175-WP-001-002

An HDF-EOS and Data Formatting Primer
for the ECS Project

White Paper

March 2001

Prepared Under Contract NAS5-60000

RESPONSIBLE ENGINEER

E. M. Taaheri /s/   3-1-2001

David Wynne, Abe Taaheri   Date
EOSDIS Core System Project

SUBMITTED BY

 Larry Klein /s/   2-28-2001

Darryl Arrington, Manager Toolkit   Date
Larry Klein, Manager Toolkit
EOSDIS Core System Project

Raytheon Company
Upper Marlboro, Maryland

This page intentionally left blank.

Abstract

This document is a release of a primer manual for using the Hierarchical Data Format (HDF) and extensions (HDF-EOS) in Earth Observing System (EOS), Version 1 and later. We confine our discussion in this document to HDF V4 and the versions of HDF-EOS (V2), which are based on that version of HDF. A new version of HDF (V5) has been released. Additional material for HDF5 and versions of HDF-EOS, based on HDF5 will be provided in a separate document.

Comments and suggestions should be sent to:

Larry Klein
Emergent Information Technologies
9315 Largo Dr. West, Suite 250
Largo, MD 20774   USA

Email: larry@eos.hitc.com
Phone: (301) 925-0764
Fax: (301) 925-0321

Keywords: HDF, HDF-EOS, Swath, Grid, Point, Data Formats, Metadata, Standard Data Products, Disk Formats, Browse

This page intentionally left blank.

Table of Contents

Abstract

1. Introduction

2. An Introduction to HDF

3.   HDF-EOS V2

4.   Related Topics

List of Figures

List of Tables

References

Appendix A. Obtaining Software

Appendix B. Additional HDF Topics

Glossary and Acronyms

This page intentionally left blank.

1. Introduction

1.1   Purpose of this Document

The purpose of this document is twofold. The first purpose is to introduce the Earth Observing System Data and Information System (EOSDIS) Version 1 (V1) community to the HDF file format that has been chosen as the EOSDIS Core System (ECS) Standard Data Format Our intention is to provide enough background information so that EOS personnel that need to use HDF today can do so as easily as possible. Additional and more detailed information is provided in referenced Users Guides.

The second purpose of this document is to discuss extensions to HDF, developed for the ECS program. These extensions are called HDF-EOS and are primarily used to provide project-wide standards for attaching geo-spatial and temporal information to science data. The format is also a container for inventory and product specific metadata, the information also stored in ECS databases, used for search and order functions.

1.2   Organization of this Document

Section 2 presents an overview of the HDF file format, and how the current file format maps to EOS data. In particular, we go over basic HDF concepts such as general philosophy and core data types. We next discuss NCSA HDF library routines in detail.

Section 3 presents the HDF-EOS format. This new format consists of the HDF standard with EOS conventions. We discuss the overall philosophy of HDF-EOS and the HDF-EOS data types that we support.

In Section 4, related and supplemental reference material is presented. ECS metadata and browse data are discussed. In addition we discuss EOSView, our 'HDF-EOS cracker tool.'

A section on related and supplemental reference material is provided

Appendix A provides information on how to obtain the software libraries.

Appendix B discusses issues related to HDF and HDF-EOS usage.

1.3   HDF and EOS: Introduction

The Hierarchical Data Format (HDF) was selected by the NASA ESDIS Project as the format of choice for standard product distribution. HDF was originally developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois to help with the storage of supercomputer simulation results.

Documentation on the disk format of HDF files is readily available. This is in contrast to other general scientific formats, such as network common data format (netCDF) and common data

format (CDF), where the emphasis on the data model is strong and documentation on the disk format is limited.

At the most basic level, an individual HDF file consists of a directory and a collection of data objects. Every data object has a directory entry, containing a pointer to the data object location, and a flag defining the datatype for that data object. NCSA has defined around a hundred different datatypes; users can define additional datatypes.

Many of the NCSA defined datatypes map very well to EOS datatypes used to store remotely sensed Earth science data products. Examples would include raster images, multi-dimensional arrays, and text blocks. There are other EOS datatypes that do not map as well to NCSA datatypes. Examples would include gridded data.

The HDF format is known for its generality, in that there are a very large number of legal ways to organize data in an HDF file. But this generality comes at a price: there is no guarantee that all data producers will store particular information such as geolocation in a particular way in the HDF files. Likewise, temporal information associated with science data could be stored in a variety of ways.

There were two ways to solve this problem: one way would be to explicitly define the layout of every EOS standard data product, and then incorporate these layout decisions in a subroutine library. The other method, and the one that we implemented as HDF-EOS V2, is to define new EOS specific datatypes such as Grid, Swath, and Point, that contain information in a specific structure. That way, the HDF-EOS library has to know only about these structures, and not about every single data product.

The goal is to make HDF-EOS files completely self-describing, so no outside information will be needed to display the information contained in the files. The HDF-EOS library is intended to make data access much more convenient for both producers and users.

NOTE: In this document, we discuss HDF V4 and versions of HDF-EOS (2.X), based on HDF4. Current Landsat 7, TRMM and EOS Terra instrument data, now being processed and archived in ECS Distributed Active Archive Centers (DAACs) are based on HDF4 and HDF-EOS 2 formats. Data from instruments on EOS Aqua, to be launched in 2001, are also based on those formats. Many EOS ancillary data products, such as Digital Elevation Model (DEM) data and converted NOAA GIRD and BUFR data are also written in HDF-EOS 2.

NCSA has released HDF5, which is a nearly complete rewrite of HDF4. HDF5 has a different user interface and underlying data model. HDF-EOS V5, based on HDF5 is available to EOS instrument teams and is the format of choice for EOS-Aura teams. Aura is scheduled to be launched in 2003. HDF-EOS 5. is designed to be very similar to HDF-EOS 2, supporting the same Grid, Swath, and Point data structures. Access to and conversions between both formats will be provided. It is anticipated that the HDF standard will migrate to HDF5-based files in the future. It is not anticipated however, that conversions will be done on EOS standard products until several years following public releases in 2000. It will be our goal with HDF-EOS 5. development, to make the differences between the underlying HDF4 and HDF5 formats, as transparent to users as possible. HDF5 and HDF-EOS 5 will be discussed in a separate Primer. References to HDF in this document refer to HDF4 and HDF-EOS 2.

175-WP-001-002

1.4   HDF and EOS: General Philosophy

For reference, we include the initial NASA direction concerning EOS and standard formats for EOS standard products. The following is a statement from the Earth Science Data and Information System (ESDIS) office concerning HDF:

> "The Earth Science Data and Information System Project has undertaken an analysis of available data format standards over the last 4 years. This analysis received input from Distributed Active Archive Centers (DAACs), EOS Instrument Investigators, related earth science projects, international investigators, computer scientists, and other members of the EOS community.

> As a result of this study, the ESDIS Project selected the National Center for Supercomputer Applications' Hierarchical Data Format (HDF) as the Standard Data Format (SDF) for Version 0 System distribution of science data.

> Based on successful experience in Version 0, including use by ECS DAACs, the Pathfinder project, and associated earth science projects, the ESDIS Project plans to adopt HDF as the baseline EOSDIS Standard Data Format for science and science-related data. In 1994, HDF was adopted as a baseline standard for EOSDIS Core System development of standard data product generation, archival, ingest, and distribution capabilities.

> The ESDIS Project will support the evolution of the EOSDIS SDF as needed to meet the requirements of science data users and producers. "

This statement provided for the beginning of HDF and HDF-EOS development for the ECS Project.

2. An Introduction to HDF

The purpose of this part of the primer is to give a conceptual overview of HDF, and how EOS data may fit into the existing HDF structure. This material is *not* meant to be a replacement for the HDF manuals available from NCSA, but as a help in the comprehension of the concepts presented there. (See documents and software at: http://hdf.ncsa.uiuc.edu)

2.1   HDF Concepts

The Hierarchical Data Format, or HDF, is a multi-object file format for storing scientific data in a distributed environment. HDF was created at the National Center for Supercomputing Applications to serve the needs of diverse groups of scientists working on projects in various fields. HDF was designed to address many requirements for storing scientific data, including:

- Support for the types of data and metadata commonly used by scientists.

- Efficient storage of and access to large data sets.

- Platform independence.

- Extensibility for future enhancements and compatibility with other standard formats.

HDF is a *disk format* and *subroutine library* for storage of most kinds of scientific data. HDF is intended for use in the storage of any kind of scientific data, although support is strongest for multi-dimensional arrays and raster images. It also contains very good support for the organization of data into hierarchical layers.

The *HDF disk format* is strictly binary, although ASCII text annotations are supported. There is a very strong emphasis on portability and machine independence, unusual for a binary format. A strength of HDF is that a single data file can contain several different *types* of objects. A color image of a molecule may be stored in the same file as the data object containing the actual positions of the atoms in space. The file may also contain an ASCII text annotation notebook describing the molecule.

At its most fundamental level, an HDF file consists of a directory and an unordered set of binary data elements. For the most part, the directory entries match up one-to-one with the binary data elements that follow. Each directory entry describes the location, the type, and the size of the corresponding element. We discuss the disk format in depth in Section 2.3.

The *HDF subroutine library* is designed to be very easy for C and FORTRAN programmers to use. Many simple HDF reads and writes can be accomplished with a single subroutine call. For example, to write a C character array that represents an 8-bit color image to an HDF file, the following HDF call is all that is required (the FORTRAN example is similar).

175-WP-001-002

```
ret=DFR8addimage("myfile.hdf",image1,rows,cols,0);
```

Figure 2-1. HDF call to write an 8-bit image

This single call to the routine DFR8addimage creates the file 'myfile.hdf', opens it, initializes 'myfile.hdf' as an HDF data file, writes the image to 'myfile.hdf', adds an HDF directory entry in the file for the image, and closes the file. The last argument in the call specifies the compression method, here 0 (no compression).

The HDF library is accessible from both C and FORTRAN programs because it contains a set of 'wrapper' functions that make the underlying C code callable from FORTRAN. Some FORTRAN compilers only accept function names that are eight or fewer characters. HDF therefore provides two names for each function; one for use in C programming and a shorter version for use in FORTRAN programming. For example, d8aimg is the FORTRAN equivalent for DFR8addimage. We discuss the HDF subroutine library in depth in Section 2.2.

Among widely used general scientific data formats, HDF may be unique in that the HDF libraries and manuals are in the *public domain*. This means that the HDF software and documentation can be used in commercial products without any licensing or even acknowledgment. The best source for HDF materials is via the NCSA anonymous file transfer protocol (ftp) server (See Appendix B).

2.2   Overview of the NCSA HDF Library

The HDF library can be thought of as three interface layers built upon a physical file format. The first interface layer, or the *low level layer*, is reserved for software developers. It provides support for things like file I/O, error handling, memory management, and physical storage. It is essentially a software toolkit for skilled programmers who wish to make HDF do something more than what is currently available through the higher level interfaces.
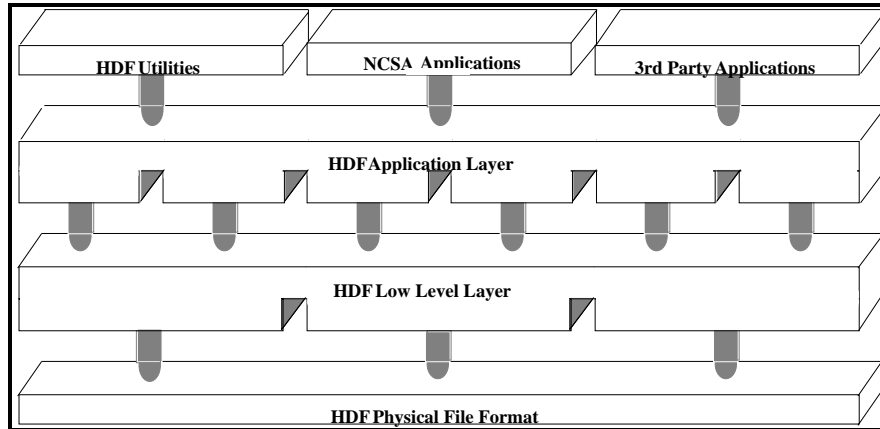
Figure 2-2. The HDF physical file format supports three levels of interaction

Above the low level interface layer is the HDF *single file* and *multi-file application layer*. The application layer includes routines designed to simplify the process of storing and accessing data. Most HDF developers spend the majority of their time working with the application interface. Although each interface module requires some programming, all the low level details can be ignored.

At its highest level, HDF includes utilities, NCSA applications, and a variety of third party applications. Applications developed by NCSA, as well as applications contributed by HDF users, are freely available on the NCSA ftp server. In addition, several software vendors also support HDF.

The HDF library consists of callable routines in the low level layer and in the application layer. Underneath each layer, the routines are grouped into interfaces. Each interface addresses a particular HDF function or a particular HDF data structure. All the callable routines within a particular interface begin with the same letters. The different interfaces are therefore known by these letters. Table 2-1 below lists all the HDF interfaces, grouped by layer. Examples of callable routines from each interface are given in the last column of the table.

175-WP-001-002

Table 2-1. The HDF Interfaces

Interface	Description	Example Routines (long names)
Low Level Layer Interfaces		
H	low level I/O, directory, query	Hopen, Hread, Hwrite, Hcreate
HDF	new version of low level routines	HDFopen, HDFclose
HE	low level error reporting	HEreport, HEprint
Single File Application Layer Interfaces (old)		
DFR8	read, write 8-bit raster images	DFR8addimage, DFR8getdims
DFP	read, write color palettes	DFPaddpal, DFPgetpal
DF24	read, write 24-bit raster images	DF24addimage, DF24setdims
DFSD	single file scientific dataset	DFSDputdata, DFSDsetdimscale
DFAN	text annotation records	DFANputlabel, DFANgetdesc
Multi-file Application Layer Interfaces (new)		
SD	multi-file scientific dataset	SDstart, SDcreate, SDdiminfo
NC	netCDF interface	nccreate, ncopen, ncvardef
VS	Vdata interface	VSattach, VSfdefine, VSgetid
VSQ	Vdata query	VSQuerycount, VSQueryname
VF	Vdata fields inquiry	VFfieldsize, VFfieldname
V	Access, Specify, Inquire Vgroups	Vattach, Vstart, Vsetname, Vgetid
VH	Simple Vdata, Vgroup creation	VHmakegroup, VHstoredata
GR	Generic raster image interface	Grcreate, GRreadimage

Note the HDF APIs are divided into two categories: multifile interfaces (new) and single-file interfaces (old). The multifile interfaces are those that provide simultaneous access to several HDF files from within an application, which is an important feature that the single-file interfaces do not support. The new interfaces should be used at all times since they are an improvement over the old interfaces. The old interfaces remain simply for backward compatibility.

The most important of these interfaces include the scientific dataset interfaces, the vdata interfaces, and the Vgroup interfaces. They are described below.

2.2.1 The Scientific Dataset Interfaces (DFSD, SD)

There are two HDF interfaces that support multidimensional arrays. The older one is a single-file interface (known as the 'DFSD' interface, because all the associated subroutines start with 'DFSD') which permits access to only one file at a time. The newer one is a multi-file interface (known as the 'SD' interface), which permits simultaneous access to more than one file. We recommend the use of the newer multi-file interface for ECS users.

The single-file DFSD interface provides a collection of routines for reading and writing an array of arbitrary rank and type. The array, along with its associated information, is known as a *Scientific Data Set*, or SDS for short.

The multi-file SD interface allows concurrent operations on more than one file and data object. The interface is also interoperable with the netCDF interface. By interoperable, we mean the netCDF interface as implemented in HDF can read both netCDF files and HDF files. Like the single-file DFSD interface, a data object written with the multi-file SD interface includes the normal SDS data element, `DFTAG_SD`. This data element specifies a unique tag-reference for the data object. However, the multi-file SD interface also includes many additional attributes that are not part of the single-file interface.

In either interface, the multi-dimensional array in the SDS can contain 8-, 16-, 32-, or 64-bit signed or unsigned integers or 32- or 64-bit floating point numbers. (64-bit representations are supported only if the particular platform supports 64-bit operations.) The SDS can also contain the following attributes:

- Coordinate system: Identifies which coordinate system to use when displaying data.

- Formatting: Specifies the format for displaying values for data and attributes.

- Label: Contains a name for each independent variable and the data.

- Ranges: Stores the maximum and minimum values in the data, as supplied by the data producer.

- Calibration: Describes the scale to use along each axis. E.g. specifies a linear scaling for axis values.

- Fill Value: Defines the value to fill areas of no data or bad data at the users discretion.

- Units: Identifies the unit associated with each dimension and the data.

- Section 2.3.5 of this document shows examples of SDS data objects in HDF files. The multi-file SD interface subroutines are divided into the following categories:

- Access routines that intialize and close the SD interface.

- Create, read, and write SDS routines for defining and reading array dimensions, rank, number type, fill value, data range, calibration information, and data values.

- Dimension attribute routines for defining and reading SDS attributes such as dimension name, format, unit, label, or scales. All attributes are optional.

- General attribute routines for managing local attributes (attributes assigned to a data object) and global attributes (attributes assigned to a file). Predefined local attributes include the coordinate system, format, labels, max/min values, scales, and units.

In EOS, we have given a strong emphasis to the multifile SD interface.

2.2.2 The Vdata Interfaces (VS, VSQ, VF)

The HDF Vdata model, which includes the VS, VSQ, and VF interfaces, makes it easy to store tables of data in HDF files. Each table consists of a series of records, each of which contains a series of fields. By field, we mean a grouping of data elements within a record. Each field can

support its own number type. However, every record in a Vdata must contain the same fields. Valid number types include 8-, 16-, 32-, 64-bit signed and unsigned integers, 32- and 64-bit floating point numbers, and ASCII characters.

Vdata tables use three kinds of identifying information: a *name*, a *class*, and a set of individual field names. A Vdata name is a label that typically describes the origin and contents of a table. A Vdata class typically identifies the meaning of data. Finally, Vdata field names identify the individual fields that make up a record.

Vdata Name	Temperature Table		
Vdata Class	Class		
	Field #1	*Field #2*	*Field #3*
Field Names	Latitude	Longitude	Temperature
Record #1	8-bit Int	8-bit Int	32-bit Float
Record #2	8-bit Int	8-bit Int	32-bit Float
Record #3	8-bit Int	8-bit Int	32-bit Float

Figure 2-3. A Typical Vdata

There are three Vdata interfaces. The VS Interface provides a collection of routines for reading and writing tables. The VS functions are divided into five categories:

- Access routines which initialize and terminate access to a Vdata, and seek record position in a Vdata.

- File inquiry routines provide information on how a Vdata is stored in a file.

- Read/Write routine which define new Vdata fields, assign names and classes, and initialize read and write permissions.

- Vdata inquiry routines which check if a Vdata exists, and return a Vdata's class name or field names, its size, and whether it exists as a lone entity. By lone entry, we mean the it is a stand-alone object and not part of another object in the file.

- Read/Write routines which retrieve and store Vdata records in HDF files.

The VSQ and VQ interfaces are described in the HDF Users Guides.

2.2.3 Vgroup interface (V)

The Vgroup interface provides a collection of routines for reading and writing groupings of HDF data objects in a particular HDF file. Each Vgroup may contain any number of other HDF data objects, even other Vgroups. In addition to its members, a Vgroup may also be given a Vgroup name and Vgroup class. The Vgroup name must be unique within a particular grouping of

objects that includes other Vgroups. Names don't have to be unique across stand-alone groupings of objects. AN HDF file can consist of many Vgroups, plus stand-alone Vdatas and SDS's.

Every function on a Vgroup begins with the prefix V. The Vgroup functions are divided into five categories:

- Access/Create routines that begin and end access to Vgroups.

- Manipulation routines modify Vgroups' characteristics, and add and delete Vgroups' members.

- Vgroup inquiry routines obtain information about Vgroups.

- Member inquiry routines obtain information about members of Vgroups

- Attributes routines provide information about Vgroups' attributes.

2.3   Overview of the HDF Disk Format

It is easier to understand the HDF software if you first have a general understanding of HDF disk formats. An HDF data file consists of three main things: a magic number, a directory that points to data elements, and the data elements themselves. We start by describing the HDF file header.

2.3.1 The HDF File Header

The first component of an HDF file is the file header, which takes up the first four bytes of the HDF file. Specifically, it consists of four one-byte values that are ASCII representations of control characters: the first is a control-N, the second is a control-C, the third is a control-S and the fourth is a control-A (^N^C^S^A). The directory begins immediately after the file header.

Note that, on some machines, the order of bytes in the file header might be swapped when the header is written to an HDF file, causing these characters to be written in little-endian order. To maintain the portability of HDF file header data when developing software for such machines, this byte swapping must be counteracted by ensuring the characters are read and written in the desired order.

The HDF definition of the term file header is in contrast to the more traditional view of a file header as a block of metadata at the beginning of a file. To avoid confusion over this point, we will not use the term 'file header' in this document.

2.3.2 The HDF Directory

An HDF directory, called a *DD list*, is usually broken up into a series of components known as *DD blocks.* Each DD block contains just three things: the number of directory entries (known as *Data Descriptors*, or *DD*s) in that block, the byte location of the next DD block (or 0 if this is the last DD block), and then the actual directory entries.

The first two bytes of each DD block contain a count of the number of entries in the block. The next four bytes contain the byte location of the next DD block. The actual DDs follow

immediately. Every DD is always exactly 12 bytes long. The layout of an HDF file with a single DD block containing two DDs is shown below.

Table 2-2. Organization of the beginning of an HDF file

Location	Value	Comment
0 to 3	^N^C^S^A	Unique HDF Number (^N = control-N, etc.)
4 to 5	2	Number of DDs in DD Block
6 to 9	0000	Location of next DD Block (none here)
10 to 21	---	Data Descriptor #1
22 to 33	---	Data Descriptor #2

Note that the location of the next DD Block entry is given in *bytes from the beginning of the HDF file*. In fact, *all* locations within the HDF file are given in terms of bytes from the beginning of the file. This contrasts with some other disk formats which locate items solely by record numbers. There are no records, as such, in HDF. Since all locations are given in 32-bit *signed* integers, the maximum size of a self-contained HDF file or of a single data element is therefore around 2×10^9 bytes, or 2 GigaBytes.

2.3.3 HDF Data Descriptors

The single most important HDF concept is that of the Data Descriptors. *Every single data element (e.g., image, array, annotation,Vgroup, etc.) in the HDF file has an associated Data Descriptor (DD) in the DD list*. We will keep returning to this statement throughout the discussion.

Every DD is of fixed length with four fields: a *Tag* field, which defines the data element *type*, a *Reference Number* field (*Ref*), which is unique for every data element with the same Tag, an *Offset* field, which gives the location of the data element in bytes from the start of the file, and a *Length* field, which gives the length of the data element in bytes.

Table 2-3. HDF data descriptor layout

Data Descriptor											
1	2	3	4	5	6	7	8	9	10	11	12
Tag		Ref		Offset				Length			

2.3.3.1 HDF Tags

The Tag field is defined as a 16-bit unsigned integer, which means there are 65535 possible types of data elements. (0 is not a legal tag number.) The possible tag values are divided into three ranges, as shown below.

Table 2-4. Ranges of possible tag values

Tag Value Range	Comment
00001 to 32767	Assigned by NCSA
32768 to 64999	Defined by user
65000 to 65535	Reserved for future use

User-defined and NCSA-defined data element types can be freely intermixed in the same file. NCSA has defined *utility tags* for general data descriptions, *raster image* tags for descriptions of pseudocolor and color images, *scientific dataset* (SDS) tags for describing multi-dimensional arrays of numbers, the *Vdata* tag for defining tables of values, and the *Vgroup* tag for grouping data elements. A selection of these tags is shown below.

