"A comparative study on Binary Scientific Data Formats" CRS4 Technical Report

E. Gobbetti, A. O. Leone

March 1997

1 Introduction to Binary Scientific Data Formats

In this technical report a set of binary scientific data formats will be examined. They will be presented by examining their main functionality and features.

2 Formats under evaluation

XDR By Sun Microsystem, Inc et al.

NetCDF By UNIDATA, current available version: 2.4

HDF By NCSA, current available version: 4.1r1

Plot 3D By NASA Ames Research Center, current available version: 3.6, distributed by COSMIC (http://www.cosmic.uga.edu/)

TH By CRS4, current available version: 1.2

The following format descriptions are derived from original sources. For easy of consultation they have been modified and summarized. The sources are:

- `ftp://ds.internic.net/rfc/rfc1832.txt`
- `http://www.unidata.ucar.edu/packages/netcdf/`
- `ftp://ftp.ncsa.uiuc.edu/HDF/Documentation/HDF.Specs/`
- E. Gobbetti, A. Leone, M-G. Setzu, G. Zanetti, "Time History: A Data Format for Scientific Data", submitted to "Eurographics Workshop on ViSC 1997"

3 XDR

3.1 Introduction

External Data Representation (XDR) it is not a data format. It is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the SUN WORKSTATION, VAX, IBM-PC, and Cray.

XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. Protocols such as ONC RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style, or least significant bit first.

3.2 XDR basic block size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

3.3 data types

XDR data types specifications are given for: Integer, Unsigned Integer, Enumeration, Boolean, Hyper Integer and Unsigned Hyper Integer, Floating-point, Double-precision Floating-point, Quadruple-precision Floating-point, Fixed-length Opaque Data, Variable-length Opaque Data, String, Fixed-length Array, Variable-length Array, Structure, Discriminated Union, Void, Constant, Typedef and Optional-data.

4 netCdf

The Network Common Data Form (netCDF) is an interface to a library of data access functions for storing and retrieving data in the form of arrays. An array is an n-dimensional (where n is 0, 1, 2, ...) rectangular structure containing items which all have the same data type (e.g. 8-bit character, 32-bit integer). A scalar (simple single value) is a 0-dimensional array.

NetCDF is an abstraction that supports a view of data as a collection of self-describing, network-transparent objects that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data. Generic utilities and application programs can access netCDF files and transform, combine, analyze, or display specified fields of the data. The development of such applications may lead to improved accessibility of data and improved re-usability of software for array-oriented data management, analysis, and display.

The netCDF software implements an abstract data type, which means that all operations to access and manipulate data in a netCDF file must use only the set of functions provided by the interface. The representation of the data is hidden from applications that use the interface, so that how the data are stored could be changed without affecting existing programs. The physical representation of netCDF data is designed to be independent of the computer on which the data were written.

Unidata supports the netCDF interfaces for C, FORTRAN, C++, and perl and for various UNIX operating systems. The software is also ported and tested on a few other operating systems, with assistance from users with access to these systems, before each major release. Unidata's netCDF software is freely available via FTP to encourage its widespread use.

4.1 File Format

To achieve network-transparency (machine-independence), netCDF is implemented in terms of XDR.

The details of the format are available. However, users are discouraged from using the format specification to develop independent low-level software for reading and writing netCDF files, because this could lead to compatibility problems when the format is modified.

4.2 Data Model

A netCDF file contains *dimensions*, *variables*, and *attributes*, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset. The netCDF library allows simultaneous access to multiple netCDF files which are identified by file ID numbers, in addition to ordinary file names.

A netCDF file contains a symbol table for variables containing their name, data type, rank (number of dimensions), dimensions, and starting disk address. Each element is stored at a disk address which is a linear function of the array indices (subscripts) by which it is identified. This obviates the need for these indices to be stored, either as fields within records, or in an index to the records (as in a relational database). This provides a fast and compact storage method, unless there are many missing values.

The notation used to describe netCDF objects is called CDL (network Common Data form Language), which provides a convenient way of describing netCDF files. The netCDF system includes utilities for producing human-oriented CDL text files from binary netCDF files and vice versa.

The CDL notation for a netCDF file can be generated automatically by using `ncdump`, a utility program. Another netCDF utility, `ncgen`, generates a netCDF file (or optionally C or FORTRAN source code containing calls needed to produce a netCDF file) from CDL input.

The CDL notation is simple and largely self-explanatory. It will be explained more fully as we describe the components of a netCDF file. For now, note that CDL statements are terminated by a semicolon. Spaces, tabs, and newlines can be used freely for readability. Comments in CDL follow the characters '//' on any line. A CDL description of a netCDF file takes the form

```
netCDF name {
  dimensions: ...
  variables: ...
  data: ...
}
```

where the name is used only as a default in constructing file names by the ncgen utility. The CDL description consists of three optional parts, introduced by the keywords dimensions, variables, and data. NetCDF dimension declarations appear after the dimensions keyword, netCDF variables and attributes are defined after the variables keyword, and variable data assignments appear after the data keyword.

