RFC: JSON VOL

**Frank Willmore**

An implementation of the VOL abstraction layer is proposed which enables HDF5 content to be written and read in an alternate bytes-on-disk format. This implementation will serve as reference for the development of other VOL implementations. JANSSON is suggested as the JSON encoding/decoding library. JSON is human-readable, widely used, and relatively easy to implement. The JSON VOL translates HDF5 API calls to JANSSON API calls according to the production rules. JANSSON then encodes/decodes as JSON.

# Introduction

The VOL abstraction layer enables HDF5 application developers to maintain the HDF5 model for data storage, while allowing the flexibility to choose different storage technologies for the backstore. The JSON VOL is specified as a reference implementation for the VOL architecture, designed to generate a file which contains equivalent information to a HDF5 formatted file, but in an alternate bytes-on-disk, human-readable format. The original VOL specification[[1]](#footnote-0) proposed a ‘raw format’ filesystem VOL, which would map the HDF5 filesystem-in-a file format to the native filesystem of the underlying OS. After discussion of the lack of standards and inherent complexity of a growing number of filesystem definitions and implementations, it was decided that it would instead be simpler to generate a single file containing a JSON representation.

Implicit in the production of a JSON VOL is the need to specify mappings for HDF5 data structures into JSON data structures. The HDF5-JSON[[2]](#footnote-1) production rules define a mapping, and the JANSSON library provides a standard set of function calls with which to implement this mapping.

# Approach / Motivation / Etc.

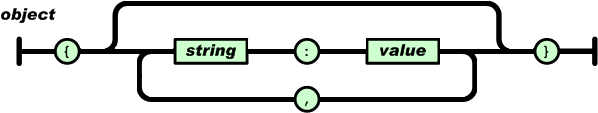
The growing need for an alternate representation of HDF5 data is rapidly becoming apparent and JSON is the emergent choice for a human-readable representation, as it is at once, simple, flexible, well-documented, and widely used. This work leverages work from the HDF5-JSON project which provides a standard set of production rules for interconversion.

As flexible and robust of an approach as possible is attempted, so as to address current and anticipated needs. The recommended approach is the Incorporation of the established JSON-C library JANSSON for the parsing and generation of JSON itself plus a layer to code the actual mappings expressed in the production rules. It was discussed that additional flexibility may be gained by using Google’s protocol buffer as an intermediate layer, however the simplicity of a single implementation language, the limiting of dependencies steered us to JANSSON.

1. **JSON[[3]](#footnote-2)**

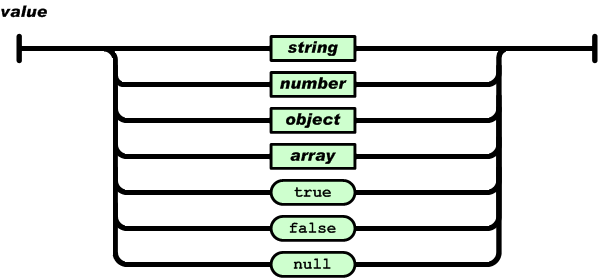
JSON (JavaScript Object Notation) is the *de facto* standard for the interchange of structured data *via* a formatted string, largely subsuming the role of XML. The JSON format itself consists of three data structures: Objects, arrays, and values. Values can be either an object, a number, or a string. Sets of key::value pairs and ordered arrays of objects. The entire grammar of JSON is summarized below, with diagrams borrowed from the official JSON site.

The top-level of a JSON file (or byte array) contains an object. This object is composed of zero or more key-value pairs, where the key is string-valued:

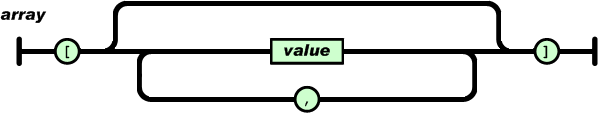


**Figure 1. Flow for generation of JSON object text.**

These values are either a string, a number, an object, an array, a boolean value, or null (no value):

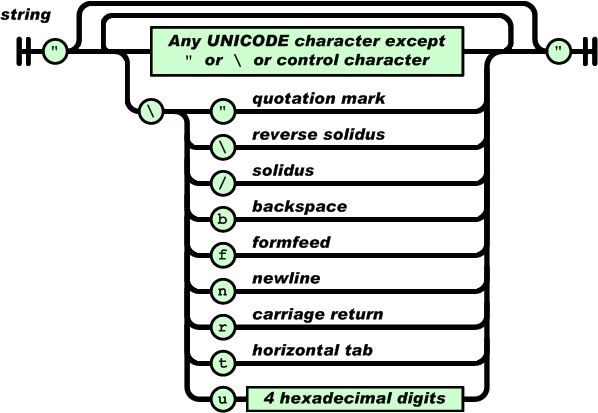


**Figure 2. Flow for inclusion of JSON value structure**



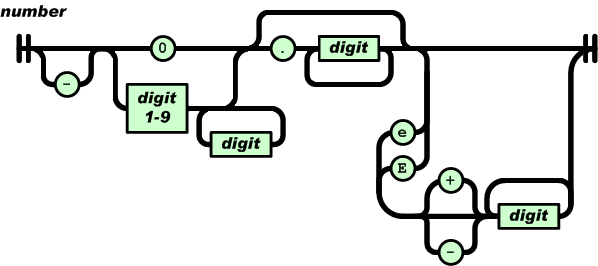
**Figure 3. Flow for generation of JSON array.**

String values can contain the usual characters as well as unicode. All string values are encased by opening and closing quotations.