Table 2-5. A selection of NCSA defined HDF tag types

Tag Name	Tag #	Comments
		Utility Tags
DFTAG_RLE	011	Specifies the Run Length Encoding used for image
DFTAG_TID	102	Tag Identifier: text string of user defined tag
DFTAG_DIL	104	Data Identifier Label: used for titles of elements
DFTAG_DIA	105	Data ID Annotation: lengthy text annotation block
DFTAG_NT	106	Number Type: values are float, integer, text, etc.
DFTAG_MT	107	Machine Type: specifies IEEE or local computer type
		Raster Image Tags
DFTAG_ID	300	Image Dimension: gives X and Y size of assoc. image
DFTAG_LUT	301	Lookup Table: color lookup table for assoc. image
DFTAG_RI	302	Raster Image: points to actual image data
DFTAG_RIG	306	Raster Image Group: lists all DDs assoc. with image
DFTAG_LD	307	LUT Dimension: size of color lookup table
DFTAG_CFM	311	Color Format: grayscale, Pseudocolor, RGB, HSI, etc.
		Scientific Dataset Tags
DFTAG_SDD	701	SDS Dimension: dimension sizes of scientific dataset
DFTAG_SD	702	Scientific Data: points to actual scientific dataset
DFTAG_SDS	703	Scientific Data Scales: Arrays for X,Y,Z locations
DFTAG_SDL	704	SD Labels: text describing data and dimensions
DFTAG_SDU	705	SD Units: text with units for data and dimensions
DFTAG_SDF	706	SD Format: text with format code for displaying data
DFTAG_SDM	707	SD Max/Min: minimum/maximum valid values for data
DFTAG_SDC	708	SD Coordinate System: text string defining CS
DFTAG_NDG	720	Numeric Data Group: lists all DDs assoc. with SD
		Vgroup/Vdata Tags
DFTAG_VG	1965	Vgroup: provides general purpose grouping of DDs
DFTAG_VH	1962	Defines the structure of a Vdata data element
DFTAG_VS	1963	Points to Vdata data element using DFTAG_VH format

2.3.3.2 HDF Reference Numbers

If you store two raster images in an HDF file, you will produce two DDs with the same tag number (`DFTAG_RI`). But this creates a problem: how does one know which `DFTAG_RI` is which?

The answer is that a *Reference number* is assigned, usually by the HDF library, to each data object as it is written to the file. The library keeps track of the reference numbers that have been used in the file and guarantees that there will never be two data objects with the same tag *and* reference number in the same file. It *is* allowable for two objects to carry the same tag (such as our two raster images) or for two objects with different tags to carry the same reference number (such as a raster image and an SDS).

Although many DDs can have the same Reference number, the HDF standard demands that there is only one instance of a particular Reference number for a particular *Tag*. This means that every *Tag/Ref* combination is *unique* within a single HDF file. A *Tag/Ref* pair can then be used as a unique ID (known in HDF terms as a '*data identifier*') to unambiguously identify particular data elements.

It is important to note at this point that, with only one exception that we will discuss in a later section, one must take care not to infer meaning on the library's choice of reference numbers. It will often be the case that some number of data objects in a file will have the same reference number. The only information that can be gleaned from this similarity in reference numbers is that the library is miserly in doling out new reference numbers. It reuses reference numbers in all cases where it can easily ascertain it to be safe to do so. Coincidentally, such cases often occur when writing out groups of data objects such as Raster Image Groups and Scientific Data Sets. However, this behavior is not an official part of the specification of HDF and it should never be relied upon.

Although it would be tempting to use Reference numbers to group data elements, that method of associating data objects is lacking in that it does not support sharing of data objects between groups. Instead HDF uses *Groups* to define relationships.

2.3.4 HDF Groups

The HDF libraries support explicit grouping of data elements using one of three group tags. The Raster Image Group Tag (DFTAG_RIG) is used to group all data objects associated with a particular *raster image*. The Numeric Data Group Tag (DFTAG_NDG) is used to group all data objects associated with a particular *scientific dataset*. The Vgroup tag (DFTAG_VG) supports a generalized grouping of *all* types of data objects, even other Vgroups.

Table 2-6. HDF group tags

Tag Name	Tag #	Comments
DFTAG_RIG	306	Raster Image Group: lists all DDs assoc. with image
DFTAG_NDG	720	Numeric Data Group: lists all DDs assoc. with SDS
DFTAG_VG	1965	Vgroup: provides general purpose grouping of DDs

The data contained within these groups is a list of *Data Identifiers* associated with the group. Since the groups do *not* contain the offset values of the data elements, a DD is *still* needed in the DD list with this information. So again, *all* data elements must have a DD in the DD list. In addition, *most* will *also* have their *Data Identifiers* listed in some group.

Both *Raster Image Groups* (*RIGs) and Numeric Data Groups (NDGs)* consist of nothing but a list of *Data Identifiers*, each four bytes long (*Vgroups* are more complicated; see below). The number of *Data Identifiers* in a RIG or NDG is just the length field divided by four.

In the table below, we show an HDF example file that uses Raster Image Groups (RIGs) to define two images of identical size. Note how, in this example, both raster image groups use the same image size data element.

Table 2-7. Organization of an HDF file with two raster image groups

Location	Length	Value				Comment
0 to 3	4				^N^C^S^A	Unique HDF Number
4 to 5	2				5	Number of DDs in Block
6 to 9	4				0000	Loc. of next DD Block
		Tag	*Ref*	*Offset*	*Length*	
10 to 21	12	300	001	70	4	DD #1 (Image Size)
22 to 33	12	302	001	90	60000	DD #2 (Ptr to Image#1)
34 to 45	12	302	002	60090	60000	DD #3 (Ptr to Image#2)
46 to 57	12	306	005	74	8	Ptr to 1st raster group
58 to 69	12	306	006	82	8	Ptr to 2nd raster group
		X Size		*Y Size*		
70 to 73	4	300		200		Size of both images
		Tag/Ref#1		*Tag/Ref#2*		
74 to 81	8	300/001		302/001		First raster group
82 to 89	8	300/001		302/002		Second raster group
		Image Data				
90 to 60089	60,000	02h………23h				Data for first image
60090 to 120089	60,000	7Ah………19h				Data for second Image

RIGs can contain only raster image and related tags, and all the *Data Identifiers* in a particular group must relate to a single image. The same holds true for an NDG; only numeric data group tags (and some utility tags such as number type) are allowed, and they must all relate to a single numeric array.

Vgroups do not have this limitation: they can contain any collection of *Data Identifiers*, including a *Vgroup* Data Identifier. This last feature means that you can construct a directory structure inside a single HDF data file, with directories containing data elements and/or sub directories.

There is, however, a major difference between Vgroup organization and disk directories. In an HDF file, all groups including Vgroups contain *Data Identifiers only*, not full DDs. This means that a DD must still be in the main DD list for *every single data element* in the HDF file, *regardless* of how deeply it is referenced in a Vgroup hierarchy.

For example, consider the HDF file shown below. (We have simplified a bit in this table, because the structure of Vgroup data elements is a bit more complicated than what is shown.)

Table 2-8. Organization of an HDF file with two Vgroups

Location	Length	Value				Comment
0 to 3	4	^N^C^S^A				Unique HDF number
4 to 5	2	3				Number of DDs in block
6 to 9	4	0000				Loc. of next DD block
		Tag	*Ref*	*Offset*	*Lth*	
10 to 21	12	300	001	46	4	DD #1 (Image size)
22 to 33	12	1965	001	50	22	DD #2 (Vgroup #1)
34 to 45	12	1965	002	72	22	DD #3 (Vgroup #2)
		X Size		*Y Size*		
46 to 49	4	300		200		Image size
		Name		*Tag/Ref*		
50 to 71	22	'Vone'		1965/002		First VGroup
72 to 93	22	'Vtwo'		300/001		Second VGroup

How this file is organized may not be immediately obvious. The graphic below may help. Here we show that the main DD list contains the 'Vone' Vgroup, which contains a single *Data Identifier*, a reference to the 'Vtwo' Vgroup. This Vgroup in turn contains a single *Data Identifier* reference to the Image Size entry.
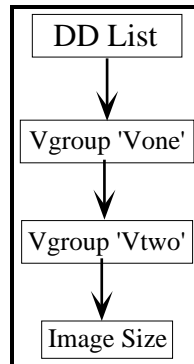
Figure 2-4. Organizational Levels for Dataset Shown in Table 2-8

The confusion is that every *Data Identifier*, even the one for 'Vtwo', is *also* listed in the DD list. How does someone know which Vgroup is at what level? The answer is that there is no unique way.

You can use the HDF utility 'Vshow' to display a list of the contents of each directory. But what should the utility do if 'Vone' contained 'Vtwo', *and* 'Vtwo' contained 'Vone'? Can a directory contain a directory that contains the original directory? Again, it is the responsibility of the data producer to ensure that the directory structure makes sense.

2.3.5 HDF Scientific Datasets

The most important data objects in HDF files are known as *Scientific Datasets* (SDS). A scientific dataset is a multidimensional array of numbers. The array can have any dimensionality up to 32767, and can consist of floating point or integer values. Since the release of HDF version 3.3 there have been two separate interfaces for SDSs: the 'DFSD' interface (old) and the 'SD' interface (new). Each interface corresponds to a different physical organization of the SDS and, therefore, has slightly different capabilities. The most significant new features of the 'SD' interface are its ability to concurrently deal with multiple SDSs in multiple files, its ability to put arbitrary attributes on SDSs, and its compatibility with the netCDF interface.

Although the 'SD' interface is the preferred interface for reading and writing SDSs in ECS, its physical implementation is a bit too complex to describe here. Therefore, we will discuss the physical implementation of the older 'DFSD' interface in order to illustrate the basic organizational concepts of the HDF library, then briefly touch on the differences between the two implementations.

2.3.5.1 The 'DFSD' Interface

An SDS, whether created by the 'DFSD' or 'SD' interface, consists of several data objects that are associated with the SDS. These data objects are attributes, numerical scales, the actual data, and so on. In the 'DFSD' interface, all the DDs of the data objects associated with a particular SDS are listed in a *numeric data group* (DFTAG_NDG). In some old HDF files with 32-bit floating point SDSs, you may run across an SDS that uses the roughly equivalent *scientific data group* (DFTAG_SDG) to collect its members. In either case, only scientific data set tags (Table 2-5) and a few utility tags are allowed in a numeric data group. An example of the data contained in an SDS is shown below.

Neutron Star Accretion Simulation				
		Time (sec)		
	1.67	1.68	1.69	1.70
32.11	0.5872	0.5872	0.5872	0.5872
31.96	0.5872	0.5872	0.5872	0.5872
31.81	0.5872	0.5872	0.5872	0.5872
31.66	2.4226	2.3604	2.4282	2.4117
31.51	1.9976	1.9957	1.9963	2.0018
31.36	1.8119	1.8102	1.8107	1.8153
31.20	1.6768	1.6726	1.6745	1.6796
31.05	1.5762	1.5736	1.5742	1.5802
30.90	1.4981	1.4942	1.4956	1.4997
30.75	1.4356	1.4319	1.4319	1.4352

Radius (km)

Density (gm/cm^3)

Figure 2-5. Example SDS

The data identifiers stored in the NDG for this SDS are shown below, along with the information pointed to by the data identifiers. In this list we ignore the actual format of the data identifiers, and concentrate instead on the information contained that they point to. Note also that the *Ref*, *Offset*, and *Length* fields for each DD are not shown.

Table 2-9. Data Objects for the SDS shown in Figure 2-4

Tag Name	Value(s)			Comments
	Data	*Dim1*	*Dim2*	
DFTAG_DIL	'Neutron Star Accretion Simulation'			Title for SDS Array
DFTAG_SDD	2 Dimensions			SD Dimension record
		4	10	Size of dimensions
	Float32	Float32	Float32	Number types
DFTAG_SDS		1.67…1.70	30.75…32.11	SD Numerical scales
DFTAG_SDL	'Density'	'Time'	'Radius'	SD Labels
DFTAG_SDU	'gm/cm^3'	'sec'	'km'	SD Units
DFTAG_SDF	F7.4	F5.2	F5.2	SD Numerical Formats
DFTAG_SD	'0.5872,0.5872,……,1.4319,1.4352'			Actual data

The `DFTAG_SDD` tag says that the data set consists of two dimensions, sized four and ten elements, respectively, where the data and both scales consist of floating-point numbers. The numerical scales are set by `DFTAG_SDS`, with a four-element array for the first dimension and a ten-element array for the second dimension. As mentioned before, all of these data identifier entries are listed both in the main DD list directory, and in the NDG that defines the SDS.

2.3.5.2 The 'SD' Interface

The basic ideas behind the SD version of the SDS are the same as those in the DFSD SDS, but some changes were needed in order to achieve the major design goal: compatibility with netCDF. This driver gave rise to three distinct differences between the two implementations.

First, rather than using a specialized tag such as `DFTAG_NDG` to group its elements, the SD interface uses a general purpose Vgroup. Using a Vgroup allows for more flexibility in the organization of the SDS without a complete redesign. Second, the handling of dimensions has changed considerably, in that they can now have their own attributes. Third, the SD interface allows the use of user-defined attributes, not just the ones that are pre-defined.

Below is a list of the different styles of SDS, in order of their adoption into the HDF library.

Scientific Data Groups (SDG)—Originally, all SDSs were written in SDG groups. An SDG group (tag value = `DFTAG_SDG`) can contain any of the tags that are currently used for NDGs. The only difference between SDGs and NDGs is that SDGs only support 32-bit floating point arrays, whereas NDGs can support several types of floating point and integer number types.

SDGs are considered obsolete, although the HDF libraries will write out *both* an SDG *and* a NDG for the same array, if that array is made up of 32-bit floating point values. Application programs that are linked with versions of HDF before 3.2 will only recognize SDGs as SDSs. We will not discuss SDGs further, since they will not be used in ECS.

Numeric Data Groups (NDGs)—What we have just described in detail is the disk format for NDG groups (tag value = `DFTAG_NDG`). The NDG tag was new with HDF version 3.2. Application programs linked with version 3.2 of HDF will recognize SDGs and NDGs as SDSs. They are referred to in the latest documentation (3.3) as the "Old Style" HDF subroutines that deal solely with NDGs and SDGs are prefixed with "`DFSD`".

Multifile SDSs—The recommended interface for SDSs in the current version of HDF (4.1) is known as the "Multifile SDS" interface, or the '`SD`' interface (after the prefix of the subroutine calls). The Multifile SDS interface defines a set of conventions and a set of library subroutines that uses Vgroups, Vdatas, and many of the parts of the NDG to create structures that are compatible with netCDF data structures.

The package is called Multifile SDS because the new subroutines allow several HDF files to remain open simultaneously, something not possible with the earlier libraries. Another significant advantage of the interface is the ability to assign arbitrary attributes to SDSs. In the documentation they are also referred to as "New Style" SDSs, or (confusingly) just SDSs. The multifile interface will recognize all older forms of the SDS. The Multifile SDS interface is used exclusively for EOS data.

2.3.6 HDF Vdatas

HDF also supports the storage of tables that are organized as named columns known as *Vdatas*. Storage of a Vdata table requires the use of two tags: `DFTAG_VH` for defining and naming the columns of values; and `DFTAG_VS` for pointing to the actual Vdata data itself. There is no explicit grouping needed for Vdatas, although they can be included in a Vgroup (and usually are).

For example, consider the table of values shown below:

Table 2-10. Example Table of Values

ID. No	Flux	Name
1	2.34	CygX1
2	-89.43	HerX1
3	0.0023	CygX3
4	1.115	Vela

The tags needed to define this table as a Vdata data object are shown below. Again, we focus on the information pointed to by the tags and ignore the actual binary formats. In particular, we have assigned names to the various fields defined in the data record pointed to by `DFTAG_VH`. These field names are also used in the HDF documentation.

Table 2-11. Vdata tags for dataset shown in Table 2-10

Tag Name	Field Name	Values			Comments
		Col1	*Col2*	*Col3*	
DFTAG_VH	<nvert>	4			Number of records
	<isize>	11			Row width in bytes
	<nfields>	3			Number of fields
	<type>	int16	float32	char[5]	Field number types
	<isize>	2	4	5	Field size in bytes
	<offset>	0	2	6	Byte offset of field
	<fldnmlen>	5	4	4	Length of field Name
	<fldnm>	'ID.No'	'Flux'	'Name'	Field names
	<namelen>	19			Length of Vdata name
	<name>	'Vdata Example Table'			Vdata name
DFTAG_VS		1,2.34,'CygX1',…,'Vela'			Actual data

In this example, the information in the DFTAG_VH data element defines a Vdata with three *fields* (columns) and four Vdata *records* (rows). The values in these fields are defined as short (2-byte) integer, 4-byte floating-point, and five-character ASCII text, respectively. In addition, each field has an ASCII text name. Note, by the way, how the entire Vdata definition can be named (<name>). This is unusual, in that most HDF data objects require a separate tag (DFTAG_DIL) to get a name.

The actual data pointed to by the DFTAG_VS tag is stored, packed as tightly as possible. The total width of every record is therefore $2 + 4 + 5 = 11$ bytes. The four records of this table take up 11 ∞ 4 = 44 bytes. This packing is shown below. Here the DFTAG_VS record associates itself with a DFTAG_VH record by having the same *Ref* number. This is the only case where the HDF libraries use the *Ref* numbers explicitly to associate two data elements.

Table 2-12. Disk layout of Vdata described in Table 2-11

	Byte Position										
	1	2	3	4	5	6	7	8	9	10	11
	Integer		*Floating Point*				*Text String*				
Record #0	1		2.34				C	y	g	X	1
Record #1	2		-89.43				H	e	r	X	1
Record #2	3		0.0023				C	y	g	X	3
Record #3	4		1.115				V	e	l	a	

2.3.7 HDF Extended Tags

The organizational features of HDF are so powerful that it is usually possible to store a complete data product granule in a single HDF file. There are, however, a couple of problems with very large single file data products.

The first problem, which we have already mentioned, is that HDF files are limited to 2 Gigabytes in size. The second problem is that HDF data elements were not initially designed to be appendable. Normally, data elements are required to be in contiguous storage and they are packed right next to each other. Therefore, to make a data element bigger, it needs to be copied to the end of the file where there is room to grow, leaving a gap in the file.

To get around these limitations, NCSA has defined *extended tags*. An extended tag allows a data element to be spread among multiple locations in the HDF file or even in a completely separate file. A data element corresponding to any NCSA defined tag value can be converted to an extended tag, although this functionality is currently only supplied for SDSs and Vdatas.

An extended tag DD does *not* point directly to the data, as the normal tags do. It instead points to a data object *defining where the data is and how it is stored*. This data object *may* point to the beginning of a linked list of data blocks that contain the entire data record. This way, a data record can be lengthened just by adding an additional data block; the entire HDF file does *not* need to be rewritten.

Alternatively, the extended tag record could define the data element as being stored in an *external element* in another disk file. This feature (which is also found in CDF), allows a user to get around the two-gigabyte limitation on total file size. It may also make a large HDF 'file' easier to handle. Several 100-megabyte files are sometimes easier to handle than a single gigabyte file.

2.4 HDF Examples

The following sections contain sample C code for writing and reading a selection of basic HDF data objects. For now, we have kept to some very simple code that, for the most part, has been copied from NCSA's HDF documentation.

2.4.1 8-bit Raster Image Output

The C program below writes an 8-bit raster image to the file "example.hdf". This program takes no input.

```
#include "hdf.h"
#define WIDTH     5
#define HEIGHT    6

main(int argc, char *argv[])
{
    /*  Initialize the image array  */
    static uint8 raster_data[HEIGHT][WIDTH] =
                { 1,  2,  3,  4,  5,
                  6,  7,  8,  9, 10,
                 11, 12, 13, 14, 15,
                 16, 17, 18, 19, 20,
                 21, 22, 23, 24, 25,
                 26, 27, 28, 29, 30 };

    /*  Write the 8-bit raster image to the file  */
    DFR8addimage("example.hdf", raster_data, WIDTH, HEIGHT, 0);
}
```

2.4.2 8-bit Raster Image with Palette Output

The program below writes an 8-bit raster image and its associated palette to the file "example.hdf". This program takes no input.

```
#include "hdf.h"
#define WIDTH    5
#define HEIGHT   6

main(int argc, char *argv[])
{
    uint8 palette_data[768];
    intn i;

    /*  Initialize the image array  */
    static uint8 raster_data[HEIGHT][WIDTH] =
                { 1,  2,  3,  4,  5,
                  6,  7,  8,  9, 10,
                 11, 12, 13, 14, 15,
                 16, 17, 18, 19, 20,
                 21, 22, 23, 24, 25,
                 26, 27, 28, 29, 30 };

    /*  Initialize the palette to standard linear grayscale  */
    for (i=0; i<256; i++) {
    palette_data[i*3] = i;
    palette_data[i*3+1] = i;
    palette_data[i*3+2] = i;
    }

    /*  Associate the palette with the image  */
    DFR8setpalette(palette_data);

    /*  Write the 8-bit raster image to the file  */
    DFR8addimage("example.hdf", raster_data, WIDTH, HEIGHT, 0);
}
```

2.4.3 8-bit Raster Image with Palette Input

The program below reads an 8-bit raster image and its associated palette from the file "example.hdf" created by the program in the section above. This program produces no output.

```
#include "hdf.h"
#define WIDTH    5
#define HEIGHT   6

main(int argc, char *argv[])
{
    uint8 raster_data[HEIGHT][WIDTH], palette_data[768];
    intn haspal;
    int32 width, height;

    /*  Get dimensions and check for palette  */
    DFR8getdims("example.hdf", &width, &height, &haspal);

    /*  Read the 8-bit raster image and palette from the file  */
    if ((width == WIDTH) && (height == HEIGHT) && (haspal == 1))
        DFR8getimage("example.hdf", (uint8 *) raster_data, width,
                    height, palette_data);
}
}
```

2.4.6 Scientific Data Set Output

The programs in this section make use of the new SDS interface introduced with HDF 3.3. We recommend that people use this interface rather than the older interface. The program below writes a Scientific Data Set to the file "example.hdf". This program takes no input.

```c
#include "hdf.h"
#include "mfhdf.h"
#define LENGTH   3
#define HEIGHT   2
#define WIDTH    5
main(int argc, char *argv[])
{
    /*  Initialize the image array  */
    static float64 scien_data[LENGTH][HEIGHT][WIDTH] =
               { 1.,   2.,   3.,   4.,   5.,
                 6.,   7.,   8.,   9., 10.,
                11., 12., 13., 14., 15.,
                16., 17., 18., 19., 20.,
                21., 22., 23., 24., 25.,
                26., 27., 28., 29., 30. };
    int32 dims[3]        = {LENGTH, HEIGHT, WIDTH};
    int16 scale0[LENGTH] = {2, 4, 6};
    int32 scale1[HEIGHT] = {1234567, 2345678};
    float32 scale2[WIDTH] = {2.2, 4.4, 6.6, 8.8, 11.0};
    float64 avg          = 15.0;
    int32 start[3]       = {0, 0, 0};
    int32 fid, sdid, dimid0, dimid1, dimid2;

    /*  Open file and initialize SD interface  */
    fid = SDstart("example.hdf", DFACC_CREATE);

    /*  Create named data set  */
    sdid = SDcreate(fid, "Sample Data Set", DFNT_FLOAT64, 3, dims);

    /*  Set up dimension zero  */
    dimid0 = SDgetdimid(sdid, 0);
    SDsetdimname(dimid0, "Dimension 0");
    SDsetdimstrs(dimid0, "The zeroth dimension", "mm", "2d");
    SDsetdimscale(dimid0, LENGTH, DFNT_INT16, (VOIDP)scale0);

    /*  Set up dimension one  */
    dimid1 = SDgetdimid(sdid, 1);
    SDsetdimname(dimid1, "Dimension 1");
    SDsetdimstrs(dimid1, "The first dimension", "cm", "8d");
    SDsetdimscale(dimid1, HEIGHT, DFNT_INT32, (VOIDP)scale1);

    /*  Set up dimension two  */
    dimid2 = SDgetdimid(sdid, 2);
    SDsetdimname(dimid2, "Dimension 2");
    SDsetdimstrs(dimid2, "The second dimension", "m", "4.1f");
    SDsetdimscale(dimid2, WIDTH, DFNT_FLOAT32, (VOIDP)scale2);

    /*  Write the data array to the data set  */
    SDwritedata(sdid, start, NULL, dims, (char *)scien_data);

    /*  Add one local attribute  */
   SDsetattr(sdid, "Average", DFNT_FLOAT64, 1, (char *)&avg);

    /*  Add one global attribute  */
    SDsetattr(fid, "Date", DFNT_CHAR8, 9, "10/29/93");

    /*  Close the data set, file, and interface  */
    SDendaccess(sdid);
    SDend(fid);
}
```

2.4.7 Scientific Data Set Input

The program below reads a Scientific Data Set from the file "example.hdf" created by the program in the section above. This program produces no output.

```c
#include <string.h>
#include "hdf.h"
#include "mfhdf.h"

#define MAXRANK   3
#define LENGTH    3
#define HEIGHT    2
#define WIDTH     5
#define DATESIZE  9

main(int argc, char *argv[])
{
    float64 scien_data[LENGTH][HEIGHT][WIDTH];
    int32 dims[MAXRANK];
    int16 scale0[LENGTH];
    int32 scale1[HEIGHT];
    float32 scale2[WIDTH];
    float64 avg;
    int32 start[MAXRANK] = {0, 0, 0};
    int32 fid, sdid, dimid;
    int32 i, index, rank, nattrs, ndatasets, nglobals;
    int32 nt, count, status;
    intn size;
    char name[80], date[80];

    /*  Open file and initialize SD interface  */
    fid = SDstart("example.hdf", DFACC_RDONLY);
    status = SDfileinfo(fid, &ndatasets, &nglobals);

    /*  Read global attribute  */
    if (nglobals == 1) {
        status = SDattrinfo(fid, 0, name, &nt, &size);
        if ((strcmp(name, "Date")) && (nt == DFNT_CHAR8) &&
            (size == DATESIZE))
            SDreadattr(fid, 0, date);
    }

    /*  Open first data set  */
    index = SDnametoindex(fid, "Sample Data Set");
    sdid = SDselect(fid, index);
    SDgetinfo(sdid, name, &rank, dims, &nt, &nattrs);

    /*  Read in data if everything looks okay  */
    if ((rank == MAXRANK) && (dims[0] == LENGTH) && (dims[1] == HEIGHT)
    && (dims[2] == WIDTH) && (nt == DFNT_FLOAT64))
    SDreaddata(sdid, start, NULL, dims, scien_data);

    /*  Read local attribute  */
    status = SDattrinfo(sdid, 0, name, &nt, &size);
    if ((strcmp(name, "Average")) && (nt == DFNT_FLOAT64) &&
        (size == 1))
        SDreadattr(sdid, 0, &avg);

    /*  Read dimensions  */
    dimid = SDgetdimid(sdid, 0);
    SDdiminfo(dimid, name, &count, &nt, &nattrs);
    if ((nt == DFNT_INT16) && (count == LENGTH))
        SDgetdimscale(dimid, scale0);

    dimid = SDgetdimid(sdid, 1);
    SDdiminfo(dimid, name, &count, &nt, &nattrs);
    if ((nt == DFNT_INT32) && (count == HEIGHT))
```

```
        SDgetdimscale(dimid, scale1);

    dimid = SDgetdimid(sdid, 2);
    SDdiminfo(dimid, name, &count, &nt, &nattrs);
    if ((nt == DFNT_FLOAT32) && (count == WIDTH))
        SDgetdimscale(dimid, scale2);

    /*  Close the data set, file, and interface  */
    SDendaccess(sdid);
    SDend(fid);
}
```

3. HDF-EOS V2

3.1   Introduction

HDF as introduced in Section 2, has been extended by the ECS Project to focus conventions for writing EOS data products. These extensions are called HDF-EOS. In this Section, we present a brief introduction to the file format of HDF-EOS. A detailed discussion, as well as an operational description of HDF-EOS can be found in HDF-EOS Users Guide for the ECS Project (Volume 1 and Volume 2).