Here is a simple example of a netCDF file content, using the CDL language:

```
netcdf example_1 {  // example of CDL notation for a netCDF file

dimensions:           // dimension names and sizes are declared first
        lat = 5, lon = 10, level = 4, time = unlimited;
variables:           // variable types, names, shapes, attributes
        float   temp(time,level,lat,lon);
                    temp:long_name     = "temperature";
                    temp:units         = "celsius";
        float   rh(time,lat,lon);
                    rh:long_name = "relative humidity";
                    rh:valid_range = 0.0, 1.0;      // min and max
        int     lat(lat), lon(lon), level(level);
                    lat:units       = "degrees_north";
                    lon:units       = "degrees_east";
                    level:units     = "millibars";
        short   time(time);
                    time:units      = "hours since 1996-1-1";
        // global attributes
                    :source = "Fictional Model Output";

data:                // optional data assignments
        level   = 1000, 850, 700, 500;
        lat     = 20, 30, 40, 50, 60;
        lon     = -160,-140,-118,-96,-84,-52,-45,-35,-25,-15;
        time    = 12;
        rh      =.5,.2,.4,.2,.3,.2,.4,.5,.6,.7,
                 .1,.3,.1,.1,.1,.1,.5,.7,.8,.8,
                 .1,.2,.2,.2,.2,.5,.7,.8,.9,.9,
                 .1,.2,.3,.3,.3,.3,.7,.8,.9,.9,
                  0,.1,.2,.4,.4,.4,.4,.7,.9,.9;
}
```

dimensions A dimension may be used to represent a real physical dimension, for example, time, latitude, longitude, or height. A dimension might also be used to index other quantities, for example station or model-run-number.

A netCDF dimension has both a name and a size. A dimension size is an arbitrary positive integer, except that one dimension in a netCDF file can have the size UNLIMITED.

Such a dimension is called the unlimited dimension or the record dimension. A variable with an unlimited dimension can grow to any length along that dimension. The unlimited dimension index is like a record number in conventional record-oriented files. A netCDF file can have at most one unlimited dimension, but need not have any. If a variable has an unlimited dimension, that dimension must be the most significant (slowest changing) one. Thus any unlimited dimension must be the first dimension in a CDL shape (and first in C declarations, but last in FORTRAN).

CDL dimension declarations may appear on one or more lines following the CDL keyword dimensions. Multiple dimension declarations on the same line may be separated by commas. Each declaration is of the form `name = size`.

There are four dimensions in the above example: lat, lon, level, and time. The first three are assigned fixed sizes; time is assigned the size UNLIMITED, which means it is the unlimited dimension.

The basic unit of named data in a netCDF file is a variable. When a variable is defined, its shape is specified as a list of dimensions. These dimensions must already exist.

The number of dimensions is called the rank (a.k.a. dimensionality). A scalar variable has rank 0, a vector has rank 1 and a matrix has rank 2.

It is possible to use the same dimension more than once in specifying a variable shape. For example, correlation(instrument, instrument) could be a correlation matrix giving correlations between measurements using different instruments. But data whose dimensions correspond to those of physical space/time should have a shape comprising different dimensions, even if some of these have the same size.

variables Variables are used to store the bulk of the data in a netCDF file. A variable represents an array of values of the same type. A scalar value is treated as a 0-dimensional array. A variable has a name, a data type, and a shape described by its list of dimensions specified when the variable is created. A variable may also have associated attributes, which may be added, deleted or changed after the variable is created.

A variable data type is one of a small set of netCDF types that have the names `NC_BYTE`, `NC_CHAR`, `NC_SHORT`, `NC_LONG`, `NC_FLOAT`, and `NC_DOUBLE` in the C interface and the corresponding names `NCBYTE`, `NCCHAR`, `NCSHORT`, `NCLONG`, `NCFLOAT`, and `NCDOUBLE` in the FORTRAN interface. In the CDL notation, these types are given the simpler names byte, char, short, long, float, and double. int may be used as a synonym for long and real may be used as a synonym for float in the CDL notation.

CDL variable declarations appear after the variables keyword in a CDL unit. They have the form

```
type variable_name  ( dim_name_1, dim_name_2, ... ) ;
```

for variables with dimensions, or

```
type variable_name ;
```

for scalar variables.

In the above CDL example there are six variables. As discussed below, four of these are coordinate variables. The remaining variables (sometimes called primary variables), temp and rh, contain what is usually thought of as the data. Each of these variables has the unlimited dimension time as its first dimension, so they are called record variables. A variable that is not a record variable has a fixed size (number of data values) given by the product of its dimension sizes. The size of a record variable is also the product of its dimension sizes, but in this case the product is variable because it involves the size of the unlimited dimension, which can vary. The size of the unlimited dimension is the number of records.

attributes NetCDF attributes are used to store data about the data (ancillary data or metadata), similar in many ways to the information stored in data dictionaries and schema in conventional database systems. Most attributes provide information about a specific variable. These are identified by the name (or ID) of that variable, together with the name of the attribute.

Some attributes provide information about the file as a whole and are called global attributes. These are identified by the attribute name together with a blank variable name (in CDL) or a special null variable ID (in C or Fortran).

An attribute has an associated variable (null for a global attribute), a name, a data type, a length, and a value. The current version treats all attributes as vectors; scalar values are treated as single-element vectors.

Conventional attribute names should be used where applicable. New names should be as meaningful as possible.

The type of an attribute is specified when it is created. The types permitted for attributes are the same as the netCDF data types for variables. Attributes with the same name for different variables should sometimes be of different types. For example, the attribute `valid_max` specifying the maximum valid data value for a variable of type long should be of type long, whereas the attribute `valid_max` for a variable of type double should instead be of type double.

Attributes are more dynamic than variables or dimensions; they can be deleted and have their type, length, and values changed after they are created, whereas the netCDF interface provides no way to delete a variable or to change its type or shape.

The CDL notation for defining an attribute is

```
variable_name:attribute_name = list_of_values ;
```

for a variable attribute, or

```
:attribute_name = list_of_values ;
```

for a global attribute. The type and length of each attribute are not explicitly declared in CDL; they are derived from the values assigned to the attribute. All values of an attribute must be of the same type.

In the netCDF example, units is an attribute for the variable lat that has a 13-character array value `degrees_north`. And `valid_range` is an attribute for the variable rh that has length 2 and values '0.0' and '1.0'.