****

**Figure 4. Flow for generation of JSON string value.**

Number values are recorded in base ten, and do not distinguish between integer and floating point types. They do not inherently specify any given level of precision or distinguish whether zero is to be interpreted as positive or negative, as provisioned in the IEEE-754 floating point standard. Specifics of type and value are left purely to the discretion of the interpreting code.

****

**Figure 5. Flow for generation of JSON number value.**

1. **Candidate JSON parsing and rendering libraries to be used in the implementation of HDF5-JSON-C**

Several libraries exist for the generation of in-memory data structures to represent JSON objects. Some are faster than others, some are more well-documented. Some are more performant and some are more flexible. Three potential options are presented here: JANSSON, YAJL, and protocol buffers. The first two because they have some history of use within HDF Group, and the third because it may ultimately provide the most flexibility.

Implementation of HDF5-JSON-C with either JANSSON, YAJL or a similar library would involve writing C code to manage to manipulate data structures within the selected library according to the HDF5-JSON production rules. Implementation using protocol buffers would require translating the production rules into .proto file, a format which the protocol buffers software would use to generate C++ code, then writing stubs to interface with the generated classes.

* 1. **JANSSON[[4]](#footnote-3)**

Jansson is a well-documented and widely used C library for parsing and rendering JSON. It was used in the original C++ version of HDF5-JSON. Strengths and weaknesses include.

* + 1. + Production-quality library, of mature code.
    2. + Well-documented API.
    3. + Useful example code.
    4. + Originally used in the C++ HDF5-JSON implementation.
    5. + MIT license
  1. **YAJL[[5]](#footnote-4)**

The YAJL (Yet Another JSON Library) is another option for the C language interface to parsing and rendering of JSON. It is used in the current implementation of several REST VOL functions. Strengths and weaknesses include:

* + 1. + Production-quality library, of mature code.
    2. - Documentation could be improved.
    3. + Performant.
    4. + Originally used in the REST VOL for JSON rendering and parsing.
    5. - Non-standard ‘permissive’ license.
  1. **PROTOCOL BUFFERS[[6]](#footnote-5)**

Protocol buffers is an open-source software project maintained by Google. It generates mappings of a user-defined data schema (such as the HDF5-JSON production rules) to classes in a variety of object-oriented languages which can be used to parse and render JSON.

* + 1. + Production-quality library, of mature code.
    2. + Supports multiple output formats (JSON, XML, etc.) for read-write.
    3. - Has C++ but not C extensions (would require additional bridge code plus C++ compiler at build time)
    4. - Requires one-time generation of mapping classes *via* the protoc compiler.
    5. + BSD license.
    6. + Supports multiple language front-ends.
    7. + Useful example code.

***update: JANSSON was selected for its ease of use, existing documentation, and licensing. While flexible, protocol buffers would introduce additional complexity and dependency, and may not generate the same JSON representation as the production rules specify.***

1. **HDF5-JSON production rules**

A complete set of production rules for the mapping of HDF5 objects to JSON objects is outlined in the HDF5-JSON project.[[7]](#footnote-6) As an example, the most basic HDF5 file object, an empty file, is mapped to the following JSON string:

{  
 "apiVersion": "0.0.0",  
 "groups": {  
 "913d8791-7b51-11e4-89fd-3c15c2da029e": {}  
 },  
 "root": "913d8791-7b51-11e4-89fd-3c15c2da029e"  
}

The top-level JSON object in the above HDF5 File object is a collection of key-value pairs, and includes the key “root” which is mapped to a UUID for the root group. All groups contained in the file are listed in a collection mapped to the key “groups” in the top-level JSON object. Additional HDF5 objects in a file are subsequently mapped to the appropriate JSON types, with keys added to the appropriate collections. HDF5-JSON contains examples, in addition to the production rules.

It is notable that despite the ostensible interpretation of the ‘H’ in HDF5 as ‘hierarchical’, objects in the standard HDF5-JSON rendering are actually store in a flat format, i.e. rather than groups being arranged hierarchically, there is a key at the HDF5-JSON for ‘groups’ and all groups appear in the associated JSON list object.

Every HDF5 object which can be rendered as a JSON value, will fit nicely as a self-contained JSON object in any place that a JSON object can be used for a value, such as in a list or in an array. This allows for use of the API beyond the generation or translation of a complete HDF5 file *via* the JSON VOL.