3.2   HDF and HDF-EOS File Formats

3.2.1 Overview

Most of the NCSA defined datatypes map well to EOS datatypes. Examples include raster images, multi-dimensional arrays, and text blocks. There are other EOS datatypes, however, that do not map directly to NCSA datatypes, particularly in the case of geolocated datatypes. Examples include projected grids, satellite swaths, and field campaign or point data.

An HDF file consists of a directory, and records pointed to by that directory. Each directory entry consists of fields for the record type (TAG), a unique (for each TAG) ID number (REF), and a location and size of the record pointed to. All locations in the HDF file are in byte locations from the beginning of the file. All record sizes are also specified in bytes.

Supported record types include images, multidimensional arrays, text and tables (known in HDF as Vdatas). One record type, known as Vgroup, lets the user group a series of records into a larger structure, similar to disk directories.

To bridge the gap between the needs of EOS data products and the capabilities of HDF, the ECS Project has developed extensions of HDF, which standardize the conventions for writing HDF files, and are called HDF-EOS. These extensions facilitate the creation of Grid, Point and Swath data structures. These structures are composed of native HDF objects and are therefore objects themselves. In the text below, Grid, Point and Swath structures are described in more detail.

The software interface for the HDF-EOS implementation is very similar to the HDF interface. The HDF-EOS interface is used to access the Grid, Point and Swath data structures created by the HDF-EOS library. The plain HDF interface is not used to access Grid, Point and Swath structures. See HDF-EOS Users Guide for the ECS Project and references.

The *Point* interface is designed to support data that has associated geolocation information, but is not organized in any well-defined spatial or temporal way. The *Swath* interface is tailored to support time-ordered data such as satellite swaths (which consist of a time-ordered series of scanlines), or profilers (which consist of a time-ordered series of profiles). The *Grid* interface is

175-WP-001-002

designed to support data that has been organized in a rectilinear array, based on a well defined and explicitly supported projection.

3.2.2 Structure of an HDF-EOS File

An HDF-EOS file is any valid HDF file (i.e., any file created by the NCSA HDF library), that contains a family of global attributes called "coremetadata.X", where ".X" is a sequence number beginning at 0 and running as high as 9. Optional data objects which may appear in an HDF-EOS file include, another family of global attributes called "archivemetadata.X" and any number of Point, Swath, and/or Grid data structures. The existence of Point, Swath, or Grid structures in an HDF-EOS file implies the existence of another family of global attributes called "StructMetadata.X". In Section 4, we give an additional overview of ECS metadata, it's role in ECS and also to metadata access and creation. The scope of this Primer however, is a discussion of data format issues for ECS standard data products. The reader is referred to The Science Data Processing (SDP) Toolkit Users Guide for the ECS Project, and other documents in the Reference Section for details of the purpose of and usage of ECS metadata. Creation of ECS metadata requires use of the ECS Science Data Processing (SDP) Toolkit (see the SDP Toolkit Users Guide for the ECS Project). (http://newsroom.gsfc.nasa.gov/sdptoolkit/toolkit.html)

Core Metadata

Core metadata represent information which is used to populate searchable database tables within the ECS archives. Data users use this information to locate particular HDF-EOS data granules. These metadata, which are defined in Release B-1 Earth Sciences Data Model, are also copied in the "coremetadata.X" (X= 0,...,n) family of global attributes within an HDF-EOS file. The syntax of these metadata is compliant with the Object Description Language (ODL). Tools for formatting, accessing and writing core metadata are provided in the SDP Toolkit.

Archive Metadata

Archive metadata represent information that, by definition, will not be searchable. It contains whatever information the file creator considers useful to be in the file, but which will not be directly accessible by ECS databases. That is ECS will not perform search and order or other services based on Archive metadata. These services are performed on Core metadata. Archive metadata are also accessed via SDP Toolkit calls and are written in ODL syntax into the "archivemetadata.X", (X=0,...,n) family of global attributes. (see SDP Toolkit Users Guide for the ECS Project).

Structural Metadata

Structural metadata describe the contents and structure of an HDF-EOS file. That is, these metadata describe how geolocation, temporal, projection information are to be associated with the data itself. Structural metadata are present in the file only if the HDF-EOS library has been invoked to create a Grid, Point, or Swath structure. These metadata are stored in the "StructMetadata.X" family of global attributes and are created and maintained by the HDF-EOS library. They are also stored in ODL format. These metadata are not intended to be directly

accessed by data producers or users. Therefore, all access to these metadata should be via appropriate function calls in the HDF-EOS library.

Point Structure

Point structures are implemented in HDF-EOS files as a hierarchy of Vgroups containing several Vdatas, i.e. tables. All Vgroups and Vdatas that are part of any Point structure carry the class "POINT". The Point structure can be implemented in a hierarchical set of 'levels'. For example, data location, data parameters at each location and parameter attributes, represent three levels in a hierarchy. Each level of data within a Point structure is implemented as a single Vdata, with each data field being a named field in the Vdata.

The following limitations apply to Point structures and should be kept in mind by EOSView users:

- The reserved field names for special purpose geolocation fields are "Longitude", "Latitude", "Colatitude", and "Time" (case sensitive). These fields are subject to the following requirements:

Field Name	Data Type	Format
Longitude	float32 or float64	Decimal degrees on the range [-180.0, 180.0)
Latitude	float32 or float64	Decimal degrees on the range [-90.0, 90.0]
Colatitude	float32 or float64	Decimal degrees on the range [0.0, 180.0]
Time	float64	TAI93 (seconds until(-)/since(+) midnight, 1/1/93)

- Fields may only be one-dimensional.

- Up to 5 levels may exist in a Point structure.

- Field names may be up to 64 characters in length.

- Any character can be used with the exception of, ",", ";", " and "/".

- Names are case sensitive.

- Names must be unique within a particular Point structure.

Swath Structure

Swath structures are implemented as a hierarchy of Vgroups containing a number of Vdatas and/or SDSs, i.e. tables and multi-dimensional arrays. All Vgroups and Vdatas that are part of any Swath structure carry the class "SWATH". Each one-dimensional field is implemented as a named field within its own Vdata. One-dimensional fields that are the same length, are merged into "communal" Vdatas, with each data field occupying one field in the Vdata.

Each multi-dimensional field is implemented as an SDS. Three-dimensional fields which share the same dimensionality, dimension sizes, and data type and which are specifically allowed by the calling program are merged into communal SDSs with three dimensions. Two-dimensional arrays are merged as if they were three-dimensional arrays with a first dimension of size 1. No merging is

performed on fields with more than three dimensions, on fields with an unlimited dimension, or on compressed fields.

The following limitations apply to Swath structures:

- The reserved field names for special purpose geolocation fields are "Longitude", "Latitude", "Colatitude", and "Time" (case sensitive). These fields are subject to the following requirements:

Field Name	Data Type	Format
Longitude	float32 or float64	Decimal degrees on the range [-180.0, 180.0)
Latitude	float32 or float64	Decimal degrees on the range [-90.0, 90.0]
Colatitude	float32 or float64	Decimal degrees on the range [0.0, 180.0]
Time	float64	TAI93 (seconds until(-)/since(+) midnight, 1/1/93)

These fields may be one- or two-dimensional.

- Non-reserved fields may have up to 8 dimensions.

- An "unlimited" dimension must be the first dimension (in C-order).

- For all multi-dimensional fields in scan- or profile-oriented Swaths, the dimension representing the "along track" dimension must precede the dimension representing the scan or profile dimension(s).

- Compression is selectable at the field level within a Swath. All HDF-supported compression methods are available through the HDF-EOS library. Specifying compression on a field prevents merging.   The compression method is stored within the file. Subsequent use of the library will un-compress the file.

- Field names may be up to 64 characters in length.

- Any character can be used with the exception of, ",", ";", " and "/".

- Names are case sensitive.

- Names must be unique within a particular Swath structure.

Grid Structure

Grid structures are implemented as a hierarchy of Vgroups containing several SDSs. All Vgroups that are part of any Grid structure carry the class "GRID". Each data field within a Grid structure is implemented as a single SDS. Merging is done the same way for Grid data fields as for multi-dimensional Swath data fields.

The following limitations apply to Grid structures:

- Fields may have from 2 to 8 dimensions.

- Compression is selectable at the field level within a Grid. All HDF-supported compression methods are available through the HDF-EOS library. Specifying compression on a field prevents merging. The compression method is stored within the file. Subsequent use of the library will un-compress the file.

- Field names may be up to 64 characters in length.

- Any character can be used with the exception of, ",", ";", " and "/".

- Names are case sensitive.

- Names must be unique within a particular Point structure.

Combinations

An HDF-EOS file can contain any number of Grid, Point and Swath data structures, up to a two Gigabyte lime for 32-bit addressing in HDF 4. An HDF-EOS file can also contain plain HDF objects for special purposes. HDF objects must be accessed by the HDF library and not by HDF-EOS extensions. A user should note however, that inclusion of HDF objects will require more knowledge of file contents on the part of an applications developer or data user. A user should also note that HDF is a directory structure and that a file containing 1000's of objects could cause program execution slow-downs.

3.3 HDF-EOS Library Functionality

The HDF-EOS libraries provide the following basic functionality for Point, Swath and Grid Structures:

- *Access routines* which initialize and terminate access to the data sets (including opening and closing files).

- *Definition* routines which allow the user to set key features of structures within data sets.

- *Basic I/O* routines which read and write data and structural metadata to data sets.

- *Inquiry* routines which return information about data structures contained within data sets.

- *Subset* routines which allow reading of data from a specified geographic, temporal or vertical regions.

Details of the user interface to the HDF-EOS libraries are presented in the HDF-EOS Library Users' Guide for the ECS Project, Volumes 1 and 2, 170-TP-510-001 and 170-TP-510-002, respectively.

In the following Sections, we give a synopsis of the Grid, Point and Swath structures and software interfaces, taken from the Users Guides.

3.3.1 Point Data Interface

3.3.1.1 Introduction

A Point Data set is made up of a series of data records taken at [possibly] irregular time intervals and at scattered geographic locations. Point Data is the most loosely organized form of geo-located data supported by HDF-EOS. Simply put, each data record consists of a set of one or more data values representing, in some sense, the state of a point in time and/or space.

Figure 3-1 shows an example of a simple point data set. In this example, each star on the map represents a reporting station. Each record in the data table contains the location of the point on the Earth and the measurements of the temperature and dew point at that location. This sort of point data set might represent a snapshot in time of a network of stationary weather reporting facilities.

Lat	Lon	Temp(C)	Dewpt(C)
61.12	-149.48	15.00	5.00
45.31	-122.41	17.00	5.00
38.50	-77.00	24.00	7.00
38.39	-90.15	27.00	11.00
30.00	-90.05	22.00	7.00
37.45	-122.26	25.00	10.00
18.00	-76.45	27.00	4.00
43.40	-79.23	30.00	14.00
34.03	-118.14	25.00	4.00
32.45	-96.48	32.00	8.00
33.30	-112.00	30.00	10.00
42.15	-71.07	28.00	7.00
35.05	-106.40	30.00	9.00
34.12	-77.56	28.00	9.00
46.32	-87.25	30.00	8.00
47.36	-122.20	32.00	15.00
39.44	-104.59	31.00	16.00
21.25	-78.00	28.00	7.00
44.58	-93.15	32.00	13.00
41.49	-87.37	28.00	9.00
25.45	-80.11	19.00	3.00

Figure 3-1. A Simple Point Data Example

A more realistic example might record the changes in the parameters over time by including multiple values of the parameters for each location. In this case, the identity and location of the reporting stations would remain constant, while the values of the measured parameters would vary. This sort of set up naturally leads to a hierarchical table arrangement where a second table is used to record the static information about each reporting station. This removes the redundant information that would be required by a single "flat" table and acts as an index for quick access to the main data table. Such an arrangement is depicted in Figure 3-2.

175-WP-001-002

```
 _ Station      Lat      Lon                    Time   Temp(C)
Chicago        41.49   -87.37                  ⎛ 0800     -3
Los Angeles    34.03  -118.14                  ⎜ 0900     -2
Washington     38.50   -77.00                  ⎝ 1000     -1
Miami          25.45   -80.11                  ⎛ 0800     20
                                               ⎜ 0900     21
                                               ⎜ 1000     22
                                               ⎝ 1100     24
                                               ⎛ 1000      6
                                               ⎜ 1100      8
                                               ⎜ 1200      9
                                               ⎜ 1300     11
                                               ⎝ 1400     12
                                               ⎛ 0600     15
                                               ⎝ 0700     16
```
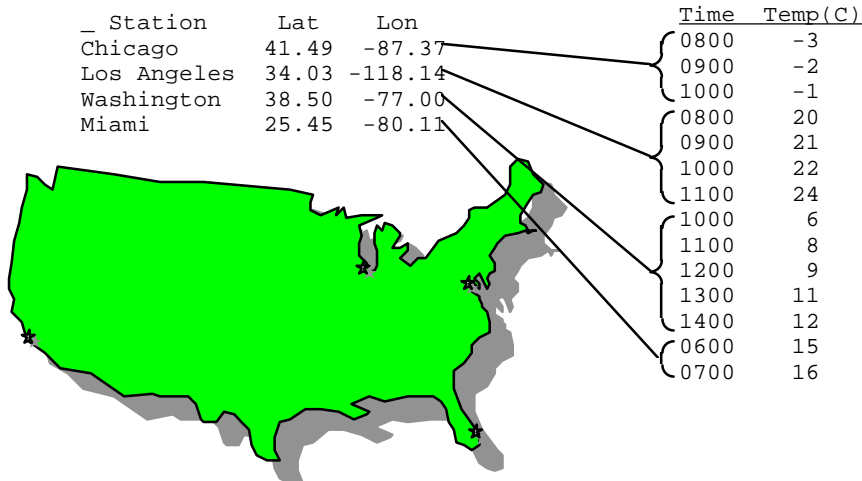
Figure 3-2. Recording Points Over Time

An even more complex point data set may represent data taken at various times aboard a moving ship. Here, the only thing that remains constant is the identity of the reporting ship. Its location varies with each data reading and is therefore treated similarly to the data. Although this example seems more complicated than the static example cited above, its implementation is nearly identical. Figure 3-3 shows the tables resulting from this example. Note that the station location information has been moved from the static table to the data table.

```
                               Lat      Lon     Temp(C)   Salinity(P
                             ⎛ 34.12  -74.56      18        146
  Ship ID                    ⎜ 33.24  -74.62      18        158
SS Botany Bay                ⎜ 32.45  -75.65      19        151
SS Dinghy                    ⎜ 32.03  -76.02      19        160
                             ⎜ 30.94  -76.48      20        162
                             ⎜ 30.04  -76.89      20        165
                             ⎝ 29.88  -76.94      21        159
                             ⎛ 25.51  -84.67      26        167
                             ⎜ 25.97  -85.06      26        173
                             ⎜ 26.22  -85.65      25        168
                             ⎜ 26.53  -86.14      26        162
                             ⎜ 26.58  -86.74      27        144
                             ⎜ 26.62  -87.45      27        156
                             ⎜ 26.87  -87.97      25        160
                             ⎝ 27.09  -88.68      25        166
```

Figure 3-3. Point Data from a Moving Platform

175-WP-001-002

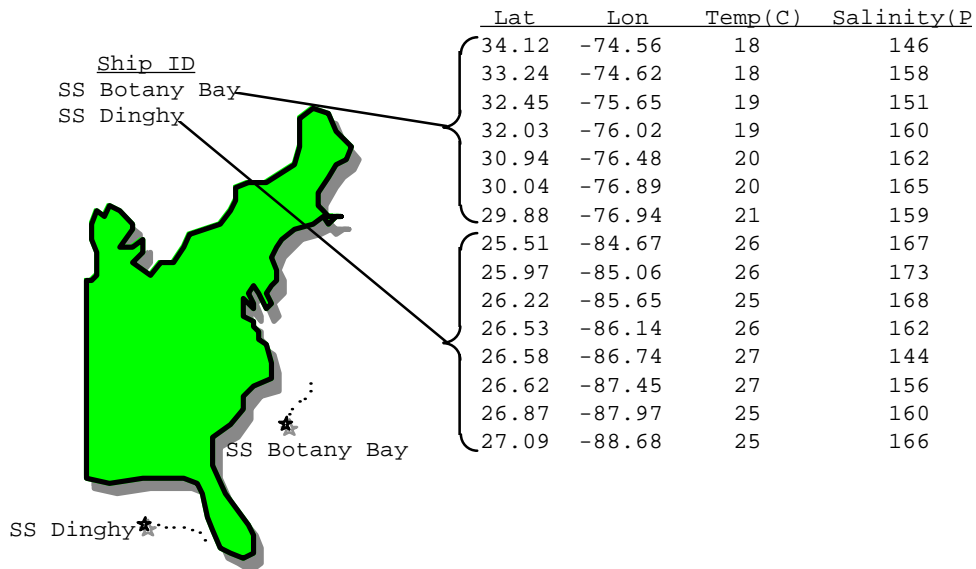In fact, the hierarchical arrangement of the tables in the last two examples can be expanded upon to include up to five indexing levels. A normal data access on a multi-level hierarchical point data set would involve starting at the top (first) level and following successive pointers down the structure until the desired information is found. As each level is traversed, more and more specific information is gained about the data.

In rare cases, it may be advantageous to access a point data set from the bottom (lowest level of the hierarchy). The point data model implemented in HDF-EOS provides for backward (or upward) pointers, which facilitate bottom-up access. That is, from the lowest level to the highest.

3.3.1.2 Applicability

The Point data model is very flexible and can be used for data at almost any level of processing. It is expected that point structure will be used for data for which there is no spatial or temporal organization, although lack of those characteristics do not preclude the use of a point structure. For example, profile data which is accumulated in sparsely located spatial averages may be most useful in a point structure.

3.3.1.3 The Point Data Interface

The PT interface consists of routines for storing, retrieving, and manipulating data in point data sets.

PT API Routines

All C routine names in the point data interface have the prefix "PT" and the equivalent FORTRAN routine names are prefixed by "pt." The PT routines are classified into the following categories:

- *Access routines* initialize and terminate access to the PT interface and point data sets (including opening and closing files).

- *Definition* routines allow the user to set key features of a point data set.

- *Basic I/O* routines read and write data and metadata to a point data set.

- *Index I/O* routines read and write information which links two tables in a point data set.

- *Inquiry* routines return information about data contained in a point data set.

- *Subset* routines allow reading of data from a specified geographic region.

The PT function calls are listed and are described in detail in the HDF-EOS Users Guides

File Identifiers

As with all HDF-EOS interfaces, file identifiers in the PT interface are 32-bit values, each uniquely identifying one open data file. They are not interchangeable with other file identifiers created with other interfaces.

Point Identifiers

Before a point data set is accessed, it is identified by a name which is assigned to it upon its creation. The name is used to obtain a *point identifier*. After a point data set has been opened for access, it is uniquely identified by its point identifier.

3.3.1.4 Programming Model

The programming model for accessing a point data set through the PT interface is as follows:

1. Open the file and initialize the PT interface by obtaining a file id from a file name.

2. Open OR create a point data set by obtaining a point id from a point name.

3. Perform desired operations on the data set.

4. Close the point data set by disposing of the point id.

5. Terminate point access to the file by disposing of the file id.

To access a single point data set that already exists in an HDF-EOS file, the calling program must contain the following sequence of C calls:

```
file_id = PTopen(filename, access_mode);
pt_id = PTattach(file_id, point_name);
<Optional operations>
status = PTdetach(pt_id);
status = PTclose(file_id);
```

To access several files at the same time, a calling program must obtain a separate id for each file to be opened. Similarly, to access more than one point data set, a calling program must obtain a separate point id for each data set. For example, to open two data sets stored in two files, a program would execute the following series of C function calls:

```
file_id_1 = PTopen(filename_1, access_mode);
pt_id_1 = PTattach(file_id_1, point_name_1);
file_id_2 = PTopen(filename_2, access_mode);
pt_id_2 = PTattach(file_id_2, point_name_2);
<Optional operations>
status = PTdetach(pt_id_1);
status = PTclose(file_id_1);
status = PTdetach(pt_id_2);
status = PTclose(file_id_2);
```

Because each file and point data set is assigned its own identifier, the order in which files and data sets are accessed is very flexible. However, it is very important that the calling program individually discard each identifier before terminating. Failure to do so can result in empty or, even worse, invalid files being produced.

It is permissible to have any number of Point (Grid, Swath) objects in a single HDF EOS file. PTopen () must be called to open each object (structure). It is o.k. to have more than one object open at a time.

3.3.2 Swath Interface

3.3.2.1 Introduction

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 3-4 below shows this traditional view of a swath.
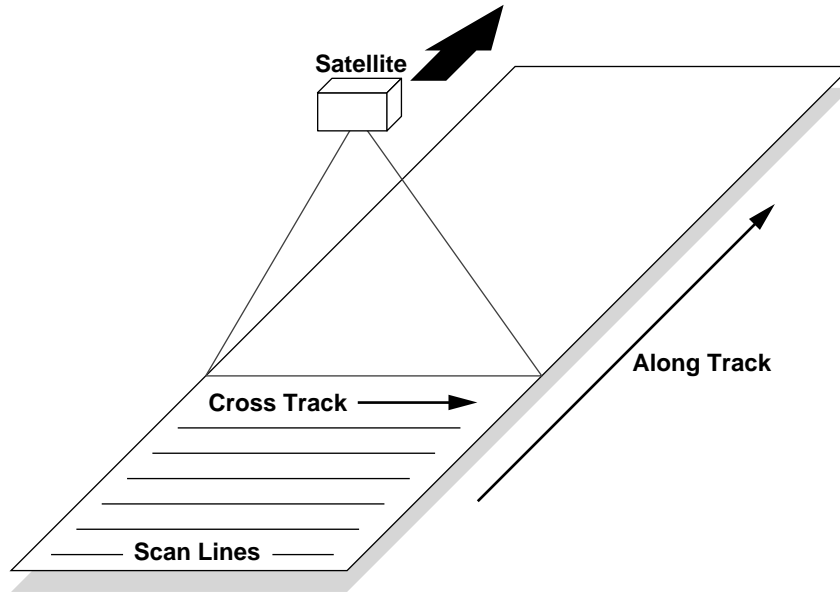
Figure 3.4. A Typical Satellite Swath: Scanning Instrument

Another type of data that the Swath is equally well suited to arises from a sensor that measures a vertical profile, instead of scanning across the ground track. The resulting data resembles a standard Swath tipped up on its edge. Figure 3-5 shows how such a Swath might look.

In fact, the two approaches shown in Figures 3-4 and 3-5 can be combined to manage a profiling instrument that scans across the ground track. The result would be a three dimensional array of measurements where two of the dimensions correspond to the standard scanning dimensions (along the ground track and across the ground track), and the third dimension represents a height above the Earth or a range from the sensor. The "horizontal" dimensions can be handled as normal geographic dimensions, while the third dimension can be handled as a special "vertical" dimension.
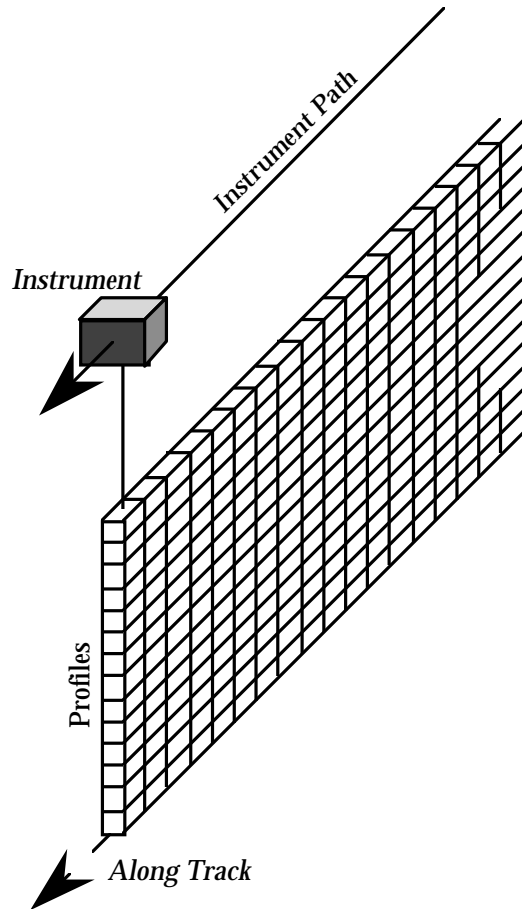
Figure 3-5. A Swath Derived from a Profiling Instrument

A standard Swath is made up of four primary parts: data fields, geolocation fields, dimensions, and dimension maps. An optional fifth part called an index can be added to support certain kinds of access to Swath data. Each of the parts of a Swath is described in detail in the following subsections.

Data Fields

Data fields are the main part of a Swath from a science perspective. Data fields usually contain the raw data (often as *counts*) taken by the sensor or parameters derived from that data on a value-for-value basis. All the other parts of the Swath exist to provide information about the data fields or to support particular types of access to them. Data fields typically are two-dimensional arrays, but can have as few as one dimension or as many as eight, in the current library implementation. They can have any valid C data type.

175-WP-001-002

Geolocation Fields

Geolocation fields allow the Swath to be accurately tied to particular points on the Earth's surface. To do this, the Swath interface requires the presence of at least a time field ("Time") or a latitude/longitude field pair ("Latitude"[1] and "Longitude"). Geolocation fields must be either one- or two-dimensional and can have any data type. A latitude/longitude pair is generally referred to the center of pixels defined in Data Fields. The values are generally in geographic coordinates.

Dimensions

Dimensions define the axes of the data and geolocation fields by giving them names and sizes. In using the library, dimensions must be defined before they can be used to describe data or geolocation fields.

Every axis of every data or geolocation field, then, must have a dimension associated with it. However, there is no requirement that they all be unique. In other words, different data and geolocation fields may share the same named dimension. In fact, sharing dimension names allows the Swath interface to make some assumptions about the data and geolocation fields involved which can reduce the complexity of the file and simplify the program creating or reading the file.

Dimension Maps

Dimension maps are the glue that holds the Swath together. They define the relationship between data fields and geolocation fields by defining, one-by-one, the relationship of each dimension of each geolocation field with the corresponding dimension in each data field. In cases where a data field and a geolocation field share a named dimension, no explicit dimension map is needed. In cases where a data field has more dimensions than the geolocation fields, the "extra" dimensions are left un-mapped.

In many cases, the size of a geolocation dimension will be different from the size of the corresponding data dimension. To take care of such occurrences, there are two pieces of information that must be supplied when defining a dimension map: the *offset* and the *increment*. The offset tells how far along a data dimension you must travel to find the first point to have a corresponding entry along the geolocation dimension. The increment tells how many points to travel along the data dimension before the next point is found for which there is a corresponding entry along the geolocation dimension. Figure 3-6 depicts a dimension map.

---

[1] "Colatitude" may be substituted for "Latitude."

Geolocation Dimension

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Mapping
Offset: 1
Increment: 2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
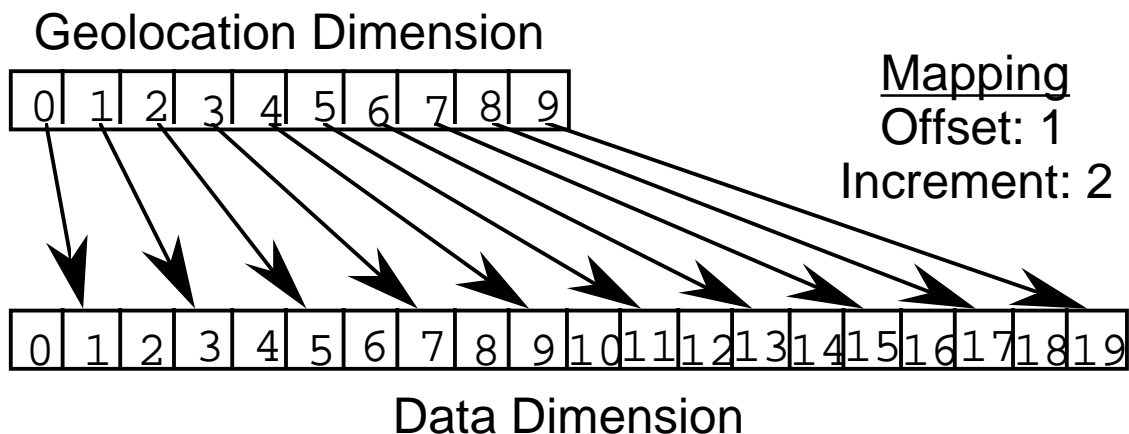
Data Dimension

Figure 3-6. A "Normal" Dimension Map

The "data skipping" method described above works quite well if there are fewer regularly spaced geolocation points than data points along a particular pair of mapped dimensions of a Swath. It is conceivable, however, that the reverse is true – that there are more regularly spaced geolocation points than data points. In that event, both the offset and increment should be expressed as negative values to indicate the reversed relationship. The result is shown in Figure 3-7. Note that in the reversed relationship, the offset and increment are applied to the geolocation dimension rather than the data dimension.

Geolocation Dimension

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

Mapping
Offset: -1
Increment: -2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Data Dimension

Figure 3-7. A "Backwards" Dimension Map

Index

The index was designed specifically for Landsat 7 data products. These products require geolocation information that does not repeat at regular intervals throughout the Swath. The index allows the Swath to be broken into unequal length *scenes* which can be individually geolocated.

175-WP-001-002

For this version of the HDF-EOS library, there is no particular content required for the index. It is quite likely that a later version of the library will impose content requirements on the index in an effort to standardize its use.
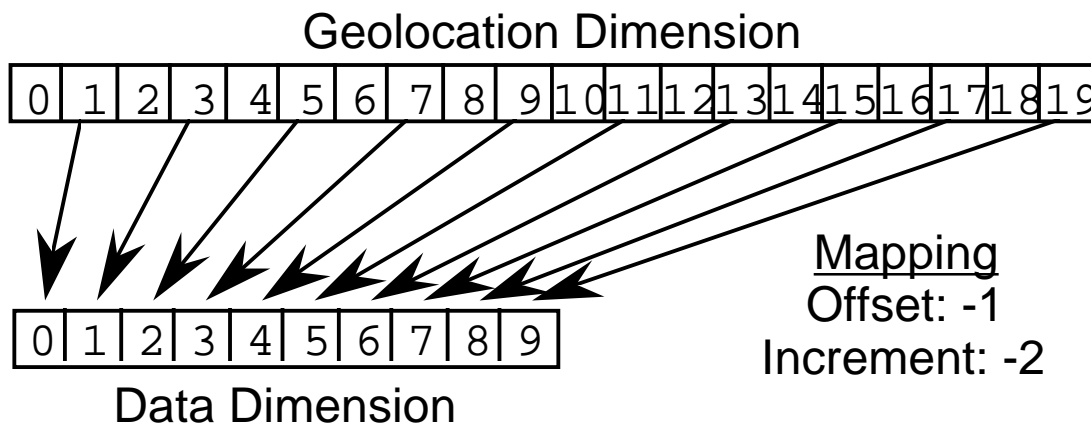
3.3.2.2 Applicability

The Swath data model is most useful for satellite [or similar] data at a low level of processing. The Swath model is best suited to data at EOS processing levels 1A, 1B, and 2. Swath structures are for data storage by MODIS, MISR, MOPITT, ASTER instrument teams on EOS-Terra and AIRS in EOS-AQUA.

3.3.2.3 The Swath Data Interface

The SW interface consists of routines for storing, retrieving, and manipulating data in swath data sets.

SW API Routines

All C routine names in the swath data interface have the prefix "SW" and the equivalent FORTRAN routine names are prefixed by "sw." The SW routines are classified into the following categories:

- *Access routines* initialize and terminate access to the SW interface and swath data sets (including opening and closing files).

- *Definition* routines allow the user to set key features of a swath data set.

- *Basic I/O* routines read and write data and metadata to a swath data set.

- *Inquiry* routines return information about data contained in a swath data set.

- *Subset* routines allow reading of data from a specified geographic region or by time

The SW function calls are listed and are described in detail in the HDF-EOS Users Guides.

File Identifiers

As with all HDF-EOS interfaces, file identifiers in the SW interface are 32-bit values, each uniquely identifying one open data file. They are not interchangeable with other file identifiers created with other interfaces.

Swath Identifiers

Before a swath data set is accessed, it is identified by a name which is assigned to it upon its creation. The name is used to obtain a *swath identifier*. After a swath data set has been opened for access, it is uniquely identified by its swath identifier.

3.3.2.4 Programming Model

The programming model for accessing a swath data set through the SW interface is as follows:

1. Open the file and initialize the SW interface by obtaining a file id from a file name.
2. Open OR create a swath data set by obtaining a swath id from a swath name.
3. Perform desired operations on the data set.
4. Close the swath data set by disposing of the swath id.
5. Terminate swath access to the file by disposing of the file id.

To access a single swath data set that already exists in an HDF-EOS file, the calling program must contain the following sequence of C calls:

```
file_id = SWopen(filename, access_mode);
sw_id = SWattach(file_id, swath_name);
<Optional operations>
status = SWdetach(sw_id);
status = SWclose(file_id);
```

To access several files at the same time, a calling program must obtain a separate id for each file to be opened. Similarly, to access more than one swath data set, a calling program must obtain a separate swath id for each data set. For example, to open two data sets stored in two files, a program would execute the following series of C function calls:

```
file_id_1 = SWopen(filename_1, access_mode);
sw_id_1 = SWattach(file_id_1, swath_name_1);
file_id_2 = SWopen(filename_2, access_mode);
sw_id_2 = SWattach(file_id_2, swath_name_2);
<Optional operations>
status = SWdetach(sw_id_1);
status = SWclose(file_id_1);
status = SWdetach(sw_id_2);
status = SWclose(file_id_2);
```

Because each file and swath data set is assigned its own identifier, the order in which files and data sets are accessed is very flexible. However, it is very important that the calling program individually discard each identifier before terminating. Failure to do so can result in empty or, even worse, invalid files being produced.

It is permissible to have any number of Swath (Grid, Point) objects in a single HDF EOS file. PTopen () must be called to open each object (structure). It is o.k. to have more than one object open at a time.

3.3.2.5 A Sample Usage of the Swath Interface

In this section, we show a programming example of the usage of the HDF-EOS Swath interface. In a series of short programs, we will open a swath file, define dimensions, write data, extract information about the file and read data from the file. We will show the underlying HDF objects that are created by the interface. For additional detail, the reader is directed to the HDF-EOS Users Guides for the ECS Project. (http://newsroom.gsfc.nasa.gov/sdptoolkit/toolkit.html)

Swath Structure Creation: SetupSwath.c


```c
#include "mfhdf.h"
/*
 * In this example we will (1) open an HDF file, (2) create the swath
 * interface within the file and (3) define the swath field dimensions.
 */

main()
{

    intn            status, i, j;
    int32           swfid, SWid, indx[12]={0,1,3,6,7,8,11,12,14,24,32,39};

    /*
     * We first open the HDF swath file, "SwathFile.hdf".  Because this file
     * does not already exist, we use the DFACC_CREATE access code in the
     * open statement.  The SWopen routine returns the swath file id, swfid,
     * which is used to identify the file in subsequent routines in the
     * library. This call creates 1 Vgroup and 2 Vdatas in the HDF file.
     * The Vgroupis named from the first parameter in SWopen, the two Vdatas are
     * used by the HDF-EOS library. They are called StructMetadata.0 and
     * HDFEOSVersion.  The 2 Vdatas are contained in the Vgroup as shown in
     * the following figure.
     */

    swfid = SWopen("SwathFile.hdf", DFACC_CREATE);

    /*
     * The first of these, SWcreate, creates the swath, "Swath1", within the
     * file designated by the file id, swfid. It returns the swath id, SWid,
     * which identifies the swath in subsequent routines.  We will show how
     * to define, write and read field swaths in later programs.
     * This call creates 4 Vgroups in the HDF file.  The main Vgroup called
     * Swath1 contains the 3 other Vgroups created by the HDF-EOS library.
     *They are called "Geolocation Fields", "Data Fields" and "Swath
     * Attributes".
     * These Vgroups are used by the HDF-EOS library to keep track of
     * the fields and other objects created by the user.
     */

    SWid = SWcreate(swfid, "Swath1");

    /*
     * Typically, many fields within a swath share the same dimension. The
     * swath interface therefore provides a way of defining dimensions that
     * will then be used to define swath fields. A dimension is defined with
     * a name and a size and is connected to the particular swath through
     * the swath id. In this example, we define the geolocation track and
     * cross track dimensions with size 20 and 10 respectively and two
     * dimensions corresponding to these but with twice the resolution.
     * We also define a dimension corresponding to a number of spectral
     * bands. The dimension information for the file is contained in the
     * StructMetadata.0 Vdata.  So this means no new HDF objects are created
     * for the dimensions.
     */
```

175-WP-001-002

```
        status = SWdefdim(SWid, "GeoTrack", 20);
        status = SWdefdim(SWid, "GeoXtrack", 10);
        status = SWdefdim(SWid, "Res2tr", 40);
        status = SWdefdim(SWid, "Res2xtr", 20);
        status = SWdefdim(SWid, "Bands", 15);
        status = SWdefdim(SWid, "IndxTrack", 12);

        /* Define Unlimited Dimension */

        status = SWdefdim(SWid, "Unlim", NC_UNLIMITED);

        /*
         * Once the dimensions are defined,the relationship (mapping)between the
         * geolocation dimensions, such as track and cross track, and the data
         * dimensions, must be established. This is done through the SWdefdimmap
         * routine.  It takes as input the swath id, the names of the dimensions
         * designating the geolocation and data dimensions, respectively, and the
         * offset and increment defining the relation.
         *
         * In the first example we relate the "GeoTrack" and "Res2tr" dimensions
         * with an offset of 0 and an increment of 2.  Thus the ith element of
         * "Geotrack" corresponds to the 2 * ith element of "Res2tr".
         *
         * In the second example, the ith element of "GeoXtrack" corresponds to
         * the 2 * ith + 1 element of "Res2xtr".
         *
         * Note that there is no relationship between the geolocation dimensions
         * and the "Bands" dimension.
         *
         * The information for the Regular dimension mapping is contained in the
         * StructMetadata.0 Vdata as character data.
         * For Index mapping relationships, that information is stored in a Vdata
         * that is contained in the "SwathFile.hdf" Vgroup created by the SWopen
         * call as integer data.  This is done because the values change and
         * this relationship could contain a fairly large number of values.
         */

        status = SWdefdimmap(SWid, "GeoTrack", "Res2tr", 0, 2);
        status = SWdefdimmap(SWid, "GeoXtrack", "Res2xtr", 1, 2);

        /* Define Indexed Mapping */
        status = SWdefidxmap(SWid, "IndxTrack", "Res2tr", indx);

        /*
         * We now close the swath interface with the SWdetach routine. This step
         * is necessary to properly store the swath information within the file.
         */

        status = SWdetach(SWid);

        /*
         * Finally, we close the swath file using the SWclose routine. This will
         * release the swath file handles established by SWopen.
         */

        status = SWclose(swfid);

        return;
```

```
}
```

Figure 3-8. shows the HDF objects created by execution of the program SetupSwath.c.
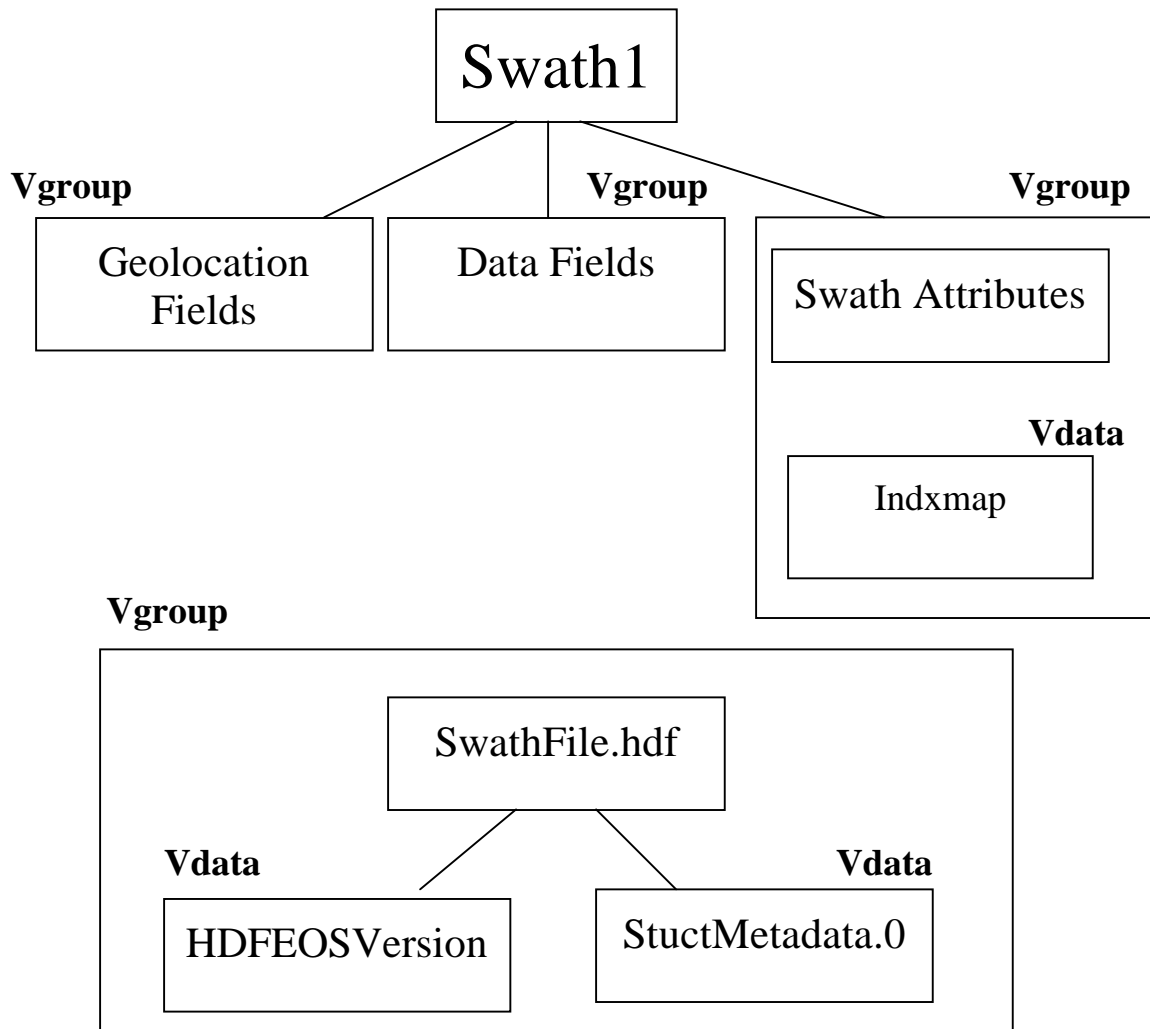

Figure 3-8. HDF Objects Created by Program: SetupSwath.c

Swath Structure Creation: DefineFields.c

```
#include "hdf.h"
#include "HdfEosDef.h"

/*
 * In this example we will (1) open the "SwathFile" HDF file, (2) attach to
```

```
 * the "Swath1" swath, and (3) define the swath fields.
 */


main()
{

    intn            status, i, j;
    int32           swfid, SWid;

/*
 * We first open the HDF swath file, "SwathFile.hdf".  Because this file
 * already exist and we wish to write to it, we use the DFACC_RDWR access
 * code in the open statement. The SWopen routine returns the swath file
 * id, swfid, which is used to identify the file in subsequent routines.
 */

    swfid = SWopen("SwathFile.hdf", DFACC_RDWR);

    /*
     * If the swath file cannot be found, SWopen will return -1 for the file
     * handle (swfid).  We there check that this is not the case before
     * proceeding with the other routines.
     *
     * The SWattach routine returns handle to the existing swath "Swath1",
     * SWid.  If the swath is not found, SWattach returns -1 for the handle.
     */

    if (swfid != -1)
    {

        SWid = SWattach(swfid, "Swath1");
        if (SWid != -1)
        {

            /*
             * We define seven fields.  The first three, "Time", "Longitude"
             * and "Latitude" are geolocation fields and thus we use the
             * geolocation dimensions "GeoTrack" and "GeoXtrack"in the field
             * definitions.  We also must specify the data type using the
             * standard HDF data type codes. In this example the geolocation
             * are 4-byte (32 bit) floating point numbers.
             * The next four fields are data fields.  Note that either
             * geolocation or data dimensions can be used.  If an error
             * occurs during the definition, such as a dimension that cannot
             * be found, then the return status will be set to -1.
             */

            /* This call creates a Vdata with 20 records located
             * in the "Geolocation Fields" Vgroup.
             */
            status = SWdefgeofield(SWid, "Time", "GeoTrack",
                                    DFNT_FLOAT64, HDFE_NOMERGE);

            /* The next two calls create a single SDS.  This is
             * also located in the "Geolocation Fields" Vgroup.  There is
             * only 1 SDS because the Merge Flag is turned on, so the library
             * will put the data for both fields into 1 object.
             */
            status = SWdefgeofield(SWid, "Longitude", "GeoTrack,GeoXtrack",
```

```
                                     DFNT_FLOAT32, HDFE_AUTOMERGE);

            status = SWdefgeofield(SWid, "Latitude", "GeoTrack,GeoXtrack",
                                   DFNT_FLOAT32, HDFE_AUTOMERGE);

            /* The next 3 calls create 3 SDSs.  They are contained in the
             * Vgroup called "Data Fields".  The Merge Flag is turned off
             * so this tells the HDF-EOS library to create separate objects
             * for each field.
             */
            status = SWdefdatafield(SWid, "Density", "GeoTrack",
                                    DFNT_FLOAT32, HDFE_NOMERGE);

           status = SWdefdatafield(SWid, "Temperature", "GeoTrack,GeoXtrack",
                                    DFNT_FLOAT32, HDFE_NOMERGE);

            status = SWdefdatafield(SWid, "Pressure", "Res2tr,Res2xtr",
                                    DFNT_FLOAT64, HDFE_NOMERGE);

            status = SWdefdatafield(SWid, "Spectra", "Bands,Res2tr,Res2xtr",
                                    DFNT_FLOAT64, HDFE_NOMERGE);

            /* Define Appendable Field */
            /* ---------------------- */
            /* This call creates a Vdata with a single value of -1.  This
             * object is located in the "Data Fields" Vgroup.
             */

            status = SWdefdatafield(SWid, "Count", "Unlim", DFNT_INT16,
                                    HDFE_NOMERGE);
        }
    }
    status = SWdetach(SWid);
    status = SWclose(swfid);

    return;
}
```