One global attribute `source` is defined for the example netCDF file. This is a character array intended for documenting the data. Actual netCDF files might have more global attributes to document the origin, history, conventions, and other characteristics of the file as a whole.

Most generic applications that process netCDF files assume standard attribute conventions and it is strongly recommended that these be followed unless there are good reasons for not doing so.

Attributes may be added to a netCDF file long after it is first defined, so you don't have to anticipate all potentially useful attributes. However adding new attributes to an existing file can incur the same expense as copying the file.

5 HDF

5.1 Introduction

The Hierarchical Data Format (HDF) was designed to be an easy, straight-forward, and self-describing means of sharing scientific data among people, projects, and types of computers. An extensible header and carefully crafted internal layers provide a system that can grow as scientific data-handling needs evolve.

A fundamental requirement of scientific data management is the ability to access as much information in as many ways, as quickly and easily as possible. A data storage and retrieval system that facilitates these capabilities must provide the following features:

Support for scientific data and metadata

> Scientific data is characterized by a variety of data types and representations, data sets (including images) that can be extremely large and complex, and the need to attach accompanying attributes, parameters, notebooks, and other metadata. Metadata, supplementary data that describes the basic data, includes information such as the dimensions of an array, the number type of the elements of a record, or a color lookup table (LUT).

Support for a range of hardware platforms

> Data can originate on one machine only to be used later on many different machines. Scientists must be able to access data and metadata on as many hardware platforms as possible

Support for a range of software tools

Scientists need a variety of software tools and utilities for easily searching, analyzing, archiving, and transporting the data and metadata. These tools range from a library of routines for reading and writing data and metadata, to small utilities that simply display an image on a console, to full-blown database retrieval systems that provide multiple views of thousands of sets of data and metadata.

Rapid data transfer

Both the size and the dispersion of scientific data sets require that mechanisms exist to get the data from place to place rapidly.

Extendibility

As new types of information are generated and new kinds of science are done, a means must be provided to support them.

The HDF File Format Structure HDF is a self-describing extensible file format using tagged objects that have standard meanings. The idea is to store both a known format description and the data in the same file. HDF tags describe the format of the data because each tag is assigned a specific meaning: the tag DFTAG_LUT stands for color palette, the tag DFTAG_RI stands for 8-bit raster image, and so on. A program that has been written to understand a certain set of tag types can scan the file for those tags and process the data. This program also can ignore any data that is beyond its scope.

The set of available data objects encompasses both primary data and metadata. Most HDF objects are machine- and medium-independent physical representations of data and metadata.

HDF Tags The HDF design assumes that cannot be known a priori what types of data objects will be needed in the future, nor cannot be known how scientists will want to view that data. As science progresses, people will discover new types of information and new relationships among existing data. New types of data objects new tags will be created to meet these expanding needs. To avoid unnecessary proliferation of tags and to ensure that all tags are available to potential users who need to share data, a portable public domain library is available that interprets all public tags. The library contains user interfaces designed to provide views of the data that are most natural for users. As we learn more about the way scientists need to view their data, we can add user interfaces that reflect data models consistent with those views.

Types of Data and Structures HDF currently supports the most common types of data and metadata that scientists use, including multidimensional gridded data, 2-dimensional raster images, polygonal mesh data, multivariate data sets, finite-element data, non- Cartesian coordinate data, and text.

In the future there will almost certainly be a need to incorporate new types of data, such as voice and video, some of which might actually be stored on other media than the central file itself. Under such circumstances, it may become desirable to employ the concept of a virtual file. A virtual file functions like a regular file but does not fit our normal notion of a monolithic sequence of bits stored entirely on a single disk or tape.

HDF also makes it possible for the user to include annotations, titles, and specific descriptions of the data in the file. Thus, files can be archived with human-readable information about the data and its origins

One collection of HDF tags supports a hierarchical grouping structure called Vset that allows scientists to organize data objects within HDF files to fit their views of how the objects go together, much as a person in an office or laboratory organizes information in folders, drawers, journal boxes, and on their desktops.

Backward and Forward Compatibility An important goal of HDF is to maximize backward and forward compatibility among its interfaces. This is not always achievable, because data formats must sometimes change to enhance performance, to correct errors, or for other reasons. However, whenever possible, HDF files should not become out of date. For example, suppose a site falls far behind in the HDF standard so its users can only work with the portions of the specification that are three years old. Users at this site might produce files with their old HDF software then read them with newer software designed to work with more advanced data files. The newer software should still be able to read the old files.

Conversely, if the site receives files that contain objects that its HDF software does not understand, it should still be able to list the types of data in the file. It should also be able to access all of the older types of data objects that it understands, despite the fact that the older types of data objects are mixed in with new kinds of data. In addition, if the more advanced site uses the text annotation facilities of HDF effectively, the files will arrive with complete human- readable descriptions of how to decipher the new tag types.

Calling Interfaces To present a convenient user interface made up of something more usable than a list of tag types with their associated data requirements, HDF supports multiple calling interfaces.

The low level calling interfaces are used to manipulate tags and raw data, for error handling, and to control the physical storage of data. These interfaces are designed to be used by developers who are providing the higher level interfaces for applications like raster image storage or scientific data archiving.

The application interfaces, at the next level, include several modules specifically designed to simplify the process of storing and accessing specific types of data. For example, the palette interface is designed to handle color palettes and lookup tables while the scientific data interface is designed to handle arrays of scientific data. If you are primarily interested in reading or writing data to HDF files, you will spend most of your time working with the application interfaces.

The HDF utilities and NCSA applications, at the top level, are special purpose programs designed to handle specific tasks or solve specific problems. The utilities provide a command line interface for data management. The applications provide solutions for problems in specific application areas and often include a graphic user interface. Several third party applications are also available at this level.

Machine Independence An important issue in data file design is that of machine independence or transportability. The HDF design defines standard representations for storing all data types that it supports. When data is written to a file, it is typically written in the standard HDF representation. The conversion is handled by the HDF software and need not concern the user. Users may override this convention and install their own conversion routines, or they may write data to a file in the native format of the machine on which it was generated.

5.2 The HDF File Format Components and Organization

This chapter introduces and describes the components and organization of Hierarchical Data Format (HDF) files.

File Header The first component of an HDF file is the file header (FH), which takes up the first four bytes in an HDF file. The file header is a signature that indicates that the file is an HDF file. Specifically, it is a 32-bit magic number with the hexadecimal value 0e031301.

HDF assumes big-endian order in reading and writing files. The order of bytes in the file header might be swapped on some machines when the HDF file header is written, causing these characters to be written in little-endian order. To maintain HDF file portability when developing software for such machines, you must make sure the characters are read and written in the exact order shown.

Data Objects The basic building block of an HDF file is the data object, which contains both data and information about the data. A data object has two parts: a 12-byte data descriptor (DD) and a data element. Figure 1.1 illustrates two data objects.

As the names imply, the data descriptor provides information about the data; the data element is the data itself. In other words, all data in an HDF file has information about itself attached to it. In this sense, HDF files are self-describing files.

Data Descriptor A data descriptor (DD) has four fields: a 16-bit tag, a 16-bit reference number, a 32-bit data offset, and a 32-bit data length. These are briefly described in the next table.

Part	Description
Tag/ref	Unique identifier for each data element (data identifier)
Tag	Type of data in a data element
Reference number	Number distinguishing data element from others with the same tag
Offset	Byte offset of data element from beginning of file
Length	Length of data element

A tag and its associated reference number (abbreviated as tag/ref) uniquely identify a data element in an HDF file. The tag/ref combination is also known as a data identifier.

Tag A tag is the part of a data descriptor that tells what kind of data is contained in the corresponding data element. A tag is actually a 16-bit unsigned integer between 1 and 65535, but every tag is also given a name that programs can refer to instead of the number. If a DD has no corresponding data element, its tag is DFTAG_NULL, indicating that no data is present. A tag may never be zero.

Tags are assigned by NCSA as part of the specification of HDF.

Reference Number Tags are not necessarily unique in an HDF file; there may be more than one data element of a given type. Therefore, each tag is associated with a unique reference number in the data descriptor.

Reference numbers are not necessarily assigned consecutively, so you cannot assume that the actual value of a reference number has any meaning beyond providing a way of distinguishing among elements with the same tag. Furthermore, reference numbers are only unique for data elements with the same tag; two 8-bit raster images will never have the same reference number but an 8-bit raster image and a 24-bit raster image might.

Reference numbers are 16-bit unsigned integers.

Data Offset and Length The data offset states the byte position of the corresponding data element from the beginning of the file. The length states the number of bytes occupied by the data element.

Offset and length are both 32-bit unsigned integers.

DD Blocks Data descriptors are stored physically in a linked list of blocks called data descriptor blocks or DD blocks. All of the DDs in a DD block are assumed to contain significant data unless they have the tag DFTAG_NULL (no data).

In addition to its DDs, each data descriptor block has a data descriptor header (DDH). The DDH has two fields: a block size field and a next block field. The block size field is a 16-bit unsigned integer that indicates the number of DDs in the DD block. The next block field is a

32-bit unsigned integer giving the offset of the next DD block, if there is one. The DDH of the last DD block in the list contains a 0 in its next block field.

Since the default number of DDs in a DD block is defined when the HDF library is compiled, changing the default requires recompilation.

Data Element A data element is the raw data portion of a data object. Its data type can be determined by examining its tag, but other interpretive information may be required before it can be processed properly.

Each data element is stored as a set of contiguous bytes starting at the offset and with the length specified in the corresponding DD.

Exceptions A data object identified by the tag DFTAG_MT does not adhere to the standards described above; it consists of the tag immediately followed by four number types. Since there can be only one DFTAG_MT tag in an HDF file, there is no need for a reference number. Since all the data can be stored in the DD with the tag, there is no need for a data element and the offset and length are unnecessary.

Several other tags, such as DFTAG_NULL and DFTAG_JPEG, serve as binary flags and convey all the required information by the mere fact of their presence in an HDF file. These tags therefore point to no data element and have offset and length values of 0. Consider these examples: DFTAG_NULL indicates a data object containing no data; DFTAG_JPEG indicates that an associated data object, indicated by another tag, contains a JPEG data image.

5.3 Overview

HDF is an amalgam of code and functionality from many sources. For example, the netCDF code came from the Unidata Program Center, and data compression and conversion software has been acquired from a variety of third parties. NCSA staff wrote the code for the basic HDF functionality and performed all of the integration work.

Here specifications for the NCSA-developed code and functionality are described.

HDF Software Layers There are three basic levels of HDF software:

- HDF low level interface
- The HDF application interfaces
- HDF applications and utilities

The lowest layer, the low level interface, includes general purpose routines that form the basis of all higher-level HDF development. The low level routines directly execute functions such as file I/O, error handling, memory management, and physical storage.

The application interfaces support higher level views of data and provide the interfaces for building user-level applications. Routines to handle raster images, palettes, annotations, scientific data sets, Vdatas and netCDF appear at this level.

The applications and utilities are implemented at the highest level. NCSA utilities, NCSA applications, and third party applications are all implemented at this level.

The utilities perform general functions, such as listing the contents of an HDF file, and more specialized functions, such as converting data from one HDF data type to another (e.g., raster images to scientific data sets). In general, the utilities have simple command line interfaces and perform data management tasks.

