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 -2n-1 to 2n-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 2n–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 x 28 = 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.

## 2.3 Predefined Numerical Datatypes

### 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.

### 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.

## 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

## 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.

### 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

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

### 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

### The functions for reference datatypes are under the H5R interface, such as H5Rcreate, H5Rdereference, and H5Rget\_obj\_type.

### 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.

### 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

### 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.

### 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

### The API functions that the library provides to create and query variable-length datatypes are H5Tvlen\_create and H5Tis\_variable\_str.

### 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

### The API functions that the library provides to create or query array datatypes are H5Tarray\_create, H5Tget\_array\_ndims, and H5Tget\_array\_dims.

### 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.

Predefined (native) datatypes

H5T\_NATIVE\_CHAR H5T\_NATIVE\_INT

H5T\_NATIVE\_FLOAT

:

Conversion

table

H5T.c

Predefined (standard) datatypes

H5T\_STD\_I8BE

H5T\_STD\_U16LE

H5T\_IEEE\_F32BE

H5T\_IEEE\_F64LE

:

User-defined datatypes

H5Tconv.c

H5detect.c

H5Tinit.c

Application

data in

memory

No conversion

H5Dwrite H5Dread

H5T API

functions

H5I\_register

Filter pipeline

Data in file

Application

H5T\_init\_interface

H5T\_path\_find

H5T\_convert

H5Tregister / H5T\_register

## Source File That Contains Datatype Properties

Inside the library, all the properties of the data types are contained in the structures. These structures are defined in H5Tpkg.h and used in memory by the library. The diagram below shows the relationship among these structures. The library developers may want to look at them for the information of the data type properties. The *HDF5 File Format Specification* describes how a HDF5 file stores the data type properties.

## How H5detect.c Works

When a user builds the HDF5 library with Makefile or CMake, the first source file to be built and run is H5detect.c under the library source directory. Running the executable of H5detect.c generates another source file called H5Tinit.c under the user’s build directory. The function H5TN\_init\_interface in H5Tinit.c contains all the property information for the library’s predefined native data types (integers and floating numbers). Then H5Tinit.c is compiled with other source files to build the library. The program H5detect.c detects the properties of all the predefined native data types, such as H5T\_NATIVE\_INT, H5T\_NATIVE\_FLOAT, H5T\_NATIVE\_UINT64, H5T\_NATIVE\_INT\_LEAST64, and H5T\_NATIVE\_INT\_FAST32.

### Integers

In H5detect.c, the macro DETECT\_I is used to detect the properties of native integers. It has the following signature:

 DETECT\_I (TYPE, VAR, INFO)

In the macro’s signature, TYPE is the native type in C such as int or long. VAR is the type name used to construct the identifier of the predefined data type. For example, if VAR is INT, the identifier for int is H5T\_NATIVE\_INT. Please see Part 2.3.1 for the list of some predefined native types. The INFO is a C struct containing properties for both integer and floating number. The information of these properties will be printed out into H5Tinit.c.

#### Byte order

In the definition of the macro DETECT\_I, it first tries to figure out the byte order of the data type (e.g. int) by this algorithm:

 int v;

 unsigned char \*x;

 for(i=sizeof(int), v=0; i>0; --i)

 v = (v << 8) + i;

 for(i = 0; x = &v; i < sizeof(int); i++) {

 j = (\*x++) -1;

 d\_g[nd\_g].perm[i] = j;

 }

If the machine is little-endian, the first loop fills each byte with sequential numbers

 byte 3 byte 2 byte 1 byte 0

 4 3 2 1

The second loop fills the array perm[] with the following numbers

 perm[0] perm[1] perm[2] perm[3]

 0 1 2 3

When the program H5detect.c prints the results into H5Tinit.c, this sequence of value is considered as little-endian. The reversed order is considered as big-endian.

#### Offset

H5detect.c also uses the permutation perm[] to decide the precision and offset of the native integer types. The function precision() checks whether the beginning or ending of perm[] is -1. If the beginning or ending bytes are value -1, they are padding, assuming the offset is always in whole byte. The precision is the size of the type subtracting the offset.

#### Alignment restriction

H5detect.c also tries to detect the alignment restriction of a data type. “Some computers allow data objects to reside in storage at any address regardless of the data’s type. Others impose alignment restrictions on certain data types, requiring that objects of those types occupy only certain addresses. It is not unusual for a byte-addressed computer, for example, to require that 32-bit integers be located on addresses that are a multiple of four. In this case, we say that the ‘alignment modulus’ of those integers is four.”[[1]](#footnote-1)

The macro ALIGNMENT(TYPE,INFO) in H5detect.c detects the information of alignment restriction for integers and floating numbers. Again, TYPE is the native type in C such as int or long. The INFO is a C struct containing properties for both integer and floating number. The basic algorithm can be expressed in the following semi-pseudo code:

 align\_value[]={1,2,4,8,16};

 char \*buf;

 TYPE value2, type\_value = 1;

 if (setjmp(jmp\_buf)) align\_num++;

 if(little\_endian)

/\* Copy the type\_value to the point of the buffer where

 \* the assumed alignment is added \*/

 memcpy(buf+align\_value[align\_num], &type\_value, sizeof(int));

 else /\* big-endian \*/

 /\* Skipped \*/

 /\* Cast the value in the buffer to another variable \*/

 value2 = \*(TYPE\*)(buf+align\_value[align\_num];

 if(type\_value != value2)

 /\* Alignment isn’t found. Go back to setjmp.

 \* Increment alignment value and try again. \*/

 longjmp(jmp\_buf, 1);

 /\* We have found the alignment \*/

 INFO.align = align\_value[align\_num];

In the algorithm, the code between setjmp and longjmp is equivalent to a loop. setjmp saves the current environment into jmp\_buf. jmp\_buf is used by longjmp to restore the program state. Just imagine that longjmp makes the program jump back to the point where setjmp saves the environment.

#### Alignment in structure

The alignment of a data type as a C structure member refers to the value expressed in the following pseudo code:

 struct {

 Char c;

 TYPE x;

 } s;

 COMP\_ALIGN = (CHAR\*)(&(s.x)) – (char\*)(&s);

This piece of code is actually the definition of the macro COMP\_ALIGNMENT(TYPE,COMP\_ALIGN) in the H5detect.c program. A C structure normally has it own alignment restriction. It must terminate on the same alignment boundary on which it starts. If it starts on an even byte boundary, it must also end on an even byte boundary. If the first member of the structure is a one-byte character, the space between this character and next member is the alignment of the next member type as a structure member. For example, if the TYPE in the code above is int and the storage of the structure is as below:

 x

 (space)

 c

 Bytes 1 3 4

The alignment of int as a structure member is 4 bytes.

### Floating-Point Numbers

In H5detect.c, the macro DETECT\_F is used to detect the properties of native floating numbers. It has the following signature:

 DETECT\_F(TYPE,VAR,INFO)

In the macro’s signature, TYPE is the native type in C such as float or double. VAR is the type name used to construct the identifier of the predefined data type. For example, if VAR is INT, the identifier for float is H5T\_NATIVE\_FLOAT. Please see Part 2.3.2 for the list of some predefined native types. The INFO is a C struct containing properties for both integer and floating number. The information of these properties will be printed out into H5Tinit.c.

#### Byte order

In the definition of the macro DETECT\_F, it first tries to figure out the byte order of floating number type (e.g. float) by this algorithm:

 int i, j, k, first\_mbyte = -1, last\_mbyte = -1;

 float value1, value2, value3;

 unsigned char buf1[sizeof(float)], buf3[sizeof(float)];

 int perm[32];

 for(i = 0, value1 = 0.0, value2 = 1.0; i < (int)sizeof(float); i++) {

 value3 = value1;

 value1 += value2;

 value2 /= 256.0;

 memcpy(buf1, (const void \*)&value1, sizeof(float));

 memcpy(buf3, (const void \*)&value3, sizeof(float));

 /\* Found out the first different byte \*/

 j = byte\_cmp(sizeof(float), &buf3, &buf1);

 /\* Record the first different byte in permutation \*/

 if(j >= 0) {

 if(0 == i || perm[i - 1] != j) {

 perm[i] = j;

 last\_mbyte = i;

 if(first\_mbyte < 0)

 first\_mbyte = i;

 }

 }

 }

In this code, we only want to find out the byte order of the mantissa part of the data type, assuming the exponent part has the same byte order as the mantissa. The byte\_cmp function simply finds the first different byte between two buffers. If no difference found, it returns -1. We can use the little-endian 4-byte float as the example to explain how the algorithm works. The following diagram shows the properties of the example data type:

 Byte 3 byte 2 byte 1 byte 0

 SEEEEEEEE

 MMMMMMMM

 MMMMMMMM

 EMMMMMMM

During iteration 1:

 i = 0 value1 = 1.0 value2 = 1/256 = 1/28 value3 = 0.0

 buf1[ ]= {0x00, 0x00, 0x80, 0x3f} or in binary format:

 buf1[3] buf1[2] buf1[1] buf1[0]

00111111 10000000 00000000 00000000

 buf3[ ] = {0x00, 0x00, 0x00, 0x00} or in binary format:

 buf3[3] buf3[2] buf3[1] buf3[0]

 00000000 00000000 00000000 00000000

 j = 2 perm[0] = 2 last\_mbyte = 2 first\_mbyte = -1

During iteration 2:

 i = 1 value1 = 1.0 + 1/28 value2 = 1/216 value3 = 1.0

 buf1[ ] = {0x00, 0x80, 0x80, 0x3f} or in binary format:

 buf1[3] buf1[2] buf1[1] buf1[0]

00111111 10000000 10000000 00000000

 buf3[ ] = {0x00, 0x00, 0x80, 0x3f} or in binary format:

 buf3[3] buf3[2] buf3[1] buf3[0]

 00111111 10000000 00000000 00000000

 j = 1 perm[1] = 1 last\_mbyte = 2 first\_mbyte = 1

During iteration 3:

 i = 2 value1 = 1.0 + 1/28 + 1/216 value2 = 1/224 value3 = 1.0 + 1/28

 buf1[ ] = {0x80, 0x80, 0x80, 0x3f} or in binary format:

 buf1[3] buf1[2] buf1[1] buf1[0]

00111111 10000000 10000000 10000000

 buf3[ ] = {0x00, 0x80, 0x80, 0x3f} or in binary format:

 buf3[3] buf3[2] buf3[1] buf3[0]

 00111111 10000000 10000000 00000000

 j = 0 perm[2] = 0 last\_mbyte = 2 first\_mbyte = 0

During iteration 4:

 i = 3 value1 = 1.0 + 1/28 + 1/216 + 1/224 value2 = 1/232 value3 = 1.0 + 1/28  + 1/216

 buf1[ ] = {0x80, 0x80, 0x80, 0x3f} or in binary format:

 buf1[3] buf1[2] buf1[1] buf1[0]

00111111 10000000 10000000 10000000

 buf3[ ] = {0x80, 0x80, 0x80, 0x3f} or in binary format:

 buf3[3] buf3[2] buf3[1] buf3[0]

 00111111 10000000 10000000 10000000

 j = -1 perm[3] = -1 last\_mbyte = 2 first\_mbyte = 0

After the loop, the permutation array has the values perm[ ] = {2, 1, 0}. Then DETECT\_F calls the function fix\_order to adjust the permutation to the byte order of little-endian. The permutation becomes perm[ ] = {0, 1, 2}, which is the same as the mantissa part of the diagram above. This byte order detection handles little-endian, big-endian, and VAX.

#### Implied mantissa bit

Next, DETECT\_F tries to figure out whether the mantissa has an implied bit in the function called imp\_bit. Some floating-point formats discard the most significant bit of the mantissa after normalizing since it will always be one except for the special value 0.0. In DETECT\_F, it calls imp\_bit in this way:

 \_v1 = 0.5;

 \_v2 = 1.0;

 INFO.imp = imp\_bit (sizeof(TYPE), INFO.perm, &\_v1, &\_v2);

imp\_bit is defined as below:

int imp\_bit(int n, int \*perm, void \*\_a, void \*\_b)

{

 unsigned char \*a = \_a;

 unsigned char \*b = \_b;

 int changed, byte\_index, bit\_index;

 int msmb; /\*most significant mantissa bit \*/

 /\*

 \* Look for the least significant bit that has changed between

 \* A and B. This is the least significant bit of the exponent.

 \*/

 changed = bit\_cmp(n, perm, a, b);

 /\*

 \* The bit to the right (less significant) of the changed bit should

 \* be the most significant bit of the mantissa. If it is non-zero

 \* then the format does not remove the leading `1' of the mantissa.

 \*/

 msmb = changed - 1;

 byte\_index = msmb / 8;

 bit\_index = msmb % 8;

 return (a[perm[major]] >> minor) & 0x01 ? 0 : 1;

}

The function bit\_cmp (explained later) compares two bit vectors and returns the index for the first bit that differs between the two vectors. For example, if float is four-bytes little-endian with the most significant mantissa bit implied, the bit sequence for the value of 0.5 represented by \*a in imp\_bit is:

 a[3] a[2] a[1] a[0]

00011111 00000000 00000000 00000000

The bit sequence for the value of 1.0 represented by \*b in imp\_bit is:

 b[3] b[2] b[1] b[0]

00011111 10000000 00000000 00000000

The value of changed return from bit\_cmp is 23. The value of the most significant mantissa bit (msmb) is 22. The value of the byte index where the msmb falls into is 2. The value of the bit index where the msmb falls into the byte is 6. The final step for the returned value is

perm[2] = 2

a[2] = 0x00

(a[2] >> 6) & 0x01 = 0

 return 1

which is interpreted as implied in print\_results.

#### Sign bit

To figure out the location of the sign bit is relatively simple:

 \_v1 = 1.0;

 \_v2 = -1.0;

 INFO.sign = bit\_cmp (sizeof(TYPE), INFO.perm, &\_v1, &\_v2);

Now we can explain how bit\_cmp works:

int bit\_cmp(int nbytes, int \*perm, void \*\_a, void \*\_b)

{

 int i, j;

 unsigned char \*a = (unsigned char \*) \_a;

 unsigned char \*b = (unsigned char \*) \_b;

 unsigned char aa, bb;

 for (i = 0; i < nbytes; i++) {

 /\* Find out where the different byte is \*/

 if ((aa = a[perm[i]]) != (bb = b[perm[i]])) {

/\* Find out where the different least-significant bit

 \* by right-shifting the variables 1-bit at a time.

 \*/

 for (j = 0; j < 8; j++, aa >>= 1, bb >>= 1) {

 /\* If the least-significant bit is different,

 \* return the bit index. \*/

 if ((aa & 1) != (bb & 1))

return i \* 8 + j;

 }

 }

 }

 return -1;

}

#### Size of mantissa

DETECT\_F checks the difference between the values of 1.0 and 1.5 to find out the size of mantissa. The values of 1.0 and 1.5 differ at the first bit of mantissa if the machine has implied mantissa bit, or at the second bit if the machine imply the first mantissa bit. The starting bit of mantissa is assumed to be the first bit of the data type.

#### Exponent

 DETECT\_F assumes the exponent is between the sign bit and the mantissa. So finding the position and size of exponent becomes very simple.

#### Bias

 When a floating number has the value 1.0, the value of its exponent is its bias. After normalization, the value 1.0 is represented by 1.0 x 20. Whatever the value of the exponent is is the bias for the floating number type.

#### Precision, alignment, and alignment in structure

Finding the precision, alignment restriction, and alignment in structure is the same as integers.

### Others

H5detect.c also detects the alignments in structure for several other things, such as pointers, hvl\_t, hobj\_ref\_t, and hdset\_reg\_ref\_t.

### H5Tinit.c

The print\_results in H5detect.c prints all the properties of predefined data types of integers and floating numbers into H5Tinit.c. We mentioned earlier that H5Tinit.c is located under the build directory if it is different from the directory of the library’s source code. After each property is assigned, the data type is registered by calling H5I\_register. An identification is attained.

## Other Predefined (Standard) Data Types

The other predefined (standard) data types are defined in H5T\_init\_interface in H5T.c using some complicated macros. The one that does the major work is H5T\_INIT\_TYPE. We will use H5T\_INIT\_TYPE(SINTBE,H5T\_STD\_I32BE\_g,COPY,native\_int,SET,4) as an example. It registers data types in four steps:

1. Gets the data type structure of the base type. The example will call H5T\_INIT\_TYPE\_COPY\_CREATE(native\_int) in this step.
2. Adjusts the size and precision for the new type. The example will call H5T\_INIT\_TYPE\_SET\_SIZE(4) in this step.
3. Adjusts other properties for the new type. The example calls H5T\_INIT\_TYPE\_SINTBE\_CORE, which calls H5T\_INIT\_TYPE\_SINT\_COMMON(H5T\_ORDER\_BE) in turn, which calls H5T\_INIT\_TYPE\_NUM\_COMMON(H5T\_ORDER\_BE) in turn.
4. Registers the new type.

# Data Conversion

## Hard vs. Soft Conversion

Internally, the library has hard and soft conversion functions for data types. A hard conversion is basically a casting done by a compiler. A soft conversion is done by the library’s own conversion function. The library maintains a conversion table. It contains both hard and soft conversion functions. A hard conversion function is for a pair of source and destination data types. A soft conversion function is for a pair of source and destination data type classes. The library’s default conversion between predefined data types is hard conversion.

