A Maintainer's Guide

for the Datatype Module in HDF5 Library

Raymond Lu

This document explains the design, architecture, organization, and algorithms of the datatype module in the HDF5 library.

Introduction

The purpose of this document is to explain the basic design of the datatype module in the HDF5 library – its architecture, organization, and algorithms. For the maintainers of the library, this document should give them enough knowledge to understand, adjust, or fix the library if any problem arises. For the users of the library, the existing documents such as the User's Guide, the Reference Manual, and the File Format Specification should give them sufficient knowledge to use the library. But if any power user wants to find out how the library is designed, this document can be helpful to some extent. This document is written based on the HDF5 release 1.8.8.

The Way That the Library Defines Data Types

Integers

Integers generally have simple bit patterns. Using the twos-complement notation, a signed integer of `n` bits in size will have a range from -2^{n-1} to $2^{n-1} - 1$. The high-order bit is the sign bit. There are `n-1` data bits. For unsigned integers, the high-order bit becomes a data bit. All the `n` bits are data bits. So an unsigned integer of `n` bit in size has a range from `0` to 2^n-1. An example bit sequence of `(signed) char` of 1 byte long is like `10010111`. The high-order (leftmost) bit is set to `1`, meaning the value is negative. If the same bit sequence represents an `unsigned char`, the high-order bit becomes a data bit, making the value be `151`.

In the HDF5 library, each integer data type, predefined or user-defined, has the following properties:

`Order`	`The byte order – big or little endian`
`Sign`	`Signed or unsigned`
`Size`	`The size of the entire integer data type`

`Precision`	`The size of the actual data part of the integer`
`Offset`	`The start of the actual data in the data type`
`Lsb padding`	`The padding bit in the least significant side`
`Msb padding`	`The padding bit in the most significant side`

These properties help the library define or identify each integer type. For example, the following properties define a four-byte little-endian signed integer:

Order	little-endian
Sign	signed
Size	4 bytes
Precision	32 bits
Offset	0
Lsb padding	0
Msb padding	0

The library provides API functions to query or adjust these properties, such as `H5Tset(get)_size`, `H5Tset(get)_order`, `H5Tset(get)_precision`, `H5Tset(get)_offset`, `H5Tset(get)_sign`, and `H5Tset(get)_pad`. These functions also work for other atomic data types, i.e. floating-point numbers.

Floating-Point Numbers

The floating-point number representation is more complicated. A more thorough description of IEEE standard floating-point numbers can be found in the *IEEE Standard 754* document. For IEEE standard floating-point numbers, there are three components for a floating-point number - the sign, the exponent, and the mantissa. The diagram below shows the layouts of IEEE `float` and `double` types.

Type	Sign	Exponent	Mantissa	Bias
Float	1[31]	8[30-23]	23[22-00]	127
Double	1[63]	11[62-52]	52[51-00]	1023

In the table, the numbers are the size of each component. The bit index is in the square brackets. To

calculate the true exponent value, the bias has to be subtracted from the value represented by the bits of exponent. The mantissa represents the precision bits. The leading bit has been implied. When the true precision is calculated, this implicit bit will be restored. Consider this bit sequence for float in little-endian order,

```
 Byte 3      byte 2      byte 1      byte 0
11000011    11110000    00000000    00000000
```

The high-order (leftmost) bit is the sign bit. It is set to indicate the number is negative. The eight bits after the sign bit, `10000111` in `byte 3` and `2`, is the exponent. The value of these eight bits is `135`. After subtracting the bias `127`, the true exponent is `8`. The 23 bits after the exponent `1110000 00000000 00000000` in `byte 2, 1, 0`, is the mantissa. After restoring the implicit leading bit and adding the radix, the mantissa becomes `1.1110000 00000000 00000000`. The value of this float number is $1.111 \times 2^8 = 111100000.0$ in binary. Adding the sign bit, that value is `-480.0` in decimal.

There are a few special values for floating-point numbers,

Denormalized – when exponent bits are all 0s but mantissa bits are non-zero. There will be no implicit bit for the mantissa.

