EOS library, not by the human data producer. We will discuss this in more detail in a separate paper on HDF-EOS structural metadata.

```
            DimList = <dimension names>;
         end_object = <parameter name>;
            .
            .
            .
      end_group = "GeoParameter";
```

where <parameter data type> is one of (uint8, uint16, uint32, int8, int16, int32, float32, float64), and DimList is a comma-separated list of the names of the dimensions. Each dimension must be previously defined, and the order in which they appear in the parameter definition is taken as the order of the dimensions of the array as they would be declared in a C program.

2.4.3 Defining Relationships from Data to Geolocation

As we discussed in Section 2.3, relationships between data dimensions and geolocation dimensions are defined using the following syntax:

PVL Code:

```
group = "Dimension Map";
   object = <map name>;
      DataDimension = <dimension name>;
      GeoDimension = <dimension name>;
      Offset = <dimension name>;
      Increment = <dimension size>;
   end_object = <map name>;
      .
      .
      .
end_group = "Dimension Map";
```

where DataDimension specifies a dimension used for defining data arrays or tables, GeoDimension specifies that there should be one DimensionMap entry for the Track dimension and CrossTrack dimension (where applicable) dimensions of each data parameter to the corresponding dimension of each geolocation parameter. As mentioned earlier, there is no need for a DimensionMap in cases where a dimension is shared between a data and geolocation parameter.

2.4.4 Defining Metadata

You can also include additional PVL statements within the swath definition which are not required by the interface. These fields may be attached to specific parameters by including them in the definitions of those parameters. Fields which are not associated with any particular parameter or data object will be stored as general metadata for the swath.

All of the general metadata fields will be collected into a swath attribute named SwathMetadata, as we discussed previously. The SwathMetadata attribute will preserve the association of the individual metadata fields with particular parameters through the use of PVL grouping structures.

2.5 An Example Swath

We will now walk through an example swath to illustrate the concepts discussed in this paper. Consider Figure 2-5, which is a representation of a swath consisting of a 3D data array, a series of 2D geolocation arrays, and a single 1D geolocation table.

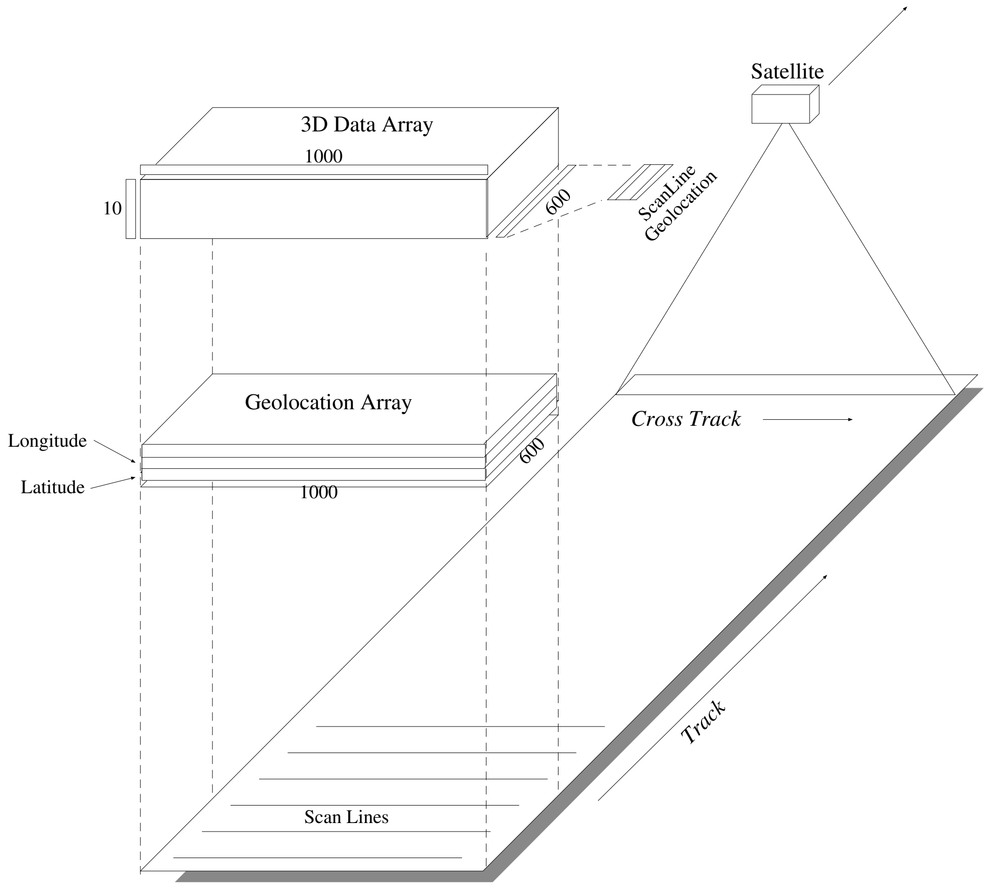

Figure 2-5. Conceptual View of Example Swath, with 3D Array, Time/Geolocation Array, and Geolocation Table

The components are listed in Table 2-5. Note one strange thing: we have always talked about geolocation arrays such as Latitude and Longitude being 2D arrays, with dimensions of 'Track' and 'Cross-Track'. So what is this 3D Geolocation array?

The answer is that for efficiency reasons, our HDF-EOS subroutine library will combine 2D arrays into 3D arrays where possible (when the dimension sizes, number type, and number of dimensions all match). Usually, this combining will happen behind the scene, and neither the program creating the HDF-EOS file or the program or person reading the HDF-EOS file need to know about this, as long as everyone uses our HDF-EOS subroutines.

Table 2-5. Components of Example Swath

Component	Dim.	Size	Comments
Data	3D	600×1000×10	Track Dim. always first
Geolocation Array	3D	600×1000×2	Lat. and Long. combined
Geolocation Table	1D	600	Several Columns

The Swath Configuration file for this example swath may look something like the following. For this example, the 3D data array is named Temperature, and the single column in the geolocation table is named Time. Note that the configuration file does not specify the fact that Latitude and Longitude will be combined; that happens automatically. Note also that the SwathStructure attribute that will actually be stored in the data file will contain most of the contents of the configuration file, but will also include additional entries related to this combining of arrays. Our HDF-EOS routines will generate those additional entries.

PVL Code Example:

```
group = "Dimension";
   object = "GeoTrack";
      Size = 600;
   end_object = "GeoTrack";

   object = "GeoCrossTrack";
      Size = 1000;
   end_object = "GeoCrossTrack";

   object = "DataX";
      Size = 600;
   end_object = "DataX";

   object = "DataY";
      Size = 1000;
   end_object = "DataY";

   object = "Band";     /* the 3rd dim. of the data array */
      Size = 10;
   end_object = "Band";
end_group = "Dimension";

group = "GeoParameter";
   object = "Latitude";
      DataType = float32;
      DimList = "GeoTrack, GeoCrossTrack";
   end_object = "Latitude";

   object = "Longitude";
      DataType = float32;
      DimList = "GeoTrack, GeoCrossTrack";
   end_object = "Longitude";

   object = "Time";     /* the geolocation table */
```

```
        DataType = float32;
        DimList = "GeoTrack";
     end_object = "Time";
  end_object = "GeoParameter";

  group = "DataParameter";
     object = "Temperature";
        DataType = float32;
        DimList = "DataX, DataY, Band";
     end_object = "Temperature";
  end_group = "DataPrarmeter";

  group = "DimensionMap";
     object = "Map1";
        DataDimension = "DataX";
        GeoDimension = "GeoTrack";
        Offset = 0;
        Increment = 1;
     end_object = "Map1";

     object = "Map2";
        DataDimension = "DataY";
        GeoDimension = "GeoCrossTrack";
        Offset = 0;
        Increment = 1;
     end_object = "Map2";
  end_group = "DimensionMap";
```

2.6 Discussion on HDF-Related Items

As mentioned previously, our HDF-EOS routines will try as much as possible to combine arrays and tables for efficiency. In this section we go into a bit more detail about why and how we do this.

First, when evaluating a component for inclusion in an HDF file, you (or our library) needs to know the following information about that component

- *Dimensionality* - Is the component only associated with the ‘Track’ dimension, or with both the ‘Track’ and ‘Cross-Track’? (In HDF SDSs, this property is referred to as the *rank* of an SDS.)
- *Size(s)* - How many values are there along each dimension of the parameter? (In HDF SDSs, these are referred to as *dimension sizes* or *dimsizes*.)
- *Data type* - Is the value a 32-bit integer? a 16-bit integer? a 32-bit floating-point? etc. (HDF refers to this as the *numbertype*.)

The dimensionality of each parameter determines which HDF data object will be used to store it. One-dimensional parameters will be stored in Vdatas, while 2-dimensional parameters will be stored in SDSs. This distinction is made because of the comparative ease with which multiple parameters can be accommodated in a Vdata as opposed to an SDS and the lack of multiple dimensions in Vdatas.