## Registering Functions in the Conversion Table

The library registers its default conversion functions, either hard or soft, into the conversion table in the initialization stage (currently in the function H5T\_init\_interface in H5T.c). It registers soft conversion functions first, which is more general. The hard conversion functions are the next, which is more specific. When the library tries to convert data, it always picks the more specific conversion function first. When hard conversion function is not available, it goes with the more general soft conversion function. The table below lists most of the soft conversion functions in the library.

|  |  |  |  |
| --- | --- | --- | --- |
| Source data type | Destination data type | Conversion function | Function type |
| Integer | Integer | H5T\_conv\_i\_i | Soft |
| Integer | Floating number | H5T\_conv\_i\_f | Soft |
| Floating number | Floating number | H5T\_conv\_f\_f | Soft |
| Floating number | Integer | H5T\_conv\_f\_i | Soft |
| String | String | H5T\_conv\_s\_s | Soft |
| Bitfield | Bitfield | H5T\_conv\_b\_b | Soft |
| One byte order | Another byte order | H5T\_conv\_order | Soft |
| Compound | Compound | H5T\_conv\_struct | Soft |
| Enum | Enum | H5T\_conv\_enum | Soft |
| Variable-length | Variable-length | H5T\_conv\_vlen | Soft |
| Array | Array | H5T\_conv\_array | Soft |

The following table shows some examples of hard conversion functions between floating number types.

|  |  |  |  |
| --- | --- | --- | --- |
| Source data type | Destination data type | Conversion function | Function type |
| Float | Double | H5T\_conv\_float\_double | Hard |
| Double | Float | H5T\_conv\_double\_float | Hard |
| Float | Long double | H5T\_conv\_float\_ldouble | Hard |
| Double | Long double | H5T\_conv\_double\_ldouble | Hard |
| Long double | Float | H5T\_conv\_ldouble\_float | Hard |
| Long double | Double | H5T\_conv\_ldouble\_double | Hard |

Both of the soft and hard conversion functions are defined in H5Tconv.c.

If users want to register their own conversion function, whether soft or hard function, they can use the API function H5Tregister. This function will replace the library’s default conversion function. Users can also use H5Tunregister to unregister a conversion function.

## Converting Data

When a user application tries to read or write data and the source and destination data types are different, or when it tries to call H5Tconvert to convert data, data conversion happens. The first thing that the library does is to search for the right conversion function of the source and destination data types (This is done through the H5T\_path\_find function in H5T.c). Then it calls H5T\_convert to use the conversion path to convert the data. This process is clearly shown in the definition of the API function H5Tconvert in H5T.c. The same process happens when a program is trying to read or write data.

## The Test dt\_arith.c

## We need to explain the basic method of testing the data conversion in dt\_arith.c. In each test, it initializes a data buffer of the source data type. If the data is integer, the macro INIT\_INTEGER fills the data buffer with bit sequences like

## 00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000,

## 00000000, 00000011, 00000111, 00001111, 00011111, 00111111, 01111111, 11111111,

## 11111111, 11111110, 11111100, 11111000, 11110000, 11100000, 11000000, 10000000

## It tries to cover most of bit sequence and avoid any casting, assignment, and comparison at the same time because it is testing some hard conversion functions that involve these operations.

## If the data is floating number, the macro INIT\_FP\_NORM fills the data buffer with values between the minimal and maximal. Each value is the previous value times a multiplication factor (value \*= multiplication factor). The macros INIT\_FP\_DENORM and INIT\_FP\_SPECIAL fill the buffer with some special values.

# Adding Support for New Predefined Data Types

To add support for new predefined native integer types or floating-point number types, simply follow the procedure in H5detect.c using the macros DETECT\_I (explained in 3.3.1) or DETECT\_F (explained in 3.3.2). To add support for new predefined standard integer or floating-point number type, follow the procedure explained in 3.4.

Revision History

|  |  |
| --- | --- |
| *January 20, 2012:* | First draft circulated for comments to Elena, Quincey, Mike M., and Frank.  |

|  |
| --- |
|  |

Appendix:

Terminology

|  |  |
| --- | --- |
| **numerical data type** | The HDF5 data type that deals with number, including integer and floating-point number. |
| **non-numerical data type****atomic data type****derived data type** | The HDF5 data type other than numerical data type.The HDF5 data type that is not composed of other data type, including integer, floating number, string, and reference type.The HDF5 data type that is composed of other data type, including compound, enum, array, variable-length, and opaque type. |
|  |  |

References

1. The HDF Group. “HDF5 Documentation,” <http://www.hdfgroup.org/HDF5/doc/doc-info.html>.
2. Data Conversion of Arithmetic Data Types.
3. Hollasch, Steve. 2003. IEEE Standard 754 Floating Point Numbers.
4. Harbison and Steele, C: A Reference Manual
1. Harbison and Steele, C: A Reference Manual, 6.1.3 Alignment Restrictions. [↑](#footnote-ref-1)