Figure 3-9. shows the HDF objects created by the program DefineFields.c

Swath1

Vgroup
Geolocation

Vgroup
Data Fields

Vgroup
Swath Attributes

Vdata
Time

SDS
Latitude
Longitude

Vdata
Indxmap

SDS
Density

SDS
Temperature

SDS
Pressure

Vdata
Count

SDS
Spectra
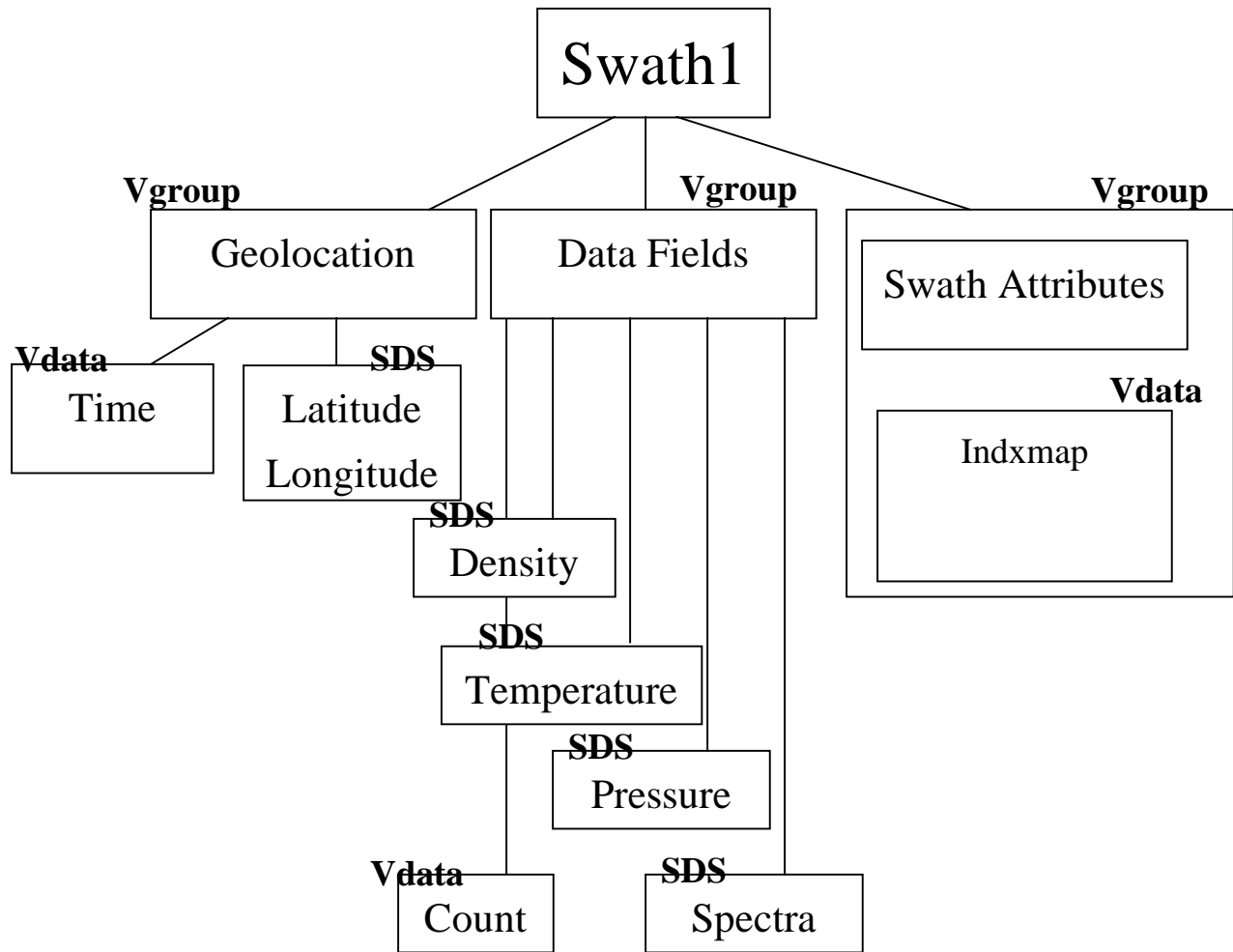
Figure 3-9. HDF Objects Created by Program: DefineField.c

Writing Data to a Swath File: WriteFields.c

```
#include "hdf.h"

/*
 * In this example we will (1) open the "SwathFile" HDF file, (2) attach to
 * the "Swath1" swath, and (3) write data to the "Longitude", "Latitude",
 * and "Spectra" fields.
 */

main()
{

    intn     i, j, k, status, track, xtrack, start[3], count[3];
    int32   swfid, SWid, attr[4]={3, 5, 7, 11};
    float32   lng[10] = {0.0, 1.0, 2.0, 3.0, 4.0,
    5.0, 6.0, 7.0, 8.0, 9.0}, lat[10];
```

```
/* Define longitude values along the cross track */

float64    plane[40*20], tme[20];

/*
 * Open the HDF swath file, "SwathFile.hdf".
 */

swfid = SWopen("SwathFile.hdf", DFACC_RDWR);

if (swfid != -1)
{

    /*
     * Attach the "Swath1" swath.
     */

    SWid = SWattach(swfid, "Swath1");

    if (SWid != -1)
    {

        /* Write data starting at the beginning of each cross track */
        start[1] = 0;
        count[0] = 1;
        count[1] = 10;

        /*
         * Loop through all the tracks, incrementing the track starting
         * position by one each time.
         */

        for (track = 0; track < 20; track++)
        {
            start[0] = track;
            status = SWwritefield(SWid, "Longitude", start, NULL,
                                  count, lng);

            for (xtrack = 0; xtrack < 10; xtrack++)
                lat[xtrack] = track;

            status = SWwritefield(SWid, "Latitude", start, NULL,
                                  count, lat);
        }


        /*
         *  Write Time Field
         */
        for (i=0;i<20;i++) tme[i] = 34574087.3 + 84893.2*i;
        status = SWwritefield(SWid, "Time", NULL, NULL,
                              NULL, tme);

        /*
         * Write Spectra one plane at a time
         * Value is 100 * track index + band index
         */

        start[1] = 0;
```

```
            start[2] = 0;
            count[0] = 1;
            count[1] = 40;
            count[2] = 20;
            for (i = 0; i < 15; i++)
            {
                start[0] = i;
                for (j=0; j<40; j++)
                    for (k= 0; k<20; k++)
                        plane[j*20+k] = j*100 + i;

                status = SWwritefield(SWid, "Spectra", start, NULL,
                                      count, plane);

            }

            /* Write User Attribute */
            status = SWwriteattr(SWid, "TestAttr", DFNT_INT32, 4, attr);

        }
    }

    status = SWdetach(SWid);
    status = SWclose(swfid);

    return;
}
```

Extracting Information from a Swath File: InquireSwath.c

```
#include "hdf.h"
#include "HdfEosDef.h"

/*
 * In this example we will retrieve (1)information about the dimensions, (2)
 * the dimension mappings (geolocation relations), and (3) the swath fields.
 */

main()
{

    intn            status, i;
    int32            swfid, SWid, ndims, nmaps, rk, nt, dim[8], nflds;
    int32          *dims, *off, *inc, *indx, *rank, *ntype;
    int32            n, strbufsize, dimsize, offset, incr, *sizes;
    char           *dimname, *dimmap, *fieldlist, dimlist[80];

    /*
     * Open the Swath File for read only access
     */

    swfid = SWopen("SwathFile.hdf", DFACC_READ);

    if (swfid != -1)
    {
```

3-23

175-WP-001-002

```
/* Attach the swath */

SWid = SWattach(swfid, "Swath1");

if (SWid != -1)
{
    /* Inquire Dimensions */
    ndims = SWnentries(SWid, HDFE_NENTDIM, &strbufsize);
    dims = (int32 *) calloc(ndims, 4);
    dimname = (char *) calloc(strbufsize + 1, 1);
    ndims = SWinqdims(SWid, dimname, dims);
    printf("Dimension list: %s\n", dimname);
    for (i = 0; i < ndims; i++)
        printf("dim size: %d\n", dims[i]);

    free(dims);
    free(dimname);

    /* Inquire Dimension Mappings */
    nmaps = SWnentries(SWid, HDFE_NENTMAP, &strbufsize);
    off = (int32 *) calloc(nmaps, 4);
    inc = (int32 *) calloc(nmaps, 4);
    dimmap = (char *) calloc(strbufsize + 1, 1);
    nmaps = SWinqmaps(SWid, dimmap, off, inc);
    printf("Dimension map: %s\n", dimmap);
    for (i = 0; i < nmaps; i++)
        printf("offset increment: %d %d\n", off[i], inc[i]);
    free(off);
    free(inc);
    free(dimmap);

    /* Inquire Indexed Dimension Mappings */
    nmaps = SWnentries(SWid, HDFE_NENTIMAP, &strbufsize);
    sizes = (int32 *) calloc(nmaps, 4);
    dimmap = (char *) calloc(strbufsize + 1, 1);
    nmaps = SWinqidxmaps(SWid, dimmap, sizes);
    printf("Index Dimension map: %s\n", dimmap);
    for (i = 0; i < nmaps; i++)
        printf("sizes: %d\n", sizes[i]);
    free(sizes);
    free(dimmap);

    /* Inquire Geolocation Fields */
    nflds = SWnentries(SWid, HDFE_NENTGFLD, &strbufsize);
    rank = (int32 *) calloc(nflds, 4);
    ntype = (int32 *) calloc(nflds, 4);
    fieldlist = (char *) calloc(strbufsize + 1, 1);
    nflds = SWinqgeofields(SWid, fieldlist, rank, ntype);
    printf("geo fields: %s\n", fieldlist);
    for (i = 0; i < nflds; i++)
        printf("rank type: %d %d\n", rank[i], ntype[i]);

    free(rank);
    free(ntype);
    free(fieldlist);

    /* Inquire Data Fields */
    nflds = SWnentries(SWid, HDFE_NENTDFLD, &strbufsize);
    rank = (int32 *) calloc(nflds, 4);
```

```
                    ntype = (int32 *) calloc(nflds, 4);
                    fieldlist = (char *) calloc(strbufsize + 1, 1);
                    nflds = SWinqdatafields(SWid, fieldlist, rank, ntype);

                    printf("data fields: %s\n", fieldlist);
                    for (i = 0; i < nflds; i++)
                        printf("rank type: %d %d\n", rank[i], ntype[i]);

                    free(rank);
                    free(ntype);
                    free(fieldlist);

                    /* Get info on "GeoTrack" dim */
                    dimsize = SWdiminfo(SWid, "GeoTrack");
                    printf("Size of GeoTrack: %d\n", dimsize);

                    /* Get info on "GeoTrack/Res2tr" mapping */
                    status = SWmapinfo(SWid, "GeoTrack", "Res2tr", &offset, &incr);
                    printf("Mapping Offset: %d\n", offset);
                    printf("Mapping Increment: %d\n", incr);

                    /* Get info on "IndxTrack/Res2tr" indexed mapping */
                    dimsize = SWdiminfo(SWid, "IndxTrack");
                    indx = (int32 *) calloc(dimsize, 4);
                    n = SWidxmapinfo(SWid, "IndxTrack", "Res2tr", indx);
                    for (i = 0; i < n; i++)
                        printf("Index Mapping Entry %d: %d\n", i+1, indx[i]);
                    free(indx);

                    /* Get info on "Longitude" Field */
                    status = SWfieldinfo(SWid, "Longitude", &rk, dim, &nt, dimlist);
                    printf("Longitude Rank: %d\n", rk);
                    printf("Longitude NumberType: %d\n", nt);
                    printf("Longitude Dimension List: %s\n", dimlist);
                    for (i=0; i<rk; i++)
                        printf("Dimension %d: %d\n",i+1,dim[i]);

            }
        }
        status = SWdetach(SWid);
        status = SWclose(swfid);

        return;
    }
```

Reading Data from a Swath File: ReadFields.c

```
  #include "hdf.h"

/*
 * In this example we will (1) open the "SwathFile" HDF file, (2) attach to
 * the "Swath1" swath, and (3) read data from the "Longitude" field.
 *
 * Unlike the WriteField routine, we will read the field all at once.
 */
```

```
main()
{

    intn            status, i, j, k, start[2],stride[2],count[2];
    int32           swfid, SWid, attr[4];
    float32         lng[20][10];
    /* Allocate space for the longitude and spectral data */

    /*
     * Open the HDF swath file, "SwathFile.hdf".
     */

    swfid = SWopen("SwathFile.hdf", DFACC_READ);

    if (swfid != -1)
    {

        /*
         * Attach the "Swath1" swath.
         */

        SWid = SWattach(swfid, "Swath1");

        if (SWid != -1)
        {
            /* Read the entire longitude field */
            status = SWreadfield(SWid, "Longitude", NULL, NULL, NULL, lng);

            /* Print field */
            for (i = 0; i < 20; i++)
                for (j = 0; j < 10; j++)
                    printf("i j Longitude: %d %d %f\n",
                            i, j, lng[i][j]);


            /* Read User Attribute */
            status = SWreadattr(SWid, "TestAttr", attr);
            for (i=0;i<4;i++)
                printf("Attribute Entry %d: %d\n",i+1,attr[i]);

        }
    }
    status = SWdetach(SWid);
    status = SWclose(swfid);
    HEprint(stdout,0);

    return;
}
```

3.3.3 Grid Interface

3.3.3.1 Introduction

This section will describe the routines available for storing and retrieving HDF-EOS *Grid Data*. A Grid data set is similar to a swath in that it contains a series of data fields of two or more dimensions. The main difference between a Grid and a Swath is in the character of their geolocation information.

As described in Section 3.3.2, Swaths carry geolocation information as a series of individually located points (tie points or ground control points). Grids, though, carry their geolocation in a much more compact form. A grid object contains a set of projection coefficients. These data are used in projection equations contained in the HDF-EOS software interface. Together, these relatively few pieces of information define the location of all points in the grid. The coefficients can then be used to compute the latitude and longitude for any point in the grid.
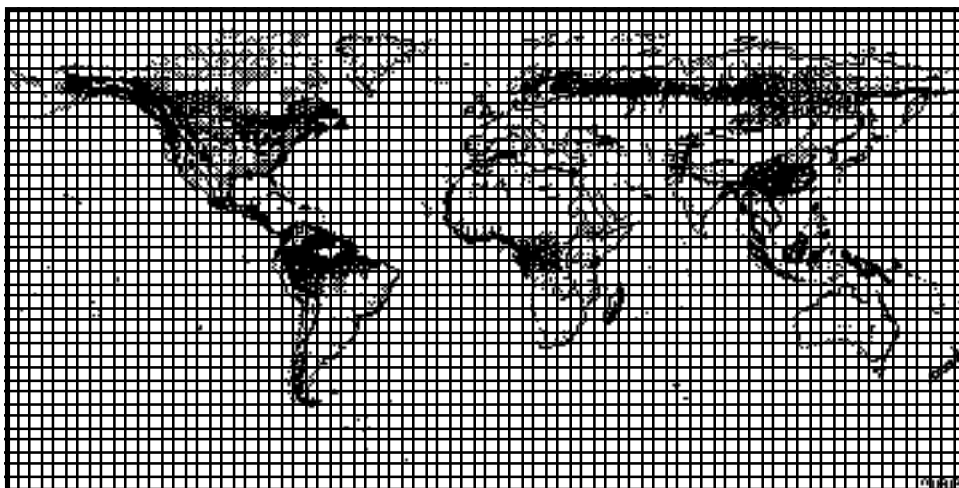

Figure 3-10. A Data Field in a Mercator-projected Grid

In loose terms, each data field constitutes a map in a given standard projection. Although there may be many independent Grids in a single HDF-EOS file, within each Grid only one projection may be chosen for application to all data fields. Figures 3-10 and 3-11 show how a single data field may look in a Grid using two common projections.

There are three important features of a Grid data set: the data fields, the dimensions, and the projection. Each of these is discussed in detail in the following subsections.
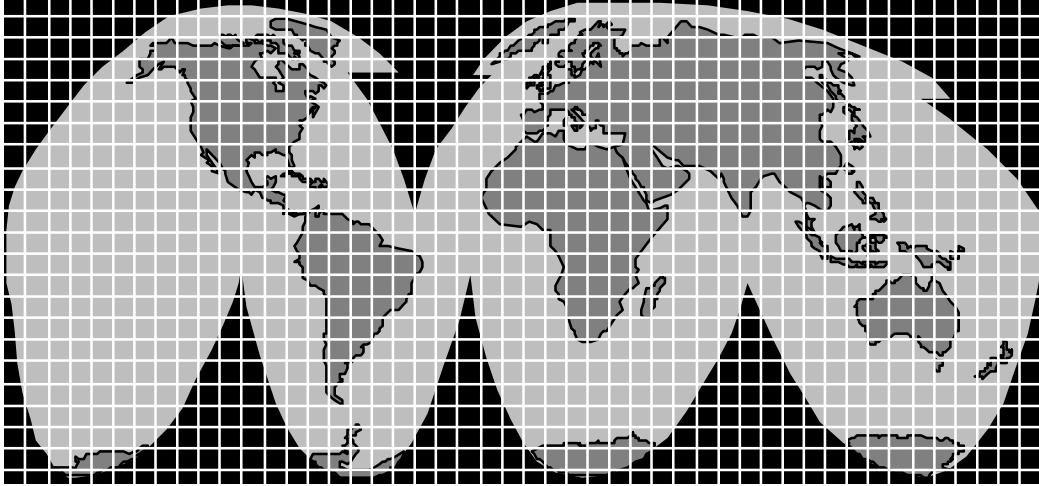
Figure 3-11. A Data Field in an Interrupted Goode's Homolosine-Projected Grid

Data Fields

The data fields are, of course, the most important part of the Grid. Data fields in a Grid data set and are rectilinear arrays of two or more dimensions. Most commonly, they are simply two-dimensional rectangular arrays. Generally, each field contains data of similar scientific nature which must share the same attributes. The data fields are related to each other by common geolocation. That is, a single set of geolocation information is used for all data fields within one Grid data set. Note that Grid Data fields can be 'stacked' to make three dimensional data sets. For example, a Grid structure could consist of a gridded temperature field, density field and a wind speed field to make an N X M X 3 structure. Several 3-D structures could be tied together by an additional parameter, such as time, to make a 4-D structure.

Dimensions

Dimensions are used to relate data fields to each other and to the geolocation information. To be interpreted properly, each data field must make use of the two predefined dimensions: "XDim" and "YDim". These two dimensions are defined when the grid is created and are used to refer to the X and Y dimensions of the chosen projection. Although there is a limit of eight dimensions a data field in a Grid data set, it is not likely that the fields will need more than three: the predefined dimensions "XDim" and "YDim" and a third dimension for depth or height.

Projections

The projection is really the heart of the Grid. Without the use of a projection, the Grid would not be substantially different from a Swath. The projection provides a convenient way to encode geolocation information as a set of mathematical equations which are capable of transforming Earth coordinates (latitude and longitude) to X-Y coordinates on a sheet of paper.

The choice of a projection to be used for a Grid is a critical decision for a data product designer. There is a large number of projections that have been used throughout history. In fact, some

projections date back to ancient Greece. For the purposes of HDF-EOS, however, only a few projections are built with the software. These come from the General Coordinate Transformation Package (GCTP) package of projections available from the US Geological Survey (USGS). Projections include: Geographic, Interrupted Goode's Homolosine, Polar Stereographic, Universal Transverse Mercator, Space Oblique, and Lambert Azimuthal Equal Area. The full set of GCTP projections is contained in the SDP Toolkit library. (SDP Toolkit Users Guide for the ECS Project.) The full library can also be used in conjunction with the HDF-EOS interface.

For the purposes of the Grid interface, the data are assumed to have already been projected. The Grid interface allows the data producer to specify the exact GCTP parameters used to perform the projection and will provide for basic subsetting of the data fields by latitude/longitude bounding box

The producer's choice of a projection should be governed by knowledge of the specific properties of each projection and a thorough understanding of the requirements of the data set's users.

3.3.3.2 Applicability

The Grid data model is intended for data processed at a high level. It is most applicable to data at EOS processing levels 3 and 4. As an example, the ASTER & MODIS teams on EOS-Terra use Grid structures to store data.

3.3.3.3 The Grid Data Interface

The GD interface consists of routines for storing, retrieving, and manipulating data in grid data sets.

GD API Routines

All C routine names in the grid data interface have the prefix "GD" and the equivalent FORTRAN routine names are prefixed by "gd." The GD routines are classified into the following categories:

- *Access routines* initialize and terminate access to the GD interface and grid data sets (including opening and closing files).

- *Definition* routines allow the user to set key features of a grid data set.

- *Basic I/O* routines read and write data and metadata to a grid data set.

- *Inquiry* routines return information about data contained in a grid data set.

- *Subset* routines allow reading of data from a specified geographic region and by time.

The GD function calls are listed and are described in detail in the HDF-EOS Users Guides

File Identifiers

As with all HDF-EOS interfaces, file identifiers in the GD interface are 32-bit values, each uniquely identifying one open data file. They are not interchangeable with other file identifiers created with other interfaces.

Grid Identifiers

Before a grid data set is accessed, it is identified by a name which is assigned to it upon its creation. The name is used to obtain a *grid identifier*. After a grid data set has been opened for access, it is uniquely identified by its grid identifier.

3.3.3.4 Programming Model

The programming model for accessing a grid data set through the GD interface is as follows:

1. Open the file and initialize the GD interface by obtaining a file id from a file name.

2. Open OR create a grid data set by obtaining a grid id from a grid name.

3. Perform desired operations on the data set.

4. Close the grid data set by disposing of the grid id.

5. Terminate grid access to the file by disposing of the file id.

To access a single grid data set that already exists in an HDF-EOS file, the calling program must contain the following sequence of C calls:

```
file_id = GDopen(filename, access_mode);
gd_id = GDattach(file_id, grid_name);
<Optional operations>
status = GDdetach(gd_id);
status = GDclose(file_id);
```

To access several files at the same time, a calling program must obtain a separate id for each file to be opened. Similarly, to access more than one grid data set, a calling program must obtain a separate grid id for each data set. For example, to open two data sets stored in two files, a program would execute the following series of C function calls:

```
file_id_1 = GDopen(filename_1, access_mode);
gd_id_1 = GDattach(file_id_1, grid_name_1);
file_id_2 = GDopen(filename_2, access_mode);
gd_id_2 = GDattach(file_id_2, grid_name_2);
<Optional operations>
status = GDdetach(gd_id_1);
status = GDclose(file_id_1);
status = GDdetach(gd_id_2);
status = GDclose(file_id_2);
```

Because each file and grid data set is assigned its own identifier, the order in which files and data sets are accessed is very flexible. However, it is very important that the calling program

individually discard each identifier before terminating. Failure to do so can result in empty or, even worse, invalid files being produced.

It is permissible to have any number of Grid (Point, Swath) objects in a single HDF EOS file. PTopen () must be called to open each object (structure). It is o.k. to have more than one object open at a time.

3.3.3.5 GCTP Usage

The HDF-EOS Grid API uses the U.S. Geological Survey General Cartographic Transformation Package (GCTP) to define and subset grid structures. This section described codes used by the package.

GCTP Projection Codes

```
GCTP_GEO         (0)    Geographic
GCTP_UTM         (1)    Universal Transverse Mercator
GCTP_ALBERS             (3)    Albers Conical Equal_Area
GCTP_LAMCC       (4)    Lambert Conformal Conic
GCTP_MERCAT             (5)    Mercator
GCTP_PS          (6)    Polar Stereographic
GCTP_POLYC       (7)    Polyconic
GCTP_TM          (9)    Transverse Mercator
GCTP_LAMAZ       (11)   Lambert Azimuthal Equal Area
GCTP_HOM         (20)   Hotine Oblique Mercator
GCTP_SOM         (22)   Space Oblique Mercator
GCTP_GOOD        (24)   Interrupted Goode Homolosine
GCTP_ISINUS1     (31)   Integerized Sinusoidal Projection**
GCTP_BCEA        (98)   Behrmann Cylindrical Equal-Area (for EASE grid)
GCTP_ISINUS             (99)   Integerized Sinusoidal Projection*
```

*The Integerized Sinusoidal Projection was not part of the original GCTP package. It has been added by ECS. See *Level-3 SeaWiFS Data Products: Spatial and Temporal Binning Algorithms*. Additional references are provided in Section 2.

**In the new GCTP package, the Integerized Sinusoidal Projection is included as the 31st projection. The Code 31 was added to HDF-EOS for users who wish to use 31 instead of 99 for Integerized Sinusoidal Projection.