An example of a one-dimensional geolocation parameter is a time value that is recorded once for each scan or profile, as shown in Figure 2-4. An example of a two-dimensional geolocation parameter is a latitude value which has been computed for every third pixel of each scan line of a scanning instrument, as shown in Figure 2-5.

If there are two or more components that share dimensionality, size, and datatype, then our routines can combine them into arrays or tables of higher dimensionality. We need to do this for efficiency: an HDF file with thousands of components is very inefficient, whereas one with just a small number of arrays of high dimensionality can be accessed very efficiently.

For example, geolocations components are always either 1D or 2D. 1D geolocation components are always associated with the 'Track' dimension, whereas 2D geolocation components are always associated with both the 'Track' and 'Cross-Track' dimensions. If two geolocation components have the same dimensionality (2D, for example), have the same size (600 by 1000, for example), and the same number type (float32, for example), then we can combine them into a larger component (a 3D array of size 600 by 1000 by 2, for example).

Note, however, that although arrays that are combined must be of the same number type, this limitation is not imposed on 1D tables. An HDF table (implemented with a Vdata) can have columns of data of differing number types. Note that the same considerations for array and table combination apply to the data components also.

In the HDF-EOS library, the combination of components happens any time the same dimension are used to describe two parameters and their types permit it. If this is not the desired behavior, users are advised to "force" the issue by defining separate dimensions for the offending parameters.

Appendix A. Swath "Attributes"

As mentioned earlier in this paper, there may be cases where individual swath data structures (which are implemented as Vgroups) will need to have attributes attached to them, such as SwathMetadata, SwathStructure or other producer-defined attributes. In such cases, it will be necessary for producers to create their own attribute structures which emulate HDF attributes (since Vgroups do not currently support attributes). This section describes how that is done.

In HDF, attributes are implemented as a very specific type of Vdata. The name of the Vdata must be the attribute name and its class must be "Attr0.0", and it must have one field named "VALUES" (case is significant). The number type of the field must be set appropriately for the data to be stored (SwathMetadata and SwathStructure attributes must be of the type DFNT_CHAR8).

For character attributes, the `VALUES` field must be declared with an *order* equal to the number of characters to be stored. For numeric attributes, multiple values may be stored as successive records in the Vdata. The completed Vdata must be a member object of the Vgroup representing the swath structure.

```
vdata_id = VSattach(file_id, -1, "w");
VSsetname(vdata_id, "SwathMetadata");
VSsetclass(vdata_id, "Attr0.0");
VSfdefine(vdata_id, "VALUES", DFNT_CHAR8, 38);
VSsetfields(vdata_id, "VALUES");
VSwrite(vdata_id, buffer, 1, FULL_INTERLACE);
Vinsert(vgroup_id, vdata_id);
VSdetach(vdata_id);
```

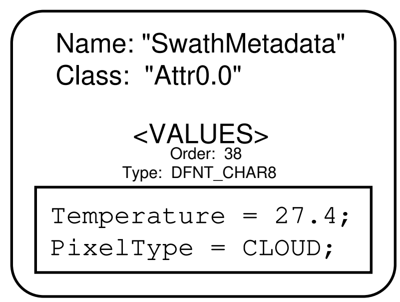

Figure A-1. A Character Attribute

The code required to create a *character* attribute and the resultant Vdata are shown in Figure A-1. The same information is shown for a *numeric* attribute in Figure A-2. Both figures assume the use of the C language and the presence of an open file with ID file_id and an open Vgroup representing the swath structure with ID vgroup_id.

```
vdata_id = VSattach(file_id, -1, "w");
VSsetname(vdata_id, "MyAttribute");
VSsetclass(vdata_id, "Attr0.0");
VSfdefine(vdata_id, "VALUES", DFNT_FLOAT32, 1);
VSsetfields(vdata_id, "VALUES");
VSwrite(vdata_id, buffer, 3, FULL_INTERLACE);
Vinsert(vgroup_id, vdata_id);
VSdetach(vdata_id);
```

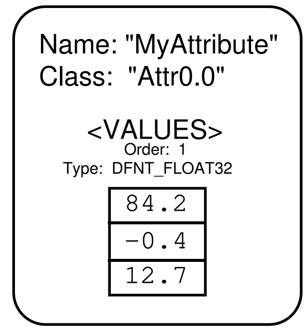

Figure A-2. A Numeric Attribute