The applications usually perform data analysis tasks and have polished interactive user interfaces. They include the NCSA Visualization Tool Suite, commercial software packages that use HDF, and other packages created at NCSA and by various third party projects.

HDF-based software comes in four basic forms:

- The HDF interface library
- HDF command line utilities
- HDF-based software tools
- User programs that store and retrieve data in HDF files

The HDF interface library includes general purpose routines that form the basis of all higher-level HDF development and application interfaces that support higher level views of data.

The HDF command line utilities are distributed with the HDF library. They range from general purpose utilities, such as listing the contents of an HDF file, to special purpose utilities, such as converting data between HDF data types (e.g., raster images to scientific data sets). In general, the utilities have simple command line interfaces and perform data management tasks.

In contrast, HDF-based software tools usually perform data analysis tasks and have polished interactive user interfaces. They include the NCSA Visualization Tool Suite and commercial software packages that use HDF.

User programs access HDF files via calls to the HDF library. User programs are attached to the HDF library when they are compiled and linked.

Since the NCSA user community writes programs primarily in C and FORTRAN, all of the HDF application interfaces developed at NCSA are callable from both C and FORTRAN programs. Since the general purpose interface is primarily for program development, not for applications, it provides C- callable routines only.

5.4 Sets and Groups

This section discusses the roles of the following sets and groups in organizing data stored in an HDF file:

- Raster image sets (RIS)
- Raster image groups (RIG)
- Scientific data sets (SDS)
- Scientific data groups (SDG)
- Numeric data groups (NDG)
- SDG-like NDGs
- Vsets
- Vgroups

Data Sets HDF files frequently contain several closely related data objects. Taken together, these objects form a data set which serves a particular user requirement. For example, five or six data objects might be used to describe a raster image; eight or more data objects might be used to describe the results of a scientific experiment.

The HDF mechanism for specifying and controlling data sets is the group. The data element of a group consists of a single record listing the tag/refs for all the objects contained in the data set. For example, the raster image groups described in the following sections each contain three tag/refs that point to three data objects that, taken as a set, fully describe an 8-bit raster image.

The current HDF implementation supports three kinds of sets:

Raster image set

A set containing a raster image and descriptive information such as the image dimensions and an optional color lookup table

Scientific data set

A set containing a multidimensional array and information describing the data in the array

Vset

A general grouping structure containing any kinds of HDF objects that a user wishes to include

Each HDF set is defined with a minimum collection of data objects that will make sense when the set is used. For example, every raster image set must contain at least the following data objects:

Raster image group

The list of the members of the set

Image dimension record

The width, height, and pixel size of the raster image

Raster image data

The pixel values that make up the image

In addition to the required objects, a set may include optional data objects. An 8-bit raster image set, for instance, often contains a palette, or color lookup table, which defines the red, green, and blue values associated with each pixel in the raster image.

Calling Interfaces for Sets NCSA provides calling interfaces for all the HDF sets that it supports. These interfaces provide routines for reading and writing the data associated with each set. The libraries currently supported by NCSA are callable from either C or FORTRAN programs.

In addition to the libraries, a growing number of command-line utilities are available to manipulate sets. For example, a utility called r8tohdf converts one or more raw raster images to HDF 8-bit raster image set format.

Groups As discussed above, HDF data objects are frequently associated as sets. But without some explicit identifying mechanism, there is often no way to tie them together. To address this problem, HDF provides a grouping mechanism called a group. A group is a data object that explicitly identifies all of the data objects in a set.

Since a group is just another type of data object, its structure is like that of any other data object; it includes a DD and a data element. But instead of containing the pixel values for a raster image or the dimensions of an array, a group data element contains a list of tag/refs for the data objects that make up the corresponding set.

A group tag can be defined for any set. For instance, the raster image group tag (RIG, DFTAG_RIG) is used to identify members of raster image sets; the RIG data element lists the tag/refs for a particular raster image set.

General Features of Groups Here are some important general features of groups:

- The contents of a group must be consistent with one another. Since the palette (DFTAG_IP8) is designed for use with 8-bit images, the image must be an 8-bit image.
- An application program can easily process all of the images in the file by accessing the groups in the file. The non-RIG information in the example can be used or ignored, depending on the needs and capabilities of the application program.
- There is usually more than one way to group sets. For example, an extra copy of the image palette (DFTAG_IP8) could have been stored in the file so that each grouping would have its own image palette. That is not necessary in this instance because the same palette is to be used with both images. On the other hand, there are two image dimension records in this example, even though one would suffice.
- Group status does not alter the fundamental role of an HDF object; it is still accessible as an individual data object despite the fact that it also belongs to a larger set.
- A group provides an index of the members of a set. There is nothing to prevent the imposition of other groupings (indexes) that provide a different view of the same collection of data objects. In fact, HDF is designed to encourage the addition of alternate views.

Raster Image Sets The raster image set (RIS) provides a framework for storing images and any number of optional image descriptors. An RIS always contains a description of the image data layout and the image data. It may also contain color look-up tables, aspect ratio information, color correction information, associated matte or other overlay information, and any other data related to the display of the image.

Raster Image Groups Tying everything about an image together is the raster image group. An RIG contains a list of tag/refs that point in turn to the data objects that make up and describe the image.

The number of entries in an RIG is variable and most of the descriptive information is optional. Complex applications may include references to image-modifying data, such as the color table and aspect ratio, along with the reference to the image data itself. Simple applications may use simple application- level calls and ignore specialized video production or film color correction parameters.

NCSA currently supports two RIG calling interfaces: RIS8 and RIS24.

Since new forms of data compression can be added to HDF raster images, incompatibilities can arise between old libraries and files created by newer libraries. For example, HDF Version 3.3 includes JPEG compression for images. A JPEG-compressed raster image in a file created by an HDF Version 3.3 library cannot be read by an HDF Version 3.2 library.

Scientific Data Sets The scientific data set (SDS) provides a framework for storing multidimensional arrays of data with descriptive information that enhances the data. Current specifications support the following types of numbers in SDS arrays.

- 8-bit, 16-bit, and 32-bit signed and unsigned integers
- 32-bit and 64-bit floating point numbers

Data in an SDS can be stored either as two's complement big endian integers, as IEEE Standard floating point numbers, or in native mode, the format used by the machine from which they were written.

One of NCSA concerns in HDF development is always to maximize backward and forward compatibility; as much as possible, any application written to use HDF should be able to read data files written with an older or a newer version of the libraries. To maximize this compatibility, NCSA had to consider the following factors in upgrading the SDS capabilities:

- Support for future variations (e.g., new number types, data compression, and new physical arrangements for SDS storage)
- Older versions of the library should be able to read new data files if the data itself can be interpreted by the older version. To do so, the older version must be able to determine whether the data in a given data object will be comprehensible to it. For example, if a newly created file contains 32-bit IEEE floating point or Cray floating point data objects, older versions of the library should be able determine that fact then read and interpret the data.
- New libraries must be able to read and interpret files created by older versions.

Scientific and Numeric Data Group The SDS capability was substantially enhanced from HDF Version 3.2. Previous versions employed a structure known as a scientific data group (SDG); Version 3.2 and subsequent versions use the numeric data group (NDG). To accommodate the enhanced structure and to remain compatible with previous releases, the current HDF library supports the following scientific and numerical data groups:

SDGs

Created by old libraries and containing 32-bit IEEE and Cray floating-point data.

NDGs

Created by the newer libraries (Version 3.2 and later) and containing any acceptable floating- point or non-floating-point data. This data group will not be recognized by old libraries.

SDG-like NDGs

Created by the new library and containing IEEE 32-bit floating-point data only. The old libraries will recognize and interpret these numerical data groups correctly.

The NDG structure supports 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating-point numbers. It also supports native mode, data sets written to HDF files in the local machine format.

SDGs must contain at least these data objects: `DFTAG_SDG` (scientific data group), `DFTAG_SDD` (dimension record for array-stored data, included the rank, the size of each dimension, and the tag/refs representing the number type of the array data and of each dimension), and `DFTAG_SD` (scientific data).

In addition to the required data objects listed above, SDGs may contain any of: `DFTAG_SDS` (scales of the different dimensions), `DFTAG_SDL` (labels for all dimensions and for the data), `DFTAG_SDU` (units for all dimensions and for the data), `DFTAG_SDF` (format specifications), `DFTAG_SDM` (maximum and minimum values of the data) and `DFTAG_SDC` (coordinate system to be used when interpreting or displaying the data).

NDGs must contain at least these data objects: `DFTAG_NDG` (numerical data group), `DFTAG_SDD` (dimension record for array-stored data), `DFTAG_SD` (scientific data) and `DFTAG_NT` (number type).

In addition to these required data objects, an NDG may contain any of the optional SDG tag described above.

Vsets, Vdatas, and Vgroups Vsets, Vdatas, and Vgroups enable users to create their own grouping structures. Unlike RIGs, SDGs, and NDGs, HDF imposes no required structure; they are implemented almost entirely at the user level and are not specified in detail in HDF. The only specifications define DFTAG_VG, DFTAG_VH, and DFTAG_VS tags and the formats of their respective data elements. A detailed discussion similar to that for the other grouping structures is, therefore, inappropriate here.

An HDF Vset can contain any logical grouping of HDF data objects within an HDF file. Vsets resemble the UNIX file system in that they impose a basically hierarchical structure but also allow cross-linked data objects. Unlike SDSs and RISs, Vsets have no prespecified content or structure; users can use them to create structural relationships among HDF objects according to their needs. Figure 4.6 illustrates a Vset.

A Vset is identified by a Vgroup, an HDF object that contains information about the members of the Vset. The tag DFTAG_VG identifies the Vgroup which contains the tag/refs of its members, an optional user-specified name, an optional user- specified class, and fields that enable the Vgroup to be extended to contain more information.

The only required Vgroup tag is the tag that defines the Vgroup itself.

5.5 HDF Portability Issues

The NCSA implementation of HDF is accessible to both C and FORTRAN programs and is implemented on many different machines and several operating systems. There are important differences between C and FORTRAN, and among implementations of each language, especially FORTRAN. There are also important differences among the machines and operating systems that HDF supports.

The list of machines and operating systems on which HDF is implemented is steadily growing. Every time a platform is added, additional code must be written to address concerns of memory management, operating system and file system differences, number representations, and differences in FORTRAN and C implementations on that system.

Supported Platforms

As of this writing, NCSA supports these platforms:

- Convex - Concentrix
- Cray X-MP, Y-MP, Cray 2 - UNICOS
- DEC Alpha - Ultrix
- DECStation - Ultrix
- HP 9000 - HPUX
- IBM PC - MS DOS, Windows 3.1
- IBM RS/6000 - AIX
- IBM RT - UNIX
- Macintosh - MPW Shell
- NeXT - NeXTStep
- Silicon Graphics - UNIX
- Sun Sparc - UNIX
- Vax - VMS

Unfortunately, not all compilers are the same. FORTRAN compilers often differ in the ways they pass parameters, in the identifier naming conventions they employ, and in the number types that they support. Similarly, though generally not as drastically, C compilers differ in the number types that they support and in their adherence to the ANSI C standard.

To minimize the difficulties caused by these differences, the HDF source code is written primarily in FORTRAN 77, ANSI C and the original C defined by Kernighan and Ritchie1. Almost all platforms have C and FORTRAN compilers that adhere to at least one of these standards.