3.3.3.6 Example Usage of the Grid Interface

The following C program is an example of the usage of the HDF-EOS Grid interface.The program will create, define, and write a simple Grid data set to an HDF-EOS file. Examples of inquiry, reading and subsetting the file are also presented. Other C and FORTRAN examples are contained in the HDF-EOS Users Guide. (http://newsroom.gsfc.nasa.gov/sdptoolkit/toolkit.html)

Grid Structure Creation: SetupGrid.c

```c
#include "hdf.h"
#include "HdfEosDef.h"

/*
 * In this example we will (1) open the "GridFile" HDF file, (2) attach to
 * the "Grid1" grid, and (3) define the grid fields.
 */

main()
{

    intn            status, i, j;
    int32           gdfid, GDid1, GDid2, nflds;
    int32           dims[8], start[8], count[8];
    float32         fillval1=-7.0, fillval2=-9999.0, f32;
    float32         datbuf[100000];
    char            fieldlist[255];

    /*
     * We first open the HDF grid file, "GridFile.hdf".  Because this file
     * already exist and we wish to write to it, we use the DFACC_RDWR access
     * code in the open statement.  The GDopen routine returns the grid file
     * id, gdfid, which is used to identify the file in subsequent routines.
     */

    gdfid = GDopen("GridFile.hdf", DFACC_RDWR);

    /*
     * If the grid file cannot be found, GDopen will return -1 for the file
     * handle (gdfid).  We there check that this is not the case before
     * proceeding with the other routines.
     *
     * The GDattach routine returns the handle to the existing grid "Grid1",
     * GDid.  If the grid is not found, GDattach returns -1 for the handle.
     */

    if (gdfid != -1)
    {

     GDid1 = GDattach(gdfid, "UTMGrid");

        /* The next two calls create 2 SDSs.  The Merge Flag is turned
         * off. The SDS's are located in the "Data Fields" Vgroup.
         */
     status = GDdeffield(GDid1, "Pollution", "Time,YDim,XDim",
                    DFNT_FLOAT32, HDFE_NOMERGE);

     status = GDdeffield(GDid1, "Vegetation", "YDim,XDim",
                    DFNT_FLOAT32, HDFE_NOMERGE);

        /* The next call adds the required Metadata for a
         * field in the StructMetadata.0 Vdata.
         */
     status = GDwritefieldmeta(GDid1, "Extern", "YDim,XDim",
```

```
                            DFNT_FLOAT32);

    /* This call sets the fill value for the field Pollution, that
     * was defined above. A fill value is written to every element
     * that is not written by the first write for a field.
     */
 status = GDsetfillvalue(GDid1, "Pollution", &fillval1);
 GDid2 = GDattach(gdfid, "PolarGrid");

    /* The next two calls create 1 SDS.  The Merge Flag is turned
     * on. The SDS is located in the "Data Fields" Vgroup.
     */
 status = GDdeffield(GDid2, "Temperature", "YDim,XDim",
                  DFNT_FLOAT32, HDFE_AUTOMERGE);

 status = GDdeffield(GDid2, "Pressure", "YDim,XDim",
                  DFNT_FLOAT32, HDFE_AUTOMERGE);

    /* The next call creates a SDS.  The Merge Flag is turned
     * off. The SDS is located in the "Data Fields" Vgroup.
     */
 status = GDdeffield(GDid2, "Soil Dryness", "YDim,XDim",
                      DFNT_FLOAT32, HDFE_NOMERGE);

    /* The next call creates a SDS.  The Merge Flag is turned
     * on. This doesn't merge with the SDS created above, because
     * the fields have different datatypes and are not the same size
     * The SDS is located in the "Data Fields" Vgroup.
     */
 status = GDdeffield(GDid2, "Spectra", "Bands,YDim,XDim",
                  DFNT_FLOAT64, HDFE_AUTOMERGE);

    /* This call sets the fill value for the field Pollution, that
     * was defined above.
     */
 status = GDsetfillvalue(GDid2, "Pressure", &fillval2);
 }

 GDdetach(GDid1);
 GDdetach(GDid2);
 GDclose(gdfid);

 return;
}
```

Figure 3-12. show the HDF objected created by the above program.

UTMGrid

Vgroup: GridFile.hdf

Vgroup

Data Fields

Vgroup

Grid Attributes

Vgroup: PolarGrid.hdf

Vgroup

Data Fields

Vgroup

Grid Attributes

Vgroup: GeoGrid.hdf

Vgroup

Data Fields

Vgroup

Grid Attributes

Vgroup:GridFile.hdf

Vdata

HDFEOSVersion

Vdata

StuctMetadata.0
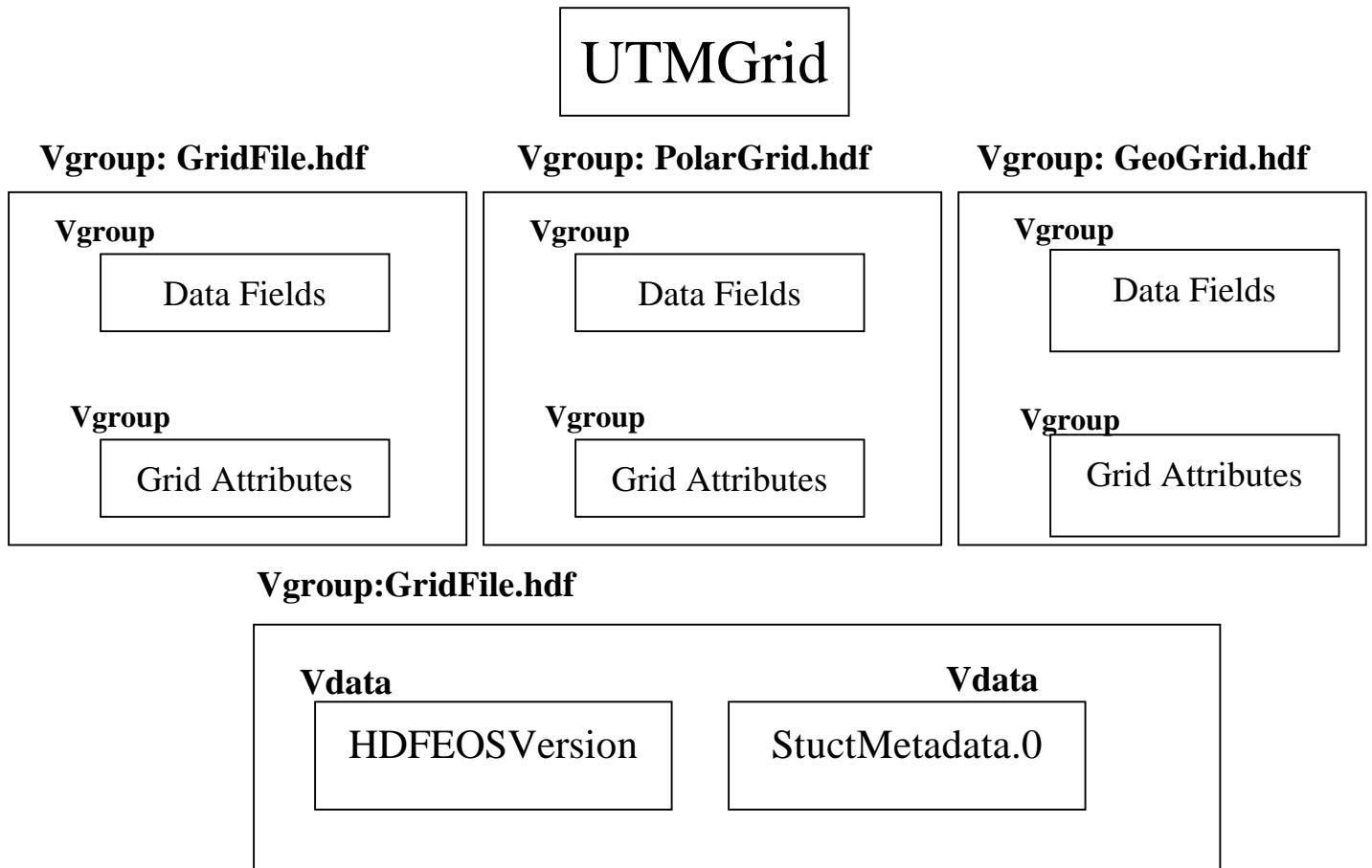
Figure 3-12. HDF Objects Created by Program: SetupGrid.c

Grid Field Definition: DefineGDflds.c

```
#include "hdf.h"
#include "HdfEosDef.h"

/*
 * In this example we will (1) open the "GridFile" HDF file, (2) attach to
 * the "Grid1" grid, and (3) define the grid fields.
 */

main()
{

    intn            status, i, j;
    int32           gdfid, GDid1, GDid2, nflds;
    int32           dims[8], start[8], count[8];
    float32         fillval1=-7.0, fillval2=-9999.0, f32;
    float32         datbuf[100000];
    char            fieldlist[255];

    /*
     * We first open the HDF grid file, "GridFile.hdf".  Because this file
```

```
 * already exist and we wish to write to it, we use the DFACC_RDWR access
 * code in the open statement.  The GDopen routine returns the grid file
 * id, gdfid, which is used to identify the file in subsequent routines.
 */

gdfid = GDopen("GridFile.hdf", DFACC_RDWR);

/*
 * If the grid file cannot be found, GDopen will return -1 for the file
 * handle (gdfid).  We there check that this is not the case before
 * proceeding with the other routines.
 *
 * The GDattach routine returns the handle to the existing grid "Grid1",
 * GDid.  If the grid is not found, GDattach returns -1 for the handle.
 */

if (gdfid != -1)
{

 GDid1 = GDattach(gdfid, "UTMGrid");

    /* The next two calls create 2 SDSs.  The Merge Flag is turned
     * off. The SDS's are located in the "Data Fields" Vgroup.
     */
 status = GDdeffield(GDid1, "Pollution", "Time,YDim,XDim",
                    DFNT_FLOAT32, HDFE_NOMERGE);

 status = GDdeffield(GDid1, "Vegetation", "YDim,XDim",
                    DFNT_FLOAT32, HDFE_NOMERGE);

    /* The next call adds the required Metadata for a
     * field in the StructMetadata.0 Vdata.
     */
 status = GDwritefieldmeta(GDid1, "Extern", "YDim,XDim",
                        DFNT_FLOAT32);

    /* This call sets the fill value for the field Pollution, that
     * was defined above. A fill value is written to every element
     * that is not written by the first write for a field.
     */
 status = GDsetfillvalue(GDid1, "Pollution", &fillval1);

 GDid2 = GDattach(gdfid, "PolarGrid");

    /* The next two calls create 1 SDS.  The Merge Flag is turned
     * on. The SDS is located in the "Data Fields" Vgroup.
     */
 status = GDdeffield(GDid2, "Temperature", "YDim,XDim",
                    DFNT_FLOAT32, HDFE_AUTOMERGE);

 status = GDdeffield(GDid2, "Pressure", "YDim,XDim",
                    DFNT_FLOAT32, HDFE_AUTOMERGE);

    /* The next call creates a SDS.  The Merge Flag is turned
     * off. The SDS is located in the "Data Fields" Vgroup.
     */
 status = GDdeffield(GDid2, "Soil Dryness", "YDim,XDim",
                        DFNT_FLOAT32, HDFE_NOMERGE);
```

```
        /* The next call creates a SDS.  The Merge Flag is turned
         * on. This doesn't merge with the SDS created above, because
         * the fields have different datatypes and are not the same size
         * The SDS is located in the "Data Fields" Vgroup.
         */
     status = GDdeffield(GDid2, "Spectra", "Bands,YDim,XDim",
                        DFNT_FLOAT64, HDFE_AUTOMERGE);

        /* This call sets the fill value for the field Pollution, that
         * was defined above.
         */
     status = GDsetfillvalue(GDid2, "Pressure", &fillval2);
    }

    GDdetach(GDid1);
    GDdetach(GDid2);
    GDclose(gdfid);

    return;
}
```

Writing Data to a Grid File: WriteGDflds.c

```
#include "hdf.h"

/*
 * In this example we will (1) open the "GridFile" HDF file, (2) attach to
 * the "UTMGrid", and (3) write data to the "Vegetation" field.  We will
 * then attach to the "PolarGrid" and write to the "Temperature" field.
 */

main()
{

    intn            i, j, status;
    int32           gdfid, GDid,start[3],stride[3],edge[3];
    float32         f32=1.0;
    float32         veg[200][120], temp[100][100];

    /* Fill veg array */
    for (i=0; i<200;i++)
     for (j=0; j<120; j++)
        veg[i][j] = 10+i;

    /* Fill temp array */
    for (i=0; i<100;i++)
     for (j=0; j<100; j++)
        temp[i][j] = 100*i+j;

    /*
     * Open the HDF grid file, "GridFile.hdf".
     */

    gdfid = GDopen("GridFile.hdf", DFACC_RDWR);

    if (gdfid != -1)
    {
```

```
    /*
     * Attach the "UTMGrid".
     */
    GDid = GDattach(gdfid, "UTMGrid");

    if (GDid != -1)
    {
      status = GDwritefield(GDid, "Vegetation",
                            NULL, NULL, NULL, veg);

      status = GDwritefield(GDid, "Vegetat",
                            NULL, NULL, NULL, veg);

      status = GDwriteattr(GDid, "float32", DFNT_FLOAT32, 1, &f32);
    }

    GDdetach(GDid);

    GDid = GDattach(gdfid, "PolarGrid");
    if (GDid != -1)
    {
        status = GDwritefield(GDid, "Temperature",
                              NULL, NULL, NULL, temp);
    }
    GDdetach(GDid);

    }

    GDclose(gdfid);

    return;
}
```

Extracting Information from the Grid Structure: InquireGrid.c

```
#include "hdf.h"
#include "HdfEosDef.h"

/*
 * In this example we will retrieve (1) information about the dimensions,
 * (2) the dimension mappings (geolocation relations), and (3) the grid
 * fields.
 */

main()
{

    intn          status, i;
    int32         gdfid, GDid1, ndim, nmap, nfld, rk, nt, nflds;
    int32         dims[32], rank[32], ntype[32];
    int32         n, strbufsize, sizes[16], GDid2;
    int32         xdimsize, ydimsize, dimsize, projcode, zonecode;
    int32         spherecode;
    float64       upleftpt[2], lowrightpt[2], projparm[16];
    char          dimname[1024], fieldlist[1024];
```

```
/*
 * Open the Grid File for read only access
 */

gdfid = GDopen("GridFile.hdf", DFACC_READ);

if (gdfid != -1)
{

 /* Attach the grid */

 GDid1 = GDattach(gdfid, "UTMGrid");
 GDid2 = GDattach(gdfid, "PolarGrid");
 ndim = GDinqdims(GDid1, dimname, dims);
 printf("Dimension list (UTMGrid): %s\n", dimname);
 for (i=0;i<ndim;i++) printf("dim size: %d\n", dims[i]);
 ndim = GDinqdims(GDid2, dimname, dims);
 printf("Dimension list (PolarGrid): %s\n", dimname);
 for (i=0;i<ndim;i++) printf("dim size: %d\n", dims[i]);

 dimsize = GDdiminfo(GDid1, "Time");
 printf("Size of \"Time\" Array: %d\n", dimsize);

 dimsize = GDdiminfo(GDid2, "Bands");
 printf("Size of \"Bands\" Array: %d\n", dimsize);

 status = GDgridinfo(GDid1, &xdimsize, &ydimsize,
                  upleftpt, lowrightpt);
 printf("X dim size, Y dim size (UTMGrid): %d %d\n",
        xdimsize, ydimsize);
 printf("Up left pt (UTMGrid): %lf %lf\n",
        upleftpt[0], upleftpt[1]);
 printf("Low right pt (UTMGrid): %lf %lf\n",
        lowrightpt[0], lowrightpt[1]);

 status = GDgridinfo(GDid2, &xdimsize, &ydimsize,
                  upleftpt, lowrightpt);
 printf("X dim size, Y dim size (PolarGrid): %d %d\n",
        xdimsize, ydimsize);
 printf("Up left pt (PolarGrid): %lf %lf\n",
        upleftpt[0], upleftpt[1]);
 printf("Low right pt (PolarGrid): %lf %lf\n",
        lowrightpt[0], lowrightpt[1]);

 status = GDprojinfo(GDid1, &projcode, &zonecode,
                  &spherecode, NULL);
 printf("projcode , zonecode (UTMGrid): %d %d\n", projcode, zonecode);
 printf("spherecode (UTMGrid): %d\n", spherecode);

 status = GDprojinfo(GDid2, &projcode, NULL,
                  &spherecode, projparm);
 printf("projcode (PolarGrid): %d\n", projcode);
 printf("spherecode (PolarGrid): %d\n", spherecode);
 for (i=0; i<13; i++)
     printf("Projection Parameter: %d %lf\n",i,projparm[i]);

 nflds = GDinqfields(GDid1, fieldlist, rank, ntype);
 if (nflds != 0)
 {
```

```
        printf("Data fields (UTMGrid): %s\n", fieldlist);
        for (i=0;i<nflds;i++)
           printf("rank type: %d %d\n",rank[i],ntype[i]);
    }

    nflds = GDinqfields(GDid2, fieldlist, rank, ntype);
    if (nflds != 0)
    {
        printf("Data fields (PolarGrid): %s\n", fieldlist);
        for (i=0;i<nflds;i++)
           printf("rank type: %d %d\n",rank[i],ntype[i]);
    }

    status = GDfieldinfo(GDid2, "Spectra", rank,
                     dims, ntype, dimname);
    printf("Spectra rank dims: %d\n",rank[0]);
    for (i=0; i<rank[0]; i++)
        printf("Spectra dims: %d %d\n",i,dims[i]);
    printf("Spectra dims: %s\n", dimname);

    n = GDnentries(GDid1, HDFE_NENTDIM, &strbufsize);
    printf("Number of dimension entries (UTMGrid): %d\n", n);
    printf("Length of Dimension List (UTMGrid): %d\n", strbufsize);

    n = GDnentries(GDid1, HDFE_NENTDFLD, &strbufsize);
    printf("Number of data fields (UTMGrid): %d\n", n);
    printf("Length of Field List (UTMGrid): %d\n", strbufsize);

    }
    GDdetach(GDid1);
    GDdetach(GDid2);
    GDclose(gdfid);

    return;
}
```

Reading from a Grid Structure: ReadGDflds.c

```
#include "hdf.h"

/*
 * In this example we will (1) open the "GridFile" HDF file, (2) attach to
 * the "UTMGrid", and (3) read data from the "Vegetation" field.
 */

main()
{

    intn          i, j, status;
    int32         gdfid, GDid;
    float32       f32=1.0;
    float32       veg[200][120];

    /*
     * Open the HDF grid file, "GridFile.hdf".
     */
```

```
        gdfid = GDopen("GridFile.hdf", DFACC_RDWR);


        if (gdfid != -1)
        {

         /*
          * Attach the "UTMGrid".
          */

         GDid = GDattach(gdfid, "UTMGrid");

         if (GDid != -1)
         {
            status = GDreadfield(GDid, "Vegetation",
                           NULL, NULL, NULL, veg);

                status = GDreadattr(GDid, "float32", &f32);

         }
        }

        GDdetach(GDid);
        GDclose(gdfid);

        return;
}
```

Subsetting a Grid Structure: SubsetGrid.c

```
  #include "hdf.h"
  #include <math.h>

  /*
   * In this example we will (1) open the "GridFile" HDF file, (2) attach to
   * the "PolarGrid", and (3) subset data from the "Temperature" field.
   */

  main()
  {
      intn            i, j, status;
      int32          gdfid, GDid, regionID, size, dims[8], ntype, rank;
      float32        *datbuf32;
      float64         cornerlon[2], cornerlat[2];
      float64        *datbuf64, upleft[2], lowright[2];

      /*
       * Open the HDF grid file, "GridFile.hdf".
       */

      gdfid = GDopen("GridFile.hdf", DFACC_RDWR);

      if (gdfid != -1)
```

```
    {

     GDid = GDattach(gdfid, "PolarGrid");

     if (GDid != -1)
     {
         cornerlon[0] = 57.;
         cornerlat[0] = 23.;
         cornerlon[1] = 59.;
         cornerlat[1] = 35.;
         cornerlon[0] = 0.;
         cornerlat[0] = 90.;
         cornerlon[1] = 90.;
         cornerlat[1] = 0.;

         regionID = GDdefboxregion(GDid, cornerlon, cornerlat);
         status = GDregioninfo(GDid, regionID, "Temperature", &ntype,
                       &rank, dims, &size, upleft, lowright);
         printf("size: %d\n",size);

         datbuf32 = (float32 *) calloc(size, 1);

         status = GDextractregion(GDid, regionID, "Temperature",
                            datbuf32);

         free(datbuf32);
     }
    }

    GDdetach(GDid);
    GDclose(gdfid);

    return;
}
```

This page intentionally left blank.

4. Related Topics

4.1   Introduction

In this Section, we discuss ECS metadata, ECS Browse specification and EOSView, the HDF-EOS browse tool.

4.2   ECS Metadata and the Science Data Processing Toolkit

In the preceding sections, we have provided an introduction to HDF and HDF-EOS. These software libraries can be used in stand-alone fashion to format science data. If products are to be inserted into ECS archives, ECS core metadata is required. The metadata is used in ECS archives to perform search and order functions. Services, such as subsetting can also be performed using metadata attributes stored in HDF-EOS files. This metadata varies from product to product. The more than 200 potential metadata attributes are described in Release B Earth Sciences Data Model, which is also known as the ECS data model. Minimal attributes, such as geographic 'bounding box' of a data granule, time stamp of the data, granule short name, are included.   Access to the metadata for data production and data applications is provided by the SDP Toolkit software library. (SDP Toolkit Users Guide for the ECS Project). The reader is referred to this document and references therein for a detailed discussion ECS metadata and access.

Because of the functional overlap of the HDF and HDF-EOS libraries, and the SDP Toolkit, it is important to understand what each one contains and how they are related. NCSA HDF is a subroutine library freely available as source code from the National Center for Supercomputing Applications. The basic HDF library has its own documentation, and comes with a selection of simple utilities.

HDF-EOS is a higher level library available from the ECS project as an add-on to the basic HDF library. It requires NCSA HDF for successful compiling and linking and is widely available (at no charge) to all interested parties. The basic HDF library is also be available from the ECS project.

The SDP Toolkit is a large, complex library of functions for use by EOS data producers. It presents a standard interface to Distributed Active Archive Center (DAAC) services for data processing, job scheduling, and error handling. It also contains common applications, such as geolocation, time/date conversion, unit conversion, coordinate transformation, Level 0 access. The SDP toolkit is also used by data producers working outside ECS DAACs. The Toolkit distribution includes source code for both HDF and HDF-EOS.

EOS instrument data producers will use the SDP Toolkit in conjunction with the HDF-EOS and HDF libraries. Of primary importance is process control and metadata handling tools. The former is used to access physical file handles required by the HDF and Toolkit libraries. The SDP Toolkit uses logical file handles to access data, while HDF (HDF-EOS) requires physical

handles. Users are required to make one additional call, using the SDP toolkit to access the physical handles. Please refer to the SDP Toolkit Users Guide for the ECS Project, for an example. This document gives examples of HDF-EOS usage in conjunction with the SDP Toolkit.

Metadata tools will be used to access and write inventory and granule specific metadata into their designated HDF structures. Please refer to the SDP Toolkit Users Guide.

We make an important distinction between core metadata and the structural metadata referred to in the software description. Structural metadata specifies the internal HDF-EOS file structure and the relationship between geolocation data and the data itself. Structural metadata is created and then accessed by calling the HDF-EOS functions. Core metadata, also known as inventory metadata, is used by ECS to perform archival services on the data. A copy will be written into HDF-EOS files as a text attribute, by SDP toolkit calls and another copy is written into database tables in the ECS archives. The two sets of metadata are not dynamically linked. However, the data producer should use consistent naming conventions when writing granule metadata when calling the HDF-EOS API. NCSA HDF libraries, on which HDF-EOS is based, is installed automatically with the SDP Toolkit installation script. Please refer to Appendix A for links to information pertaining installation and maintenance of the SDP Toolkit. These terms were also discussed in Section 3.3.2 in this document.

A summary of toolkit functionality is shown in Table 4-1.

Table 4-1. Summary of Toolkit Functions

Process Control	Provides connection logical file ID to physical data location, paths to staged data files and access to file attributes.
Error/Status Message Handling	Communication of user messages to a log file (Status Message File)
Generic File I/O	User access to staged data.
Metadata Access Tools	Formatting, reading, writing, updating metadata attributes. Writing is done to a pre-defined template called a Metadata Configuration File (MCF). The completed output has the extension ".met".
Time/Date conversion library	Toolkit internal time is TAI, conversion between many other systems is provided.
Ancillary data access	Several specialized data sets are available: 1km DEM, 3 Km DEM, 100m DEM, Land water mask. Access by geolocation provided.
Constants and Unit Conversions	Access to standard conversions
Geolocation tools	Transforms from instrument and platform coordinates to lat./long. e.g. Locate a pixel in geodetic of geocentric coordinates and find pierce point position and velocity.
Geo-coordinate system conversion	Generalized Coordinate Transform Package (GCTP) and other projections provided.

A Toolkit Programming Example: Writing and Reading Metadata in HDF Files

The following C program is an example of writing ECS metadata into an HDF (HDF-EOS) file. The program will also read metadata back out of the same file. The program assumes that a Process Control File (PCF) has been supplied. This file is a list of logical unit numbers associated with physical file handles. The program also assumes the availability of a Metadata Configuration File (MCF), which is a template of for writing metadata attributes. The MCF contains a list of parameters for which values are to be written by the code.

The metadata is stored as ASCII text in HDF text objects. In the program, these objects are callad 'coremetadata' and 'ProductMetadata'. Core Metadata are also know as inventory metadata. A copy is placed in ECS data bases for purposes of searching and ordering data. Product or archive metadata is also stored which the physical data files, but is not accessible by a database.

Details of these ancillary files, function descriptions and additional examples are found in the SDP Toolkit Users Guide for the ECS Project. The examples include FORTRAN programs.

```
 /*
  * C METADATA EXAMPLE
  *
  * In this example we show how to write ECS metadata to an HDF file.
  * Although not included in these examples, we recommend that after each
  * function call a conditional "IF" statement should be used to test for
  * errors.  We just include a simple write statement.
  * The file logical IDs are mapped to physical file names in a PCF file
  * and user should set environment variable PGS_PC_INFO_FILE to the user's
  * PCF file before runing the executable.
  */

 #include <PGS_MET.h>
 #include <stdio.h>
 #include <string.h>
 #include "hdf.h"
 #include <PGS_SMF.h>
 #include <PGS_PC.h>

 #define INVENTORY 1
 #define ARCHIVED 2
 #define OUT_FILE  22222      /* Logical ID for the HDF file where metadata
                                 to be written */
 #define MCF_FILE 10250       /* Logical ID for MCF file */

 main ()
 {
     char filename[PGSd_PC_FILE_PATH_MAX];
     char AttrName[256];
     char AttrValString[256];
     char *cptr;
     int32 sdid;
```

```
PGSt_integer version;
PGSt_MET_all_handles mdHandles;
PGSt_SMF_status status = PGS_S_SUCCESS;
PGSt_integer i;

char *asversion ="105";
char *svals[10];
char *svals2[5];
char *svals3[5] ={"1997","1998","1999",""};

/* Initialize MCF file into memory */

status = PGS_MET_Init ( MCF_FILE, mdHandles );

/*************************************************/
/* The following calls will set a few attributes */
/*      in the INVENTORYMETADATA section         */
/*************************************************/

/* set PGEVersion */

 status = PGS_MET_SetAttr ( mdHandles[INVENTORY], "PGEVersion",
                               &asversion );

 /* set InputPointer */

 strcpy ( AttrName, "InputPointer" );
 status = PGS_MET_SetAttr ( mdHandles[INVENTORY], AttrName, svals3 );

 /* set AdditionalAttributeName.1 and ParameterValue.1 */

 strcpy ( AttrName, "AdditionalAttributeName.1" );
 strcpy ( AttrValString, "string 1" );
 cptr = AttrValString;
 status = PGS_MET_SetAttr ( mdHandles[INVENTORY], AttrName, &cptr );

 strcpy ( AttrName, "ParameterValue.1" );
 strcpy ( AttrValString, "string 11" );
 cptr = AttrValString;
 status = PGS_MET_SetAttr ( mdHandles[INVENTORY], AttrName, &cptr );

 /* ===== Get attribute values set by previous calls ===== */

svals[0] = (char *) malloc(30);
svals[1] = (char *) malloc(30);
svals[2] = (char *) malloc(30);
svals[3] = (char *) malloc(30);
svals[4] = (char *) malloc(30);

for(i = 0; i<4; i++) strcpy(svals[i], "");
status = PGS_MET_GetSetAttr( mdHandles[INVENTORY],
                               "AdditionalAttributeName.1", svals);
```

```
for(i = 0; i<4; i++) printf("%s \n", svals[i]);


for(i = 0; i<5; i++) strcpy(svals[i], "");
status = PGS_MET_GetSetAttr( mdHandles[INVENTORY],
                             "ParameterValue.1", svals);
for(i = 0; i<5; i++) printf("%s \n", svals[i]);


for(i = 0; i<5; i++) strcpy(svals[i], "");
status = PGS_MET_GetSetAttr( mdHandles[INVENTORY], "PGEVersion", svals);
for(i = 0; i<5; i++) printf("%s \n", svals[i]);


/* Get HDF file name from PCF */

version = 1;
status = PGS_PC_GetReference ( OUT_FILE, &version, filename );

/* Open HDF File to write metadata to it */

sdid = SDstart ( filename, DFACC_RDWR );

/* Write INVENTORY metadata. This will write INVENTORY metadata to
   the hdf file and will also write INVENTORY metadata to *.met ASCII
   file */

status = PGS_MET_Write ( mdHandles[INVENTORY], "coremetadata",
                         sdid );

/* Write out ARCHIVED metadata */

status = PGS_MET_Write ( mdHandles[ARCHIVED], "ProductMetadata",
                         sdid );

/* Close HDF file */

status = SDend ( sdid );

/* Reading Attributes from the HDF file */
/* Get attribute value for "string 1" from the HDF file */

strcpy ( AttrName, "string 1");
for(i = 0; i<5; i++) strcpy(svals[i], "");
status = PGS_MET_GetPCAttr(OUT_FILE, 1, "coremetadata", AttrName,svals);
for(i = 0; i<5; i++) printf("%s \n", svals[i]);

/* Get attribute value for PGEVersion from the HDF file */

strcpy ( AttrName, "PGEVersion");
for(i = 0; i<5; i++) strcpy(svals[i], "");
status = PGS_MET_GetPCAttr(OUT_FILE, 1, "coremetadata",AttrName,svals);
for(i = 0; i<5; i++) printf("%s \n", svals[i]);
```

```
        /* Get attribute value for InputPointer from the HDF file */

        strcpy ( AttrName, "InputPointer");
        for(i = 0; i<5; i++) strcpy(svals[i], "");
        status = PGS_MET_GetPCAttr(OUT_FILE, 1, "coremetadata",AttrName,svals);
        for(i = 0; i<5; i++) printf("%s \n", svals[i]);

        /* Remove MCF from memory */

        PGS_MET_Remove ();

        /* Free allocated memory */

        free(svals[0]);
        free(svals[1]);
        free(svals[2]);
        free(svals[3]);
        free(svals[4]);
        return (0);
    }
```

4.3   ECS Browse Specification

4.3.1   Overview

Browse data granules are associated with ECS standard product granules. The former are used as aids to ordering the usually much larger latter granules. Browse products are also considered to be standard products. They contain minimal ECS metadata for purpose of rapid searches.   A more detailed description is given in the ECS Browse Granule Description.

Images are the preferred form of browse products in ECS because images can be compressed before storage and they are easily displayed and readily understood. Images conform with the intended purpose of browse which is to provide rapidly-accessible, on-line *representations* of data, not reduced-resolution samples of the data itself. However, tables (arrays) and text are also acceptable forms of browse. These are most useful as adjuncts to images.

All browse granules reside on rapid-access file storage. In order for the Browse service to satisfy its purpose of supporting on-line, rapid-access to browse, the overall size of a single browse granule is limited to *one megabyte*. HDF supports JPEG and other lossy and lossless compression methods. The one megabyte limit is on the object size *after* compression has been applied.

ECS Browse granules are to be composed of any combination of the vanilla HDF data objects illustrated in Figure 5-1. The use of 8-bit (RIS8) or 24-bit (RIS24) raster images best supports the intended purpose of providing a real-time, on-line aid to ordering. Extensions to basic images are also possible. *Text* may be used to provide more lengthy annotations than provided for in the ECS core metadata attribute: *BrowseDescription*. *Science Data Tables* would be used to provide either data values or to generate overlays, scatter plots, or linegraphs. Further

discussion of image, text and table browse objects, as well as recommended associated metadata, is contained in the ECS Browse Granule Description.
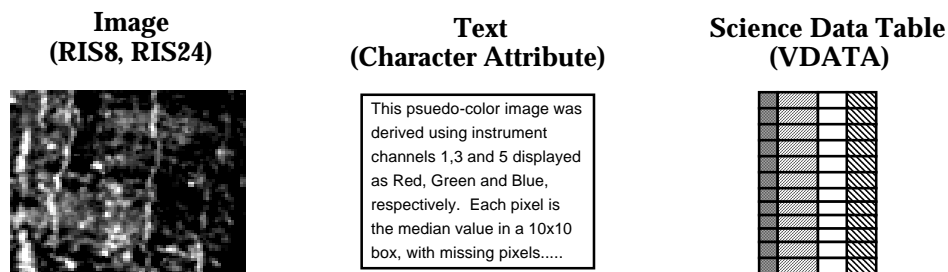
Image **(RIS8, RIS24)**	**Text** **(Character Attribute)**	**Science Data Table** **(VDATA)**
	This psuedo-color image was derived using instrument channels 1,3 and 5 displayed as Red, Green and Blue, respectively. Each pixel is the median value in a 10x10 box, with missing pixels.....	

Figure 4.1. Browse Package Data Objects

The simplest form an ECS browse granule can take is an HDF file containing either an 8-bit (RIS8) or 24-bit (RIS24) raster image. Command-line utilities are available from NCSA for converting raw 8-bit or 24-bit images and JPEG files into generic HDF image files, with options to include an associated color palette. The recommended size is 300 by 300 pixels. This pixel size is a good target for the size of browse images. Larger images are acceptable, within the 1Mb file size limit, but with the ECS Data Gateway Client (EDG), the user may need to scroll around to view the entire image. This Client is the users view on ECS data collections. It is a search and order tool.

No geolocation services such as latitude-longitude under cursor location, overlay of coastlines, or a latitude-longitude graticule are provided on generic HDF browse granules.

If the ECS Science Data Processing (SDP) Toolkit is being used to generate browse granules, the Toolkit's metadata routines can be used to write ECS core metadata attributes. In the Science Computing Facility (SCF) environment it can also be used to write any other product-specific metadata.

4.3.2 Browse Package Guidelines

The purpose of browse data in the ECS is to serve as a real-time on-line aid to ordering full data product granules. As such, its design must provide for fast access, small size, and simple organization that leads to easy interpretation. Our assumptions about these browse packages include the following:

- We expect that every Standard Data Product granule will have a corresponding browse package. This browse package can contain many different objects, as described below.

- We expect that instrument teams will generate these browse packages as part of their normal production processing.

- We expect that browse data packages will be static, not dynamic. That is; they will be generated once and archived rather than being produced in response to each user's request.

- We finally expect that some browse packages will consist of an "example" dataset plus additional data (e.g., cloud cover) rather than a subsampled product granule.

Objects that we plan to support in browse packages include the following (along with their associated metadata):

- Image: 8-bit raster image with palette, or 24-bit raster image.

- Table: Science Data Table meant to be displayed as numbers.

- Plot: Science Data Table meant to be displayed graphically.

- Text: ASCII Text (Plain or Formatted).

- Animation: A series of raster images.

Most common browse packages will consist of a single Image generated by a subsampling algorithm applied to a large array (along with associated metadata). Note that multidimensional array is *not* a supported browse package component. This prohibition will hopefully encourage the use of browse for visualizations of data, and not for delivering data itself.
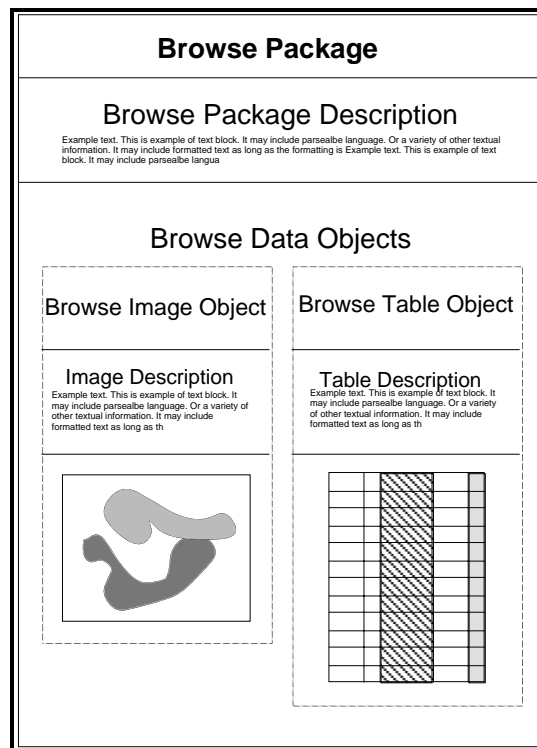

Browse Package

Browse Package Description

Example text. This is example of text block. It may include parsealbe language. Or a variety of other textual information. It may include formatted text as long as the formatting is Example text. This is example of text block. It may include parsealbe langua

Browse Data Objects

Browse Image Object

Image Description
Example text. This is example of text block. It may include parsealbe language. Or a variety of other textual information. It may include formatted text as long as th

Browse Table Object

Table Description
Example text. This is example of text block. It may include parsealbe language. Or a variety of other textual information. It may include formatted text as long as th

Figure 4-2. A Browse Data Package

175-WP-001-002

We further propose the following general specifications for browse data products:

- A maximum overall package size of around one megabyte. A package much larger than this will tend to be less an interactive aid to ordering of data, and more of a data product in and of itself. Note that browse images can be compressed using lossy compression to help them fit under this limit (it can be lossy since all we care about is the image appearance, not the image data values). We further propose that no particular component of the browse package exceed the one megabyte limit.

- A target image size of around 620 X 620 pixels for images. Larger images will be allowed where absolutely required, if they can fit under the one megabyte limit when compressed.

- 8-bit raster images with an associated palette will be limited to say 150 entries in the palette. This limitation will reduce the chance that the image display will be substantively degraded when displayed on a system with a reduced color range. In addition, we would like fixed colors (colors used for burned in overlays, etc.) to be stored only in the highest and lowest palette entries (0 and 255, for example). This is because when a color palette is compressed to fit into a smaller color range, all entries except the first and last one may move by an entry or two.

4.4   EOSView: An HDF-EOS 'Browse Tool'

4.4.1 Introduction

The HDF-EOS team has developed 'EOSView', a tool for examining and verifying HDF and HDF-EOS data files.   The contents of HDF files are displayed and individual objects can be selected for display. Displays include raster images, datasets in tables, pseudocolor images of datasets, attributes, and annotations.   Simple animations can be performed for a file with multiple raster images.   A unique interface has been provided for handling HDF-EOS data structures. The Swath/Point/Grid interface uses only HDF-EOS library calls.   The EOSView user will not see the underlying HDF structures but will be prompted for what part of the HDF-EOS object they wish to view.
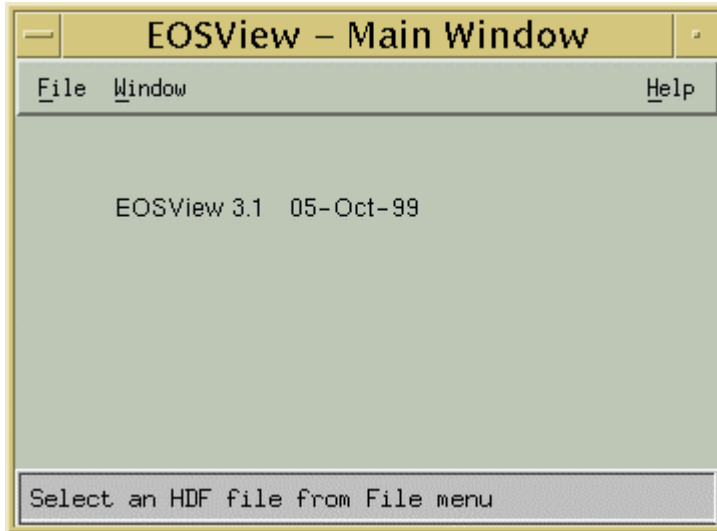
Figure 4-3. EOSView Main Window

4.4.2 EOSView Features

From the EOSView main window (figure 4-3) the user may select a file for viewing and then display the contents of the HDF file in a File Contents Window (figure 4-4).

Figure 4-4. EOSView File Contents Window

Item in the File Contents Window list the top-level HDF and/or HDF-EOS objects in the file. These items are selectable and will be displayed in their proper form. Vgroups listed in the

contents window will be expanded in a tree format to allow the user to select objects stored inside of the Vgroup grouping. In the above example the user has selected the file cosmic.hdf. This file contains an image of size 625 pixels by 427 pixels, and a palatte. Other items which may be displayed in the list are Vgroups, Vdatas, 24-bit images, Scientific Data Sets or Numeric Data Groups, Swath/Point/Grid objects, file identifiers or descriptors. If the user were to double-click on the Image in the above example the EOSView Image Display Window (figure 4-5) would then appear.
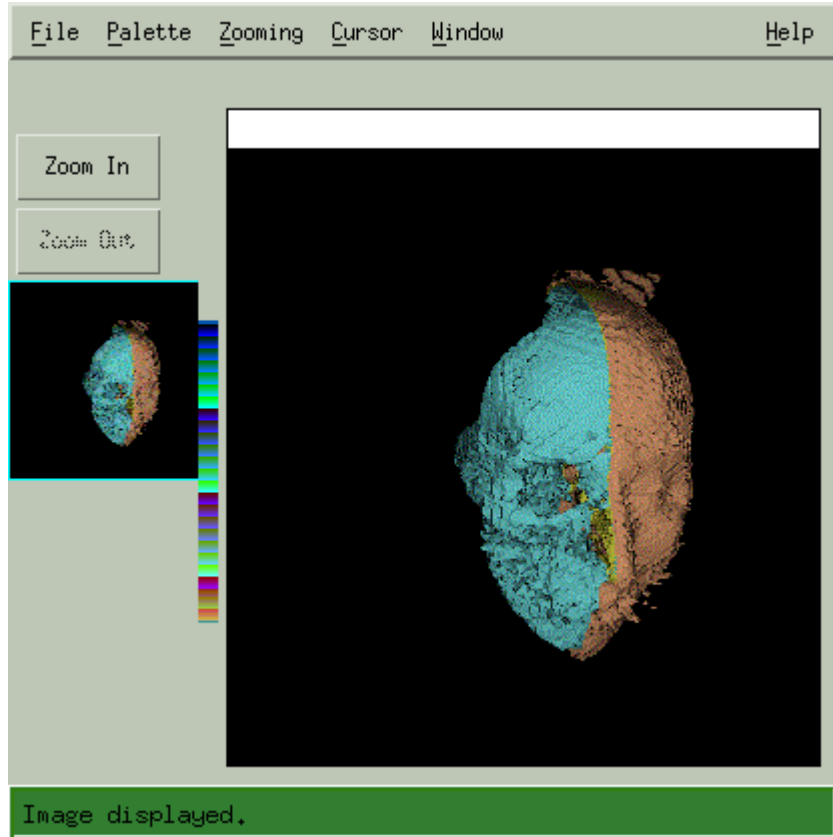

Figure 4-5. EOSView Image Display Window

The EOSView Image Display Window contains a host of features. The user is allowed to select multiple palettes, zoom in and out using two (2) zooming methods, panning of zoomed images, and cursor tracking and placing. These features remain the same for pseudocolor images created from a numeric data set and for 24-bit images.
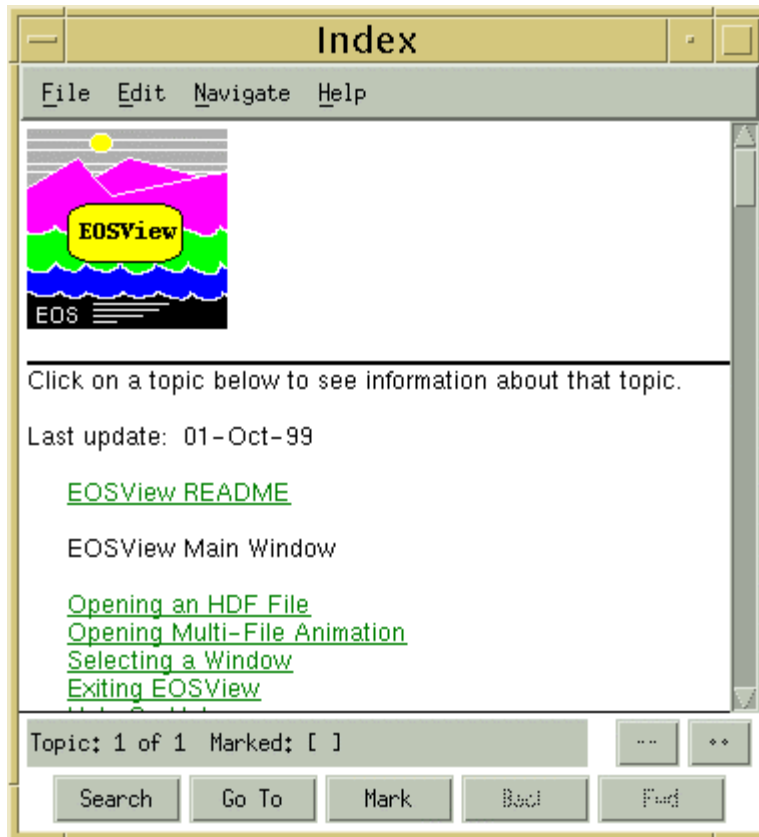
Figure 4-6. EOSView Help Display

The user will notice that on each EOSView window menu bar the Help option is always present. EOSView has an on-line hypertext help system that can help the user navigate through each display in EOSView.   Underlined phrases in the help system are selectable links to more extensive help.   The user may 'mark' help pages for a quicker return and a search may be performed based on keywords

The EOSView Table Display (figure 4-7) is the way in which EOSView can display tabular data commonly associated with numeric data sets.

Figure 4-7. EOSView Table Display

The EOSView table display has several features of note. For numeric data the user may convert the table to an image or a plot (figure 4-8). Many tables in HDF files may be quite large so EOSView has added the option to allow the user to 'Jump To' a specific row number. The data in this table may be saved to either a binary or an ASCII file. The ASCII file is written to meet the HDF ASCII Interchange Format (HAIF).

Figure 4-8. EOSView Text Display

EOSView has the ability to display attribute data in a text display window (figure 4-8). The text display window is also used to display File Indentifier and File Descriptor Data.

In conclusion, EOSView has become a valuable tool to the HDF-EOS user community in verifying and browsing HDF and HDF-EOS data files. It has the proven ability to handle large files, handle multiple files, and to breakdown HDF-EOS objects into verifiable data.

References

HDF

HDF Reference Manual v4.1r4, UIUC, 2000

HDF User's Guide v4.1r4, UIUC, 2000

HDF Specification and Developer's Guide v4.1r4, UIUC, 2000

For more information on HDF and User Support, contact:

http://hdf.ncsa.uiuc/edu

E-mail:hdfhelp@ncsa.uiuc.edu

HDF-EOS

HDF-EOS Users Guide for the ECS Project, Volume 1: Overview and Examples, 170-TP-510-001, Raytheon Systems Co., Upper Marlboro, MD

HDF-EOS Users Guide for the ECS Project, Volume 2: Function Reference Guide, 170-TP-511-001, Raytheon Systems Co., Upper Marlboro, MD

HDF-EOS Design for the ECS Project, 456-TP-013-001, Raytheon Systems Co., Upper Marlboro, MD

Software and documents available at:

http://newsroom.gsfc.nasa.gov/sdptoolkit/toolkit.html

ECS Science Data Processing Toolkit

SDP Toolkit Users Guide for the ECS Project, 333-CD-600-001, Raytheon Systems Co., Upper Marlboro, MD

A Data Formatting Toolkit for Extended Data Providers to NASA's Earth Observing System Data and Information System (V1), Raytheon Systems Co., Upper Marlboro, MD - Note: This is a subset of the SDP toolkit, containing only metadata access, time and date conversion tools.

Software and documents available at:

http://newsroom.gsfc.nasa.gov/sdptoolkit/toolkit.html

Additional information and help:

pgstlkit@eos.east.hitc.com.

Browse Specification

ECS Browse Granule Description, 170-TP-004-002, Raytheon Systems Co., Upper Marlboro, MD

ECS Metadata

SDP Toolkit Users Guide for the ECS Project, 333-CD-600-001, Raytheon Systems Co., Upper Marlboro, MD

Release B-1 Earth Science Data Model, 420-TP-017-001, Raytheon Systems Co., Upper Marlboro, MD

The Role of Metadata in EOSDIS, 160-TP-013-001, Raytheon Systems Co., Upper Marlboro, MD

EOSView

EOSView Users Guide for the ECS Project, 445-TP-006-002, Raytheon Systems Co., Upper Marlboro, MD

Appendix A. Obtaining Software

A.1   Obtaining the HDF Library and Documentation

Access to NCSA software and documentation is available at http://hdf.ncsa.uiuc.edu.