Zero – when exponent and mantissa bits are all set to 0s. There can be both +0 and -0.

Infinity – when exponent bits are all 1s and mantissa bits are all 0s. There can be both positive and negative infinities.

NaN(Not a Number) – when exponent bits are all 1s and mantissa bits are not all 0s. NaN can be either positive or negative.

For other predefined or used-defined types, they should be similar to IEEE standard. There should be the sign, exponent, mantissa, and bias. The bits of exponent or mantissa should be contiguous. The floating-point numbers for VAX are different from IEEE standard. Their byte order is a mixture of little-endian and big-endian. But that is the only difference from IEEE standard. So we will not discuss it in detail here.

The properties of floating-point number datatypes are more complicated than integer types. Each floating-point datatype, predefined or user-defined, has the following properties:

Order	The byte order – big endian, little endian, or VAX
Size	The size of the entire data type
Precision	The size of the actual data part of the data type

Offset	The start of the actual data in the data type
Lsb padding	The padding bit in the least significant side
Msb padding	The padding bit in the most significant side
Sign	The position of the sign bit
Exponent position	The position of the start of exponent bits
Exponent size	The number of the exponent bits
Exponent bias	The value of exponent bias
Mantissa position	The position of the start of mantissa bits
Mantissa size	The number of the mantissa bits
Norm	The flag for normalized floating number
Padding	The padding bit

For example, an IEEE standard little-endian single floating-point number is four bytes in size and thirty-two bits in precision. Its sign bit is at the thirty-first bit. The exponent is eight bits long and starts at the twenty-third bits. The mantissa is twenty-three bits long and starts at the beginning bit. We can use the following diagram to illustrate this floating number:

```
  byte 3   byte 2    byte 1   byte 0
SEEEEEEE  EMMMMMMM  MMMMMMMM  MMMMMMMM
```

So it has the following properties:

Order	little-endian
Size	4 bytes
Precision	32 bits
Offset	0
Lsb padding	0
Msb padding	0
Sign	31

Exponent position	23

Exponent size	8
Exponent bias	127
Mantissa position	0
Mantissa size	23
Norm	Implied
Padding	0

Besides the API functions for atomic datatypes, the library provides several functions for floating numbers specifically. These functions are `H5Tset(get)_fields`, `H5Tset(get)_ebias`, `H5Tset(get)_norm`, and `H5Tset(get)_inpad`.

1) 2.3 Predefined Numerical Datatypes

2) 2.3.1 Integers

The integer datatypes include all the library's predefined integers and any user-defined integers. The library's predefined integers include standard, Unix-specific, Intel-specific, Alpha-specific, MIPS-specific, ANSI C9x-specific, and native data types. The *HDF5 Predefined Datatypes* section in the *HDF5 Reference Manual* lists all these predefined data types.

3) 2.3.2 Floating-Point Numbers

The floating-point datatypes include all the library's predefined floating numbers and any user-defined floating numbers. The library's predefined floating numbers include IEEE standard and C native datatypes. The *HDF5 Predefined Datatypes* section in the *HDF5 Reference Manual* lists all these predefined datatypes.

4) 2.4 User-Defined Numeric Datatypes

Users can define their own datatypes based on the default datatypes in the library. By adjusting the properties of the existent data types through some API functions for data types, users can create new datatypes. These API functions are listed under the *Atomic Datatype Properties* category of *H5T Datatype Interface*. After defining the properties, users should call H5Tcommit to register the datatype into the file.

For example, a user defines a two-byte big-endian unsigned integer. But its precision is only ten bits long. The offset is four bits. The padding is one. We can represent this integer as

```
Byte 0      byte 1
1111XXXX  XXXXXX11
```

where `X` stands for the data part and `1` stands for the padding. This integer should have the following properties:

Order	big-endian

Sign	unsigned
Size	2 bytes
Precision	10 bits
Offset	4 bits
Lsb padding	1

Msb padding	1

Another example is a user-defined three bytes big-endian floating number. Its precision is eighteen bits. The offset is four bits. The other properties are displayed below:

Order	big-endian
Size	3 bytes
Precision	18 bits
Offset	4 bits
Lsb padding	0
Msb padding	0
Sign	19
Exponent position	13
Exponent size	6
Exponent bias	31
Mantissa position	2
Mantissa size	11
Norm	Implied
Padding	0

We can use the following diagram to illustrate this floating number:

```
  byte 0    byte 1      byte 2
0000SEEE   EEEMMMMM   MMMMMM00
```

5) 2.5 Non-Numerical Datatypes

The datatypes (integers and floating-point numbers) we discussed above are numerical. There are non-numerical datatypes in the library. Some of them are derived from the numerical datatypes, such as *enum* and *array* types. The library does not have default data types for these non-numerical types. Users must define them. It is necessary to define a few terms that we normally use to describe the kinds of data types in the library. Please see the *Terminology* for the definitions of these terms.

6) 2.5.1 String Datatypes

The string types are atomic datatypes. They have the following properties:

Cset	ASCII or Unicode character set
Pad	space or null padding for extra bytes

7) The functions that the library provides to query or adjust these properties are `H5Tset(get)_cset` and `H5Tset(get)_strpad`.

8) 2.5.2 Reference Datatypes

The reference types are another kind of atomic datatypes. A reference datatype only has one property:

Rytpe	object or region reference

9) The functions for reference datatypes are under the H5R interface, such as `H5Rcreate, H5Rdereference`, and `H5Rget_obj_type`.

10) 2.5.3 Compound Datatypes

A HDF5 compound datatype can contain any HDF5 data type as its member. All the properties for compound datatypes are related to its members, such as:

`Nmembs`	`The number of member types`
`Sorted`	`How the members are sorted`
`Packed`	`whether the members packed together`
`Members`	`information about each member`

Besides its own properties as a HDF5 datatype, each member has the following individual properties:

`Name`	`the name of this member`
`Size`	`the size of this data type`
`Offset`	`the offset from the beginning of the C struct`

The functions that the library provides to query or adjust these properties are `H5Tinsert, H5Tpack, H5Tget_nmembers, H5Tget_member_class, H5Tget_member_name, H5Tget_member_index, H5Tget_member_offset`, and `H5Tget_member_type`.

11) 2.5.4 Enumerate Datatypes

Enumerate datatypes are derived from integers. They have the following properties:

`Nmembs`	`number of members`
`Sorted`	`how the members are sorted`
`Names`	`member names`
`Values`	`member values`

12) The library provides these API functions to create enumerate datatypes or query their properties: `H5Tenum_create, H5Tenum_insert, H5Tenum_nameof, H5Tenum_valueof, H5Tget_member_value, H5Tget_nmembers, H5Tget_member_name`, and `H5Tget_member_index`.

13) 2.5.5 Variable-length Datatypes

The variable-length datatype is a derived datatype. It has the following properties:

`Type`	`string or sequence of other type`
`Cset`	`character type for VL string`
`Pad`	`space or null padding for extra bytes for VL string`

14) The API functions that the library provides to create and query variable-length datatypes are `H5Tvlen_create` and `H5Tis_variable_str`.

15) 2.5.6 Array Datatypes

The array data type is a derived data type. Its base type can be any HDF5 data type. The array datatype has the following properties:

`Nelem`	`total number of elements in the array`
`Ndims`	`number of dimensions`
`Dim[ ]`	`sizes of dimensions`

16) The API functions that the library provides to create or query array datatypes are `H5Tarray_create, H5Tget_array_ndims`, and `H5Tget_array_dims`.

17) 2.5.7 Opaque Datatype

The opaque datatype only has one property:

```
      Tag                      short description string
```

The library provides these two API functions, `H5Tset(get)_tag`, to query or adjust the property of opaque datatypes.

Library's Internal Design for Datatypes

The Architecture of Datatype Module

The following diagram illustrates the basic design of the data type module in the library. The left side of the figure focuses on how the library creates data types and the conversion table. The right side of the figure focuses on the relationship of the conversion table with the IO flow. We will explain the detail of the library's internal design using this diagram.