6 Plot3d

There are two types of *Plot3D* files: the **XYZ** grid files that specify the irregular coordinate information, and the **Q** solution files that contain a vector of values for each point in the grid.

XYZ and Q files pairs can contain a single set of grid/data mapping, or multiple grid/data mappings. The XYZ file can also contain an IBLANK value for each point. The data within the files can be either in binary, or FORTRAN formatted or unformatted format. XYZ grid file and Q solution file formats must match in all respect.

Unfortunately, there is no documentation about this format available in public domain. We point the reader of this document to the *PLOT3D Reference Manual* from Cosmic NASA's Software Technology Transfer Center, for further details about this format.

7 TH

Time History (TH) is a new data format and library for scientific applications. The format is self-describing, easy to use and efficient in storing data. These characteristics make it particularly well suited for managing CFD-like data produced by numerical simulations. The TH data format specification is derived from the NCSA (National Center for Super Computing Application - University of Illinois at Urbana-Champaign, IL, USA) HDF data format. TH's interface library, built on top of the HDF library, has been designed so as to reduce the number of user calls and parameter settings and to facilitate the evolution of the format.

The TH project started at CRS4 (Centre for Advanced Studies, Research and Development in Sardinia, Italy) in 1992, with the need of finding a consistent format for data description which had to be particularly well suited to represent data produced by CFD-like numerical simulations.

In order for such a data format to be suitable to the task, essential characteristics are the following:

- *Self describing format*: together with the data, enough information should be stored so as to allow programs reading the data to automatically understand the data structure. This should be done with minimal overhead both in user time and storage requirements.
- *Ease of use*: there should be minimal user involvement in parameter setting and functions calls; in particular, the user should not be forced to set parameters that will not be used.
- *Efficiency in storing data*: data should be stored in binary mode, optionally compressed, and should be readable from multiple architectures. Replication of common data should be avoided.

The TH data format and library have been designed to answer these needs.

TH is derived from NCSA (University of Illinois, Urbana Champaign) HDF data format, and its interface library is built on top of the HDF library, more precisely on top of the VSet interface. Although the HDF format is more powerful and more general than TH in saving data (in fact, TH uses only a small part of the HDF specification), TH has a simpler interface, which was very important to have it accepted as a basic vehicle for data interchange among scientists.

Many available data formats have been evaluated to directly accomplish the task, including HDF, netCDF and Plot3D. Some of these formats, such as HDS and netCDF, are not general enough to handle CFD-like data in a simple way. Although the other formats we have evaluated are well defined and efficient in storing particular classes of data, they generally lack a simple interface. We consider ease of use a necessary condition for the format to be well accepted and effectively used by scientists and programmers.

7.1 Introduction

TH is mostly geared towards storing the data produced by numerical simulations, but, in principle, it is not limited to that. The TH library consists of a set of routines that allows C and FORTRAN users to write their multi-domain, time dependent, structured and unstructured data in a way that is as independent from format details as possible.

Data is stored in a TH file in a hierarchical way: at the first level of the structure, all information about a particular sampling time is stored; at the second level, all information about a particular sub-region of the whole computational domain is stored. These data deal with the spatial sampling information (e.g. the computational grid) and the numerical data (scalar or vector field) defined on those point locations.

The basic TH interface library is complemented with a set of tools for easily examining or manipulating the contents of a TH file. For example, a command line program can be used to get high-level information contained in a TH file, such as the number of computational time steps, the name of the scalars and vectors, or even the minimum and maximum value of a field stored in the internal structure.

Furthermore, some visualization programs have been extended in order to recognize the TH format and to directly access data stored in TH files. In this way, data stored in TH files can be directly accessed and rendered from such visualization tools, removing the annoying job of converting data from one format to another.

TH source code has been put in the public domain. Documentation and source code for the current version of TH (1.2) is available through the World Wide Web at the URL: `http://www.crs4.it/~hdf/TH/`.

7.2 Format specifications

The TH specification is based on the NCSA HDF VSet one, which allows users to store data in a HDF file by building a high-level hierarchical structure. The basic building blocks used by the HDF VSet format are: the **Vdata** elements, where numerical values are stored together with their description (such as format and number of values stored in it), and the **Vgroup** elements, where Vdata and Vgroup itself can be grouped to build the internal file structure.

Using the appropriate HDF routines that compose the VSet API, it is possible to build complex data structures or to traverse an existing data tree to retrieve the information stored in a file. In HDF, there are no constraints nor guidelines on how to organize information in a VSet, thus leaving the user the choice of building the appropriate structure where to store his own data.

The TH specification removes the freedom of choice in the VSet structure and fixes in a rigid way relations and meanings of Vgroups and Vdatas stored in a TH file. The internal data structure of a TH file has been defined so as to be well suited for complex CFD-like data: in fact it is able to contain time-dependent, multi-block, scalar and vector field data defined over regular or irregular grids.

Depending on the position a VSet element occupies in the internal structure stored in a TH file, four different **TH object** types can be identified. If we also include TH files in this classification, there are thus, at the whole, five distinct TH object types:

File. It represents the most external object and it is essentially a HDF file containing Vsets with a predefined hierarchy on vgroup and vdata.

Frame. A *file* can contain one or more frames, that can be seen as big containers with a "step" or a "time" associated with them. They represent substantially a snapshot at a fixed instant of the system we are working with.

Blocks. A *frame* can contain one or more blocks each of them containing spatial informations about the particular region we are considering. The information is described through a structure (grid) over which fields (scalars and vectors) are defined.

Grids. A *block* can contain only one grid, i.e. a collection of ordered or non-ordered spatial coordinates which define the points over which the fields are defined. TH provides four different kinds of grids: regular, rectangular, curvilinear or unstructured. While regular, rectangular and curvilinear grids have an ordered structure where connections between points are defined implicitly, unstructured grids are non ordered collections of points with or without connections between them that form **polygons**.

Fields. Over the *grid* we can define one or more fields, i.e. scalars and vectors. Multiple data types are supported (integers, real*4, real*8, etc). Data can be produced by numerical simulations or collected experimentally. It should be noted that a field defined over the spatial structure is needed for a lot of visualization programs.

7.3 TH api

A set of routines, collected in a library, makes up the TH Application Programming Interface (API). These routines allow the user to easily access a TH file for reading or writing data while leaving the management of the internal structure to the underlying software.

The library has been built having two main goals in mind:

1. **stability**: The library can be extended in future by adding new routines or generalizing some existing one, but old sequences of TH calls must be kept working also in successive versions of TH. This provides an ever-working and stable interface to TH data. Even if the underlying software changes (as it was in the past for the HDF VSet calls), TH users still continue to have the same interface to TH data, leaving the job of managing data format evolution and/or changes to the library;

2. **small interfaces**: to have a smart interface in order to reduce the number of user calls and the user parameter settings necessary when accessing TH data. This has been achieved by writing each routine so as to understand the context in which it is invoked in order to perform the appropriate task.

The backward compatibility goal of TH has been implemented by recording in any TH file a tag coding the version of library used to create it. It is then a library task to manage appropriately the content of the file with respect to its version and the version of the file.

The context-sensitivity of TH routines has been implemented by introducing the concept of *current objects*. Any time a TH file is accessed, a list of four element is created. Each entry carries a reference to a frame, a block, a grid and a field respectively, to which successive TH calls will apply. When a TH call is invoked, references to current TH objects are taken into account, the appropriate task is performed by the routine and the current object list is updated.

For example, if the routine to create a new block is called, then the block is created, inserted into the current frame and promoted to be the current block for that file.

In that way, it is not necessary to specify to which file, frame, block, grid and variable a particular call has to be applied, because it can be resumed by the current object list.

TH library routines are callable from C as well as FORTRAN programs and can be grouped in the following categories:

Functions to create TH objects: to create *files* by giving their name; *frames* by specifying the absolute time that frame is referred to; *blocks* by specifying a number called user index for further reference; *grids* by specifying useful parameters (different for regular, rectangular, curvilinear or unstructured kind) to identify spatial positions of numeric data; *fields* by specifying the name and type (currently scalar or vector).

Functions to annotate TH objects: to associate annotations to TH objects by specifying a string of characters.

Functions to set objects: once a TH object has been defined you can set it as current by calling one of these functions specifying the corresponding identifier number. You can also specify the time in case of frame, the user index in case of block, the name in case of fields.

Functions to get objects: these routines are useful to recover information about TH objects in a file. You can get information about how many objects are present, the name of an object, the time in case of frame, the user index in case of blocks, the grid type and all the associated information in case of grids, the data values of current variable set (scalar or vector component), defined over the grid points.

Functions to list objects: these functions are used to get identifier numbers of all TH objects inside the current object set. A routine allows to get TH identifiers of all frames in the current file with their associated time.

Functions to insert objects in other objects: these functions are useful if you want to insert, for instance, a previous defined grid inside the current block or if you wish to insert a new scalar or vector defined over the current grid.

I/O and miscellaneous functions: we grouped here functions to open and to close a TH file; to deal with the global rank associated to a TH file; to write a variable in a TH file, that is a scalar field or a vector component, by specifying the type, the number of points over which it is defined and the values; to deal with the minimum and maximum values of the current variable.

Each routine has prefix `TH` or `FTH`, depending ow whether it is called from a C or a FORTRAN programs, then followed by `def`, `ann`, `set`, `get`, `list`, `ins` depending on which of the previous categories it belongs.

7.4 Platform support

The current version of TH (1.2) has been successfully tested on the following platforms:

- SGI-INDIGO2 IRIX/5.3
- IBM-RS/6000-550 AIX/3.2.5
- SUN-4M/75 SunOs/4.1.3
- HP9000/712-80 HPUX/9.05

- DEC-ALPHA OSF1/3.0

TH source code has been put in public domain from the end of last year. Documentation and source code is available through the World Wide Web at the URL: `http://www.crs4.it/~hdf/TH/`. Compressed tar files containing TH1.2 pre-compiled binaries for different platforms can be found at the same address.

8 Conclusions

Besides XDR (not really a data format) and Plot3D (because of lack of documentation), only three data format have been evaluated. Furthermore, netCDF is fully supported by HDF format, reducing the evaluation task substantially to only two binary format: HDF and TH. Both can completely and efficiently accomplish CFD-like complex projects. The first one has a robust data model, a long list of features and it is widely diffused in the scientific community. The second, based on the first one, although less powerful than HDF, has the advantages to be easy to use and fully supported by one of the project partners.

9 References

There are official web pages for some of the formats examined in this document. Their URLs are:

- `http://www.unidata.ucar.edu/packages/netcdf/`
- `http://hdf.ncsa.uiuc.edu/`
- `ftp://ds.internic.net/rfc/rfc1832.txt`
- `http://www.crs4.it/ hdf/TH/`

Other reference can be found in:

- Michael Folk, Laura Kalman, William Whitehouse, HDF Users' Guide, National Centre for Supercomputing Applications, University of Illinois at Urbana-Champaign, IL, USA, 1996
- Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, NetCDF User's Guide - An Interface for Data Access, Unidata Program Center, February 1996
- Pamela P. Walatka, Pieter G. Buning, Larry Pierce, and Patricia A. Elson, Plot3d User's Manual, NASA Ames Research Center , March 1990