A.2   Obtaining HDF-EOS and the SDP Toolkit

Access to HDF-EOS, SDP Toolkit and EOSView is found at:

http://newsroom.gsfc.nasa.gov/sdptoolkit/toolkit.html

The site contains explicit instructions for downloading the software.

175-WP-001-002

This page intentionally left blank.

Appendix B. Additional HDF Topics

In this Appendix, we discuss various issues involved with the HDF.

B.1   Data Interleaving

Data interleaving is an important subject from the perspective of efficiency of data access and, sometimes conversely, from the perspective of organizational simplicity. Efficiency is important at the lower processing levels, where data volumes are large and the number of users is small. Organizational simplicity is important at higher processing levels, where the needs of large numbers of users are a major concern. The interleaving method chosen by the data producer effectively dictates the best access method for the users, including further processing software and user visualization tools.

In HDF, interleaving is handled separately for each type of data object for which it is necessary. For instance, interleaving for SDSs (n-dimensional arrays of scalars) are handled differently from interlacing for RIS24s (24-bit raster images). The HDF library allows the user/producer to control the interleaving of SDSs, RIS24s, and Vdatas (tabular data). Note that in the HDF documentation; the term *interlacing* is used to refer to the concept of interleaving. The two terms will be used interchangeably in this document.

B.1.1 SDS Interleaving

The interleaving of an SDS is implied by the order of the dimensions as given to HDF in the creation of the SDS. The rule is that HDF will store the SDS on disk in the order specified by the dimensions given at creation time, using a row-major interpretation. For example, an SDS with dimensions defined as 2 by 3, or [2, 3] will be stored as an array of 2 rows of 3 columns each. Similarly, an SDS of dimensions [4, 5, 6] is regarded as an array of 4 planes of 5 rows of 6 columns.

C programmers will recognize this as 'normal' array order. FORTRAN programmers, however, should take note that this is the reverse order from the traditional column-major ordering used in FORTRAN programming. Given these rules for the specification of interleaving, a data producer can cause an SDS to be stored in any conceivable order, simply by re-arranging the order of the dimensions at creation time.

B.1.2 RIS24 Interleaving

The RIS24 interface will probably not be extensively used in the ECS, but it does provide flexible interleaving facilities, so we will discuss it here. In memory, a 24-bit raster image is implemented as an array of type *uint8* (unsigned char) with dimensions width by height by 3 (depth), in some order. As in the case of the SDS, the exact ordering of the dimensions is at the heart of the interleaving question.

175-WP-001-002

By default, HDF assumes that the user will be working with images interleaved by pixel. This means that an image that is 200 pixels wide by 100 high is equivalent to an array of bytes with dimensions [100, 200, 3] and that the red, green, and blue values that make up an individual pixel are stored contiguously.

You may also choose to interleave by scan-line or by scan-plane. Choosing interleaving by scan-line will require that the user declare arrays for the image mentioned above with dimensions [100, 3, 200], while scan-plane interleaving leads to dimensions of [3, 100, 200].

Regardless of the interleaving method used to *store* an image, a user may request to *read* an image using any of the interleaving schemes. Of course, there is a performance penalty for any reorganization of the data.

B.1.3 Vdata Interleaving

There are two interleave options for Vdatas: FULL_INTERLACE and NO_INTERLACE. The terms are defined as follows:

> FULL_INTERLACE — The first value from each field is collected into a record. Successive records contain subsequent values from each field. This can also be called record-oriented storage, since whole records are stored contiguously.

> NO_INTERLACE — All data for the first field in the Vdata is stored contiguously in the file, then all the data for the second field, and so on. This can also be called field-oriented storage.

The method of interleaving is defined for a particular Vdata at creation time with a call to a special library function. If no interleaving method is specified, then FULL_INTERLACE is assumed. The interleaving method used is encoded in the Vdata.

As with the case of RIS24s, you can request that the library read Vdatas using either interleaving method. But again, you must pay a penalty in extra processing time to reorganize the data during the read operation.

B.2 Subsetting

The term "subsetting" has come to mean two very different processes in the context of the ECS system. The first meaning is where a user would like to extract a particular component from a standard data product granule. For example, she may want to look at just sensor channel 3. The corresponding data products include all channels in the same file, but she does not want to waste time downloading information she does not want. She would then request a subset of the standard data product, which delivers to her a file containing but that single array representing sensor channel 3. This sort of subsetting is easily handled by the existing HDF mechanisms.

The second type of subsetting is where a particular region of a data element is requested. For example, a user may want to specify a latitude/longitude subrange within which to provide data. In HDF, this situation will need to be handled differently for each type of data element. HDF provides the capability to read or write any subpart of any data element, but the logic required to

decide which subpart is needed is highly dependent on the nature of the data element. Every effort is made to provide intelligent subsetting of each datatype. Specifically, each of the geolocated datatypes are capable of being subsetted by latitude/longitude box.

B.3   File Sizes

HDF imposes a two gigabyte limit (2,147,483,648 bytes) on the size of a single, self-contained HDF file and on the individual physical files that make up a single logical HDF file. Some teams say that this limit is too low for their needs. However, we are primarily concerned with user access to data. A file approaching HDF's two gigabyte limit far exceeds the capabilities of the average current or near future science data user's computing facilities. We therefore do not consider the two gigabyte limit to be unreasonable

We would even propose that user delivered HDF files not exceed around 512 megabytes in size. We further suggest that files be kept under 200 megabytes, wherever possible.

Note that a newer version of HDF, HDF5 uses 64-bit addressing, which breaks the 2 GB size limit. HDF5 will be described in separate documents. Given difficulties in managing files this large on end-user systems, we still recommend smaller (< 2GB) files.

B.4   Compression Methods

Compression algorithms come in two different types:

- *Lossless* — A compression method that allows the original data to be reconstructed exactly as it was before the compression algorithm was applied.

- *Lossy* — A compression method that can construct a reasonable facsimile of the original input data from the compressed data.

Lossless compression will be used for most EOS data, because the loss of perfectly valid information cannot be tolerated. The only exception so far noted is browse data; because browse data is only meant to be a *representation* of the data as an aid to ordering.

HDF-EOS will incorporate all HDF compression methods as they become available.

B.5   Performance Issues

B.5.1 Many Data Objects

During Pathfinder and Version 0 efforts (which used HDF 3.2), it was found that HDF files containing many data objects caused performance problems. The main symptom was extreme slowness in opening the file, although there was also a slowdown in accessing individual data elements.

The HDF libraries have greatly improved the situation. The multifile SDS and Vdata interfaces dramatically reduce the number of times the file is opened and closed. These interfaces are capable of concurrently dealing with more than one open file, rather than covertly performing

multiple file opens and closes like the older interfaces. The behavior of these interfaces effectively sidesteps the issue of slow file opens. This new library also implemented a more efficient, tree-based lookup scheme to locate data elements more quickly.

Although major improvements have been realized in access to files with many data objects, it is still wise to avoid creating files with more than about 500 individual elements. The HDF utility program 'hdfls' can be used to aid in counting the number of data objects in a file. When invoked with the '-l' option, hdfls will produce a list containing roughly one line for each data element in the file. The output of hdfls can be piped into the UNIX utility 'wc' or a similar program to count the number of data elements.

It is important to note that the number of data elements in an HDF file can greatly exceed the number of data objects, such as SDSs. For example, a single SDS can easily generate a dozen data elements that refer to its attributes, its numerical scales, and so on.

B.5.2 Large Data Objects

Many of the proposed data products for EOSDIS require the use of very large data elements. Specifically, individual Vdatas and SDSs for some products will likely reach tens of megabytes in size. When a data element reaches such a size, it becomes important to optimize I/O performance for that element.

One area where major performance gains can be realized is in data buffering. Buffering is where you fill up a memory buffer and then periodically write the buffer to disk. You optimally should have a large buffer, but not so large as to force the buffer to swap to disk, negating the point of having the buffer in the first place.

When doing buffering on HDF data elements, it is best to use the extended tag feature called *linked-blocks* (see section 2.3.7), where the directory entries point not to the data, but to a list of blocks containing the data. HDF provides application programming level access to functions that can produce arbitrarily long chains of data blocks linked to form a large, expandable data element.

Each block that makes up a linked-block element is also a full-fledged (but somewhat hidden) data element. The implication here is that, if you write out an HDF file that contains only a single linked-block element and you ask the library to use a small block size, you may end up creating several thousand data elements, even though you only really wanted one. The key is to ask for a 'reasonable' block size.

Here are some general guidelines in choosing block sizes:

• One large block containing the entire data element yields the best performance. Of course, you will not always know how big that block should be, so it could be impractical to follow this rule in some cases.

• Choose a block size that makes a compromise between minimizing the number of blocks needed to write the expected nominal-size element (large block size) and minimizing the

wasted space left in the last block, which may be only partially filled with actual data (small block size).

B.6   Fill Values

HDF-EOS provides functions for establishing a common fill value for missing data across a structure, eg. Swsetfillvalue() is used to define a fill value. The filled data pixels can then be compressed out.   This value can also be used to replace bad or suspect data in a data structure. There is no standard among HDF-EOS users ads to what these values should be, however. It is up to the data producer to decide.

This page intentionally left blank.

Glossary and Acronyms

8-bit Raster .... Refers to a Raster Image where each pixel is represented by a single byte. This allows each pixel to be displayed with one of only 256 possible colors (using an associated color table). Known in HDF as a 'RIS8'.

24-bit Raster................... Refers to a Raster Image where each pixel is represented by three bytes, one each for red, green, and blue. This allows each pixel to be displayed with one of over 16 million possible colors. Known in HDF as a 'RIS24'.

Annotation .... In HDF language, a plain text data element that can be used to describe or identify any other data element, an entire file, or a specific tag number.

API .... Stands for <u>A</u>pplication <u>P</u>rogramming <u>I</u>nterface. A set of functions designed for use by applications programmers. Often used interchangeably with the term 'subroutine library', or especially in this project, 'toolkit'.

Arrays of Records .... A proposed structure where every element in an array is not a number but a record. One could think of Vdatas as a one dimensional version of an array of records.

ASCII Text .... Plain textual data stored using the <u>A</u>merican <u>S</u>tandard <u>C</u>ode for <u>I</u>nformation <u>I</u>nterchange.

Attribute......................... In HDF language (borrowed from netCDF), a text or binary data object used to store a single value or a list of values. Attributes can currently be associated with an SDS or an entire file, and are most often used for storing metadata.

Binning .... Describes the process of combining data taken at various locations and placing the information, properly interpolated, into bin locations that are defined on a particular grid.

Bitmap Image.................. A synonym for *Raster Image*. Bitmap comes from the fact that every pixel location has associated with it a string of bits (usually 8 or 24).

Browse Package .... In the EOS world, refers to a collection of images, tables, or text that is meant to be a *representation* of a data product. Browse packages are designed to be an *aid to ordering data*, and not to be data in itself.

CCSDS .... <u>C</u>onsultive <u>C</u>ommittee for <u>S</u>pace <u>D</u>ata <u>S</u>ystems

CDF .... Stands for <u>C</u>ommon <u>D</u>ata <u>F</u>ormat. A standard data format developed at Goddard Space Flight Center.

Color Palette .... A synonym for *Color Table*.

Color Table .... A table used to map pixel values in raster images to actual colors. Color tables are usually 256 entries in size, corresponding to the 256 possible values in an 8-bit raster image. Each entry in the table consists of three numbers: a red, a green, and a blue value to uniquely specify the color for that pixel value.

Computer-Readable .... In this document, computer-readable refers to fields that a computer program can read, but not necessarily interpret. An example would be where a program could display stored latitude/longitude values from a file, but not necessarily use those values in further calculations to say display a map grid.

DAAC .... <u>D</u>istributed <u>A</u>ctive <u>A</u>rchive <u>C</u>enter

Data Descriptor .... An internal 12-byte HDF structure containing a *Data Identifier*, an *Offset*, and a length that uniquely identifies and locates a data element within an HDF file. The HDF directory consists of a list of Data Descriptors (DDs).

Data Dictionary .... Refers to a record that contains detailed information about keywords; especially keywords used in metadata. An example would be an entry for 'Satellite_Name' that enumerates a keyword title ("Satellite Name"), a field type ("text"), a field width ("10 characters"), and perhaps allowed values ("EOS-AM, Tropical Rainfall Measuring Mission (TRMM), EOS-PM", etc.).

Data Element .... Refers to the individual components of *Data Objects* within an HDF file. For example, a Data Object of datatype 'scientific dataset' would consist of several data elements: one for each of the attributes, another for the array itself, and so on.

Data Granule .... In the EOS world, refers to a particular instance of a *Standard Data Product*.

Data Identifier .... In the HDF world, refers to combination of a *Tag* and a *Reference Number* that uniquely identifies a *Data Element* within an HDF file.

Data Location .... In this document, data location refers to values that are meant to be used to locate a particular data value. Examples of data locations would be X, Y, Z values, or Latitude, Longitude, Altitude values.

Data Model .... A description of the conceptual data model of a particular scientific data format. For example, one netCDF data model is of a series of records, each a different time. Each record contains a series of n-dimensional arrays, each a different physical parameter. Contrast Data Model with *Disk Format*, which defines the actual physical organization of the disk files written with that scientific data format.

Data Object .... A particular instance of a *Datatype*. For example, an HDF file may consist of a series of data objects, each of a different datatype (raster, n-dimensional array, and so on).

Data Product .... Synonym for *Standard Data Product*.

Datatype.......................... Refers to classes of data structures such as *Grids*, *Swath* structures, *Science Data Tables*, and so on that will be supported in HDF-EOS. Often there will be an exact mapping of an HDF-EOS datatype to an HDF data object. However, there are cases where an HDF-EOS datatype is made up of several HDF data objects grouped together.

DD Block......................... A physically contiguous group of *Data Descriptors* (DDs). One or more DD Blocks make up the DD List.

DD List........................... The DD List, which is made up of one or more linked *DD Blocks*, contains every DD entry in a HDF file. The DD List can be considered as the internal HDF file directory.

DD .... See *Data Descriptor*.

Dimension Scales............ Refers to the series of one dimensional arrays associated with a particular *Multidimensional Array* (SDS). These arrays, one per dimension of the SDS, list the data location values (latitude, longitude, altitude, for example) for each dimension in the array. Note that this way of describing the data locations for an array only works for regularly gridded data.

Disk Format .... A description of the actual physical byte values and locations stored in a disk file written in a particular scientific data format. Contrast with *Data Model*, which refers to the conceptual and not physical organization.

ECS .... Stands for <u>E</u>OSDIS <u>C</u>ore <u>S</u>ystem. Refers to the core software and hardware system used to support EOSDIS.

EOS .... Stands for <u>E</u>arth <u>O</u>bserving <u>S</u>ystem.

EOSDIS .... Stands for <u>E</u>arth <u>O</u>bserving <u>S</u>ystem <u>D</u>ata and <u>I</u>nformation <u>S</u>ystem. Refers to the ground based data archive and management system for EOS.

EOSView......................... Our multi-platform HDF-EOS analysis and visualization application. Also called an 'HDF-EOS cracker tool', for its ability to display HDF-EOS file contents and organization.

Equal-Angle Grid............ A way of storing geolocated data where the size of each bin location is defined by fixed degree changes in latitude and longitude. The problem with this method of storing data is that each degree of longitude represents very different distances at high and low latitudes.

Equal-Area Grid..............A way of storing geolocated data where each the size of each bin location are defined by fixed distances. The problem with this method of storing data is that the number of bins has to be different for every latitude.

ESDIS..........................Earth Science Data and Information System

ESDT............................Stands for Earth Science Data Type. Refers to a higher level structure of EOS data than CSDTs. For the most part, ESDT refers to particular *classes* of *Standard Data Products*: Every standard data product is of a particular ESDT.

Extended Tags................An extended tag DD does *not* point directly to the data, but to a data element *defining where the data is and how it is stored*. This data object *may* point to the beginning of a linked list of data blocks that contain the entire data record. Alternatively, the extended tag record could define the data element as being stored in an *External Element* in another disk file.

External Elements...........An external element is an HDF data element that is stored not inside the physical HDF file, but as a separate physical file.

FGDC ....Stands for Federal Geographic Data Committee. The FGDC has proposed a set of metadata data standards, for possible use in geolocated earth data. These standards include supported keywords, along with allowed values.

FITS.............................Stands for Flexible Image Transport System; a data format popular in the astronomy field.

ftp ....file transfer protocol

Geolocation...................Refers to data locations that are specific to physical locations on the Earth, or on another planetary body. Geolocation is usually (but not always) specified in terms of latitude, longitude, and altitude.

Granule ....Synonym for *Data Granule*.

Grid ....Refers to a particular example of a *Gridding* scheme.

Grid Structure ....Refers to an HDF-EOS datatype that will be designed to support gridded data, by making the geolocation information for the grid data *computer-comprehensible*.

Gridding.........................For EOS, refers to schemes for dividing locations on the Earth or on a projection of the Earth into many bins or cells. Each bin has a unique spatial location on the Earth. Although independent of projection, gridding schemes should be chosen to map well to a given projection: for example, projections that favor one area of the Earth in some way should have many bins in that location.

GSFC.............................Goddard Space Flight Center

GUI................................Stands for Graphical User Interface.

HDF...............................Stands for Hierarchical Data Format. The format was developed and is maintained by NCSA at UIUC.

HDF-EOS ....A shorthand notation for our proposal of establishing EOS conventions for the organization of HDF files used by the EOS system and the software library that will implement and enforce them.

HTML.............................Hypertext Markup Language

http................................hypertext transport protocol

IDL ....Stands for Interactive Data Language. A cross-platform data manipulation and visualization tool developed by Research Systems, Incorporated.

Image..............................Synonym for *Raster Image*.

Indexed Pointer ....A data value that by HDF-EOS convention refers to a particular data element, or a particular location within a data element. Can be used to create links between tables and data elements.

Interlacing.......................Refers to the process of deciding how to organize the storage of an array: in particular, deciding which data locations will be close to each other physically. For example, a 3D array that was interlaced by altitude would have every 2D altitude plane stored together.

Interleaving.....................Synonym for *Interlacing*.

ISCCP............................International Satellite Cloud Climatology Project

JPEG...............................Stands for Joint Photographic Experts Group. A lossy compression method for 24-bit raster images. The method has been expanded to include 8-bit images, as well.

Length.............................In HDF, the number of bytes that comprise a data element.

Lookup Table..................A synonym for *Color Table*.

Low-Level Interface........Refers to *APIs* that refer to data elements such as arrays, attributes, and so on. Compare to a High-Level Interface, where data is referred to as Swaths, Grids, and so on.

LUT ....Look Up Table. Yet another synonym for *Color Table*.

Metadata ....Data that describes data. This term is fairly nebulous, as one person's data is someone else's metadata. In this document, we use the term Metadata to refer to 'Parameter=Value' information that is associated with a Standard Data Product, such as "SPACECRAFT_ID='EOS_AM' ".

Multidimensional Array .. A synonym for *N-dimensional Array*.

MultiFile SDS .... An *API* for *Scientific Datasets* that lets you access to more than one SDS in more than one file at the same time. Commonly known as the "SD" interface. The term can also be used to describe the data object produced by the API, which supersedes the *NDG* and the *SDG*.

N-Dimensional Array ...... Refers to an array of any dimension that contains either scalar data values or a record of various data values at every data location in the array.

NASA .... <u>N</u>ational <u>A</u>eronautics and <u>S</u>pace <u>A</u>dministration

NCSA .... Stands for <u>N</u>ational <u>C</u>enter for <u>S</u>upercomputing <u>A</u>pplications. HDF, NCSA Image, Datascope, NCSA Telnet, NCSA Mosaic, and Collage are all creations of NCSA.

NDG .... Stands for <u>N</u>umeric <u>D</u>ata <u>G</u>roup. Refers to the data objects created with the 'DFSD' *SDS* interface. We recommend using the 'SD' interface, which produces *Multifile SDSs*, for EOS data.

netCDF .... <u>n</u>etwork <u>C</u>ommon <u>D</u>ata <u>F</u>orm. netCDF is another data format library, developed by Unidata, which is freely available and is primarily used by the atmospheric science community.

Numerical Scales .... Synonym for *Dimension Scales*.

ODL .... Stands for <u>O</u>bject <u>D</u>escription <u>L</u>anguage. Developed by the Planetary Data System at the Jet Propulsion Laboratory, it is a text-based language for describing metadata and data dictionaries.

Offset .... In HDF, offsets are used to specify the location of data elements. These offsets are expressed as a number of bytes from the beginning of the file.

Palette .... Yet one more synonym for *Color Table*.

Point Data .... Refers to data collected at random locations, that cannot easily be stored on a regular grid. An example would be a record of temperature measurements taken at various airports across the country.

Point Structure .... Refers to a proposed HDF-EOS datatype for storing *Point Data*.

Projection .... Used here to mean a set of transformation equations that map a sphere onto a flat surface.

Pseudocolor Image .... Here, a synonym for *Raster Image*.

Raster Image .... A rectangular array that is meant to be displayed on a computer screen, with each element in the array corresponding to a particular pixel in the computer display.

Raster Image Group ....In HDF, a structure for gathering the data elements required to represent a raster image. It is like a Vgroup, but is specific to raster images.

Record ....In HDF, a term used to refer to the repeated sequence of fields in a Vdata.

Reference Number ....A unique number assigned to an HDF data element to distinguish it from other element with the same tag number.

RIG.................................Abbreviation for *Raster Image Group*.

RIS24.............................Abbreviation for the 24-bit *Raster Images* provided in HDF. Also refers to the subroutine library for reading and writing 24-bit raster images.

RIS8...............................Abbreviation for 8-bit *Raster Images* provided in HDF. Also refers to the subroutine library for reading and writing 8-bit raster images

RLE ....Stands for Run Length Encoding. A compression method used in the RIS8 interface wherein contiguous runs of pixels with the same color value are expressed as a single color entry with a pixel count.

RTF ....Stands for Rich Text Format. An ASCII-encoded format for storage of formatted text.

Scalar Arrays ....An N-dimensional rectilinear data structure in which all data values are of the same basic type (e.g., 4-byte integer).

Science Data Table..........A proposed EOS datatype organized as a set of named columns and a set of rows where each row contains one entry for each column. Should be very similar to Vdatas.

Scientific Dataset ....A data model and API provided in HDF for the reading and writing of multidimensional homogeneous arrays of data and the attributes of such arrays. The term is also used to refer to the data object produced by the API.

SDF ....Stands for Standard Data Format. Refers to the standard format used for EOS data. Currently the HDF file format has been designated as the SDF for EOS.

SDG...............................See Scientific Data Group.

SDS ....See Scientific Data Set.

Self-describing...............With a self-describing data file, no outside information is needed to fully comprehend the contained data, other than a subroutine library encapsulating the file format design.

Single File Interface........An API that allows access to only one file at a time. HDF has only recently begun to move away from this method of file access.

Standard Data Product.....

Station Data ....Term used to refer to data that comes from a fixed location, with respect to the Earth.

Swath.............................A data model for the reading and writing of data oriented around satellite orbital track. An API for swath data will be designed for HDF-EOS.

Swath Structure...............Our proposed EOS datatype for organizing swath data, especially with making sure geolocation is in standard places.

Table..............................In this document, a synonym for *Science Data Table*.

Tag.................................In HDF-speak, a number assigned by the HDF library that identifies the nature or intended interpretation of the data in a data element.

TIFF...............................Stands for <u>T</u>agged <u>I</u>mage <u>F</u>ile <u>F</u>ormat.

TRMM............................<u>T</u>ropical <u>R</u>ainfall <u>M</u>easuring <u>M</u>ission

UIUC ....Stands for the <u>U</u>niversity of <u>I</u>llinois at <u>U</u>rbana-<u>C</u>hampaign.

V0..................................Usually refers to the operational prototype of the EOSDIS system.

V1..................................Usually refers to the first release of the EOSDIS system.

Vdata ....A record-oriented HDF data model and API provided in HDF. A Vdata corresponds to a data table, where each fixed-length record is made up of a set of individually named and typed fields which make up the columns of the table. *Science Data Tables* make extensive use of Vdata.

Vdata Field ....A named and typed set of data values that make up one column in the table-like Vdata structure.

Vgroup...........................An arbitrary grouping mechanism in HDF used to signify associations between otherwise unrelated data objects.