Parsing Hierarchical Data Format (HDF) Files

Karl Nyberg
Grebyn Corporation
P. O. Box 47
Sterling, VA 20167-0047
703-406-4161
karl@nyberg.net

ABSTRACT
This paper presents a description of the creation of a library to parse Hierarchical Data Format (HDF) Files in Ada. It describes a "work in progress" with discussion of current performance, limitations and future plans.

Categories and Subject Descriptors
D.2.2 [Design Tools and Techniques]: Software Libraries; E.2 [Data Storage Representations]: Object Representation; I.2.10 [Vision and Scene Understanding]: Representations, data structures, and transforms.

General Terms
Algorithms, Performance, Experimentation, Data Formats.

Keywords
Ada, HDF.

1. INTRODUCTION
Large quantities of scientific data are published each year by NASA. These data are often accompanied by metadata files that describe the contents of individual files of the data. One example of this data is the ASTER (Advanced Spaceborne Thermal Emission and Reflection Radiometer) [1]. Each file of data (consisting of satellite imagery) in HDF (Hierarchical Data Format) [2] is accompanied by a metadata file in XML (Extensible Markup Language) [3], encoded according to a published DTD (Document Type Description) that indicates the components and types of data in the metadata file.

Each ASTER data file consists of an image of the satellite as it passes over the earth [4]. Information on the location of the data collected as the satellite passes is contained in the metadata file. Over time, multiple images of the same location on earth are obtained. For many purposes of analysis (erosion, building patterns, deforestation, glacier movement, etc.), these images of the same location are compared over time.

In order to determine which images contain data from common locations, the metadata describing these images can be parsed and a list of files according to location can be ascertained. A separate article describes the ASTER project and the metadata parsing effort [8]. This article describes the creation of a library to parse the individual HDF files.

2. HIERARCHICAL DATA FORMAT

2.1 History
The Hierarchical Data Format (HDF) was originally developed at the Graphics Foundations Task Force (GFTF) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. Its purpose was to:

> create an architecture-independent software library and file format to address the need to move scientific data among the many different computing platforms in use at NCSA at that time. Additional goals for the format and library included the ability to store and access large objects efficiently, the ability to store many objects of different types together in one container, the ability to grow the format to accommodate new types of objects and object metadata, and the ability to access the stored data with both C and Fortran programs [6]

HDF has been maintained and supported since 2005 by the HDF Group, which was separated from the University of Illinois as a non-profit 501 (3) (c) company.

2.2 Description
The HDF format allows the definition of data collections in a hierarchical manner with an embedded directory structure and data descriptions. The following sections describe those data structures within HDF that were encountered in this project and for which support in reading from the corresponding files was developed. Additional documentation is available at the HDF Group's web site – http://www.hdfgroup.org.

An HDF file essentially (after a four byte file identification header) consists of two types of data structures – data descriptors and data elements. Objects of these two types are stored within an HDF file with data descriptors being packaged into blocks (of up to 200 in length for the data sets under consideration) and data elements being stored individually throughout the file. In general, due to the manner in which data is written into an HDF file, there will be blocks of data descriptors followed by data objects, followed by data descriptors, etc.

2.2.1 Data Descriptors
Data descriptor objects contain descriptive information on the data they describe (a tag to indicate the type and a reference number to disambiguate objects of the same type), an offset into

the file to indicate the location of the object and an indicator of the length of the data. A block of data descriptors indicates the quantity of valid data descriptors contained within the block and the offset within the file to the next data descriptor block as needed.

```
type Header_Block is record
      Num_Dds : Interfaces.Integer_16;
      Block_Size : Interfaces.Integer_16;
      Next_Block : Interfaces.Unsigned_32;
end record;

type Data_Descriptor is record
   Tag : Interfaces.Integer_16;
   Reference_Number : Interfaces.Integer_16;
   Offset : Interfaces.Integer_32;
   Length : Interfaces.Integer_32;
end record;

type Data_Descriptor_Array is array
   (Interfaces.Integer_16 range <>) of
   Data_Descriptor;
```

The reference numbers are merely unique identifiers used to group the objects. The tags are defined in the HDF specification and are the real workhorse component of the HDF format. Additional description of tags follows below. Reading in the data descriptors was as straightforward as walking the array of descriptors in memory and following any "pointers" to successive blocks.

2.2.2 Data Elements

Data element objects can either be collections of objects (called a data set), sometimes also put together in a group to identify all the objects in a set. View Groups (Vgroups) allow the organization of objects to associate these objects in the current application.

```
type Vgroup_Record is record
   Nelt : Interfaces.Integer_16;
   Tags : Integer_16_Array_Ptr;
   Refs : Integer_16_Array_Ptr;
   Namelen : Interfaces.Integer_16;
   Name : String_Ptr;
   Classlen : Interfaces.Integer_16;
   Class : String_Ptr;
   Extag : Interfaces.Integer_16;
   Exref : Interfaces.Integer_16;
   Version : Interfaces.Integer_16;
end record;
```

In addition to Vgroups, also of interest were objects that specified Scientific Data Groups (SDG) contents, as this was the type of data of interest in this project. In a SDG, there were required objects with three tags (HDF 5.5.3 SDG Structures):

DFTAG_SDG	Scientific Data Group
DFTAG_SDD	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag / refs representing the number type of the array data and of each dimension.
DFTAG_SD	Scientific data.

```
type Scientific_Data_Dimension_Record is
record
   Rank : Interfaces.Integer_16;
   Dimensions : Integer_32_Array_Ptr;
   Data_Nt_Ref : Interfaces.Integer_16;
   Scale_Nt_Refs : Integer_16_Array_Ptr;
end record;
```

Once the type and dimensions of the scientific data were determined, it was simple enough to instantiate a record with the particular type_code (an enumeration type covering the basic data types – unsigned and signed integers of various sizes, floating point values, strings, etc.) upon encountering an object's specification.

```
type Scientific_Data_Record
  (Of_Type : Type_Info_Codes) is record
   case Of_Type is
      when DFNT_NONE =>
         null;
      when DFNT_UINT8 =>
         Data_Unsigned_8 :
            Unsigned_8_Array_Ptr;
      when DFNT_UINT16 =>
         Data_Unsigned_16 :
            Unsigned_16_Array_Ptr;
      when DFNT_UINT32 =>
         Data_Unsigned_32 :
            Unsigned_32_Array_Ptr;
      when DFNT_VERSION =>
         Data_Version : String_Ptr;
      ...
```

3. LIBRARY STRUCTURE

The HDF library is neatly broken up into four separate packages: data types, reading routines, parsing routines, and display routines.

3.1 Data Types

The HDF data types include the `Type_Info_Codes` (identifying underlying physical data types on the computer, *e.g.,*various integer and float precisions, character and string), corresponding array types, defining HDF tag types, instantiation of low level conversion procedures and HDF data type declarations. Of particular interest are the tag types and some of the data type declearations.

3.1.1 Tag Types

Tag types, much like TIFF tags [9], indicate the type of data that the tag describes. The tags fall into the following categories (HDF Specification 9.3):

- Utility tags
- Annotation tags
- Compression tags
- Raster Image tags
- Composite image tags
- Vector image tags
- Scientific data set tags

- Vset tags
- Obsolete tags
- Extended tags

Currently, the following tags are recognized and supported:

- DFTAG_VERSION – Version Description
- DFTAG_NT – Numeric Type
- DFTAG_SDD – Scientific Data Dimension
- DFTAG_SD – Scientific Data
- DFTAG_NDG – Numeric Data Group
- DFTAG_VH – Vdata Description
- DFTAG_VS – Vdata
- DFTAG_VG – Vdata Group

3.1.2 Data Type Declarations

The data type declarations are simply straightforward to implement from the specification. Some additional examples:

```
type Scientific_Data_Dimension_Record is
   record
   Rank : Interfaces.Integer_16;
   Dimensions : Integer_32_Array_Ptr;
   Data_Nt_Ref : Interfaces.Integer_16;
   Scale_Nt_Refs : Integer_16_Array_Ptr;
end record;

type Numeric_Data_Group_Record is record
   Tags : Hdf_Types.Integer_16_Array_Ptr;
   Refs : Hdf_Types.Integer_16_Array_Ptr;
end record;
```

3.2 Reading Routines

The reading routines include both lower level data objects (such as the various integer and float types) and the various HDF data elements. If additional underlying machine architectures were to be supported, this package might reasonably be split to provide additional packaging mechanism opportunities for different such architectures rather than the simple alternative reading scheme chosen (described below) by separating the lower level reading routines out. What remains are mid-level routines between the parsing package and the low-level reading routines.

3.3 Parsing Routines

The parsing routines parse data descriptor items. The majority of routines in this package currently contain stubs, since only those data descriptors corresponding to tags encountered in the current application were implemented. For example, to read the contents of a Version object:

```
procedure Read_Version
   (File_Identifier : Hdf_Io.File_Type;
    Data_Descriptor :
       Hdf_Types.Data_Descriptor;
    Version : in out
   Hdf_Types.Version_Record_Ptr) is
begin
   Hdf_Io.Set_Index
      (File_Identifier,
       Hdf_Io.Count (Data_Descriptor.Offset)
                     + 1);
   Version := new Hdf_Types.Version_Record;
   Read_Unsigned_32
      (File_Identifier, Version.Major_V);
   Read_Unsigned_32
      (File_Identifier, Version.Minor_V);
   Read_Unsigned_32
      (File_Identifier, Version.Release);
   Read_String
      (File_Identifier,
       Interfaces.Integer_16
          (Data_Descriptor.Length),
       Version.The_String);
end Read_Version;
```

3.4 Display Routines

The display routines are provided primarily for learning and debugging purposes. The HDF Group tools include programs for displaying the Data Descriptors contained within a file as well as for "dumping" various objects of particular data elements. A portion of the test software developed to exercise the functionality of the library performs similarly.

```
Procedure Display_Version
   (Version : Hdf_Types.Version_Record_Ptr)
   is
begin
   Put ("Major_V: " &
        Interfaces.Unsigned_32'Image
        (Version.Major_V));
   Put (", Minor_V: " &
        Interfaces.Unsigned_32'Image
        (Version.Minor_V));
   Put (", Release: " &
        Interfaces.Unsigned_32'Image
        (Version.Release));
   Put_Line (", Name: " &
             Version.The_String.all);
end Display_Version;
```

4. PERFORMANCE

To date, only a single application (other than test programs) is being developed against the library - to generate digital elevation model data from the stereo-correlation of multiple images in ASTER data sets. Performance of reading in the portion of the data sets necessary to determine the regions of interest appears adequate (reading in the necessary data descriptions, finding, allocating space for and reading in two arrays containing 20 million plus IEEE 64 bit floating point values each from a 118 megabyte HDF file takes approximately 20 seconds.)

Of the approximately three thousand lines of code developed about a quarter are type declarations, a quarter parsing routines, a quarter display routines (for testing and debugging) and the remaining quarter for actually reading the data at the lowest level. An additional thousand lines of test tool software was also completed.

5. LIMITATIONS

5.1 Subset Implementation

The library described in this article was developed for only a specific effort and has some significant limitations. First, it was

developed only for reading HDF files, not for creating them. While this may be a significant matter for more general purpose use, it was acceptable here. Second, only procedures necessary to parse and access the data types used in the NASA ASTER data were developed. While this is a significant amount of functionality, it did not cover all possible input data. No attempt was made to mimic the structure or style of the available implementations.

5.2 Lifecycle Considerations

Only a limited amount of effort was put into making the library memory-friendly and none into making it tasking-safe and there are certainly still memory leaks, which could prove fatal in other large applications. The library actually implements an obsolete (from the point of view of "support") version 4 of HDF. Consideration was given to implementing version 5 and using supplied tools to convert between the two versions of the format, but given that the individual files under consideration were already in the hundreds of megabytes (and there were tens of thousands of them!), it seemed prudent to create the library in the native format of the files rather than convert files and have to handle either additional processing or intermediate storage.

The library was initially developed on Linux / x86 and ported to Solaris / Sparc to utilize a larger address space and available memory (the available Sparc equipment utilized a 64 bit Solaris operating system and had 16GB of memory [Nyberg]) in anticipation of future modifications to support a multicore architecture. The porting required a modification of only five low level reading procedures to account for a mechanism to specify the underlying architecture and byte-swapping differences between the two architectures.

```
procedure Read_Unsigned_32
    (File_Identifier : Hdf_Io.File_Type;
     Result : out Interfaces.Unsigned_32) is
   X : Hdf_Types.Unsigned_8_Array_4;
      begin
   if Machine = X86 then
      Hdf_Io.Read (File_Identifier, X (4));
      Hdf_Io.Read (File_Identifier, X (3));
      Hdf_Io.Read (File_Identifier, X (2));
      Hdf_Io.Read (File_Identifier, X (1));
   elsif Machine = Sparc then
      Hdf_Io.Read (File_Identifier, X (1));
      Hdf_Io.Read (File_Identifier, X (2));
      Hdf_Io.Read (File_Identifier, X (3));
      Hdf_Io.Read (File_Identifier, X (4));
   else
      Put_Line ("ERR:hdf.read_unsigned_32");
      raise Program_Error;
   end if;

end Read_Unsigned_32;
```

5.3 Usability Considerations

The naming convention used in the HDF specification was used for name selection through the implementation. Some items (*e.g.*, constants, enumerations, etc.) have direct correspondence to the same items in the C implementation. Other items (particularly record types, procedure names, etc.) have names more in keeping with the Ada style.

6. PRIOR AND RELATED WORK

The HDF Group (http://www.hdfgroup.org) provides documentation and tools for operating with HDF files in C, Fortran and Java. These tools allow for creation, updating and reading HDF files. At one time, an Ada 95 binding to HDF5 had been developed [5], but it has been lost to antiquity.[1]

The documentation is excellent and provided a more than adequate basis for the development of the library for reading the collection of files acquired under this effort. It is anticipated that extending the library to operate with additional data elements as they are acquired will be straightforward.

7. ACKNOWLEDGMENTS

Thanks to Richard Biby for the suggestion to do something interesting with the ASTER data. Thanks also to David Emery for encouraging the writeup of this approach and to both he and Bruce Barkstrom for reviewing rough drafts of it.

8. REFERENCES

[1] http://asterweb.jpl.nasa.gov

[2] http://www.hdfgroup.org/products/hdf4/

[3] http://www.w3.org/TR/REC-xml/

[4] http://www.science.aster.ersdac.or.jp/en/documnts/pdf/ASTER_Ref_V1.pdf

[5] Barkstrom, Bruce R., "Ada 95 Bindings for the NCSA Hierarchical Data Format". In *SIGAda Ada Letters* Volume XXI, Issue 4 (December 2001) SIGAda 2001 09/01 Bloomington, MN, USA. 27-30.

[6] http://www.hdfgroup.org/about/history.html

[7] Nyberg, Karl A. "A Constructive Approach to Integer Factorization"; http://www.grebyn.com/t1000

[8] Nyberg, Karl A. "Automatically Generating DTD-Specific XML Parsers"; *SIGAda Ada Letters*, *to appear...*

[1] In early January 2010, it appeared that a copy of this binding had been found and preparations were being made to have it released from the appropriate government agencies.