1. **VOL Design:**
   1. **Persistent state and library object lifecycle**

In a manner similar to the REST VOL, the persistent state of all objects in the JSON VOL is maintained in a space isolated from the library and user space, and accessible only by API calls. In the REST VOL, authoritative state is maintained on the HSDS server. In the JSON VOL all state is to be maintained within a JANSSON object. The VOL layer maps public API calls to the JSON VOL layer. The JSON VOL layer manipulates the persistent state of the JANSSON object according to the HDF5-JSON production rules. The JANSSON object is able to encode and decode JSON.

In making calls to the JSON VOL, familiar HDF5 hid\_t ‘handles’ are generated, and they behave as expected in any HDF5 application. Within the VOL layer, these handles are associated with the VOL ‘library object’ type H5VL\_object\_t, the superclass for all persistent object types. The lifecycle of VOL objects is an extension of the hid\_t handle mechanism, and the lifecycle management (opening and closing) of these VOL library objects corresponds exactly to the lifecycle management of the hid\_t handles.

In general:

* A \_create\_ call will manipulate the underlying JANSSON representation of the object, as well as create or delete the ‘library object’ handles needed to access it.
* A \_delete\_ call will manipulate the underlying JANSSON representation of the object.
* An \_open\_ or \_close\_ call does not *manipulate* the underlying JANSSON. It will simply allocate or free the ‘library object’ resources needed to track and access the object.

Files:

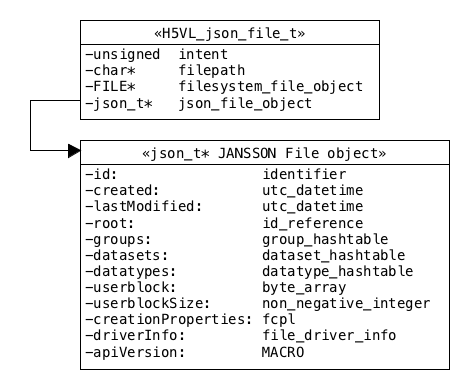
* Creating a *file* object causes the creation of a new top-level JANSSON object, as well as causing a resource to be allocated on the underlying storage.
* Opening a file causes an HDF5 JSON file on the underlying storage to be opened and read/decoded from JSON into a new top-level JANSSON object. The *H5VL\_json\_file\_open*() callback validates whether the file read represents a legitimate HDF5 JSON file.
* Closing a file object causes the top-level JANSSON object to be encoded to JSON and written on underlying storage.

The JSON VOL touches the filesystem only as a convenient means of persisting the data. All interactions with the filesystem occur within three calls: H5VL\_json\_file\_create(), H5VL\_json\_file\_open(), and H5VL\_json\_file\_close() calls, which create a new, empty file on the filesystem, read JSON from an existing file to generate the in-memory JANSSON representation, and write out JSON from the in-memory representation, respectively. The JSON parsed and generated to and from the in-memory representation could just as easily be written/read to/from a pipe, fifo, socket, etc. as to a file.

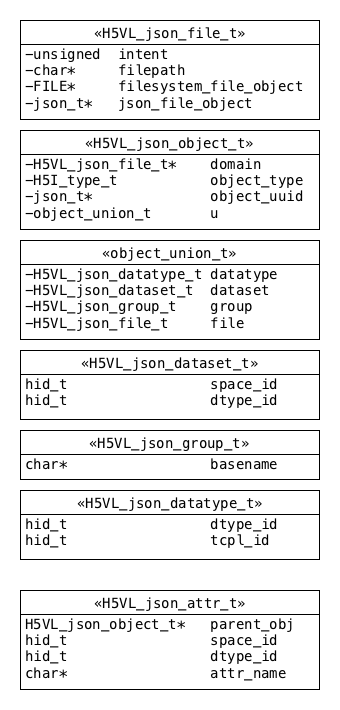
When a new VOL File object is created, two important things happen:

* A H5VL\_json\_file\_t object is created. This is the type which HDF5 VOL layer uses to keep track of any interactions with persistent storage. Since it is a well-understood model, a POSIX layer is assumed for convenience of IO.
* A json\_t\* (JANSSON object type) object is created, and a reference to it is stored in the above object. This object contains all of the *content* which will be persisted to a file. It is capable of generating, based on its state, the JSON representation, as defined by the HDF5-JSON production rules. Since H5VL\_json\_file\_create() generates a new file, this object would generate the corresponding ‘empty file’ representation.

The ‘empty file’ in-memory data structure (aka json\_file\_object) is initialized to contain the required elements. As data is added via the subordinate object types (Groups, Datasets, Attribute, Types), json\_file\_object is manipulated via calls to JANSSON. Subordinate object state is contained wholly in json\_file\_object and will not have state in any other location.



The JSON VOL maps a filesystem object to an in-memory representation. The json\_file\_object points to a JANNSON structure which is the authoritative source / database for everything to be persisted in the file. All of the members of this object are the keys declared in the HDF5-JSON production rules.



The JSON VOL library types. These types are the JSON VOL’s handles that allow it to connect to the persistent JANSSON representation. Note that the H5VL\_json\_object\_t serves as a wrapper for the file, dataset, group, and datatype types. This type contains an object UUID which is the key every persistent object uses to identify itself in persistent storage. For convenience, it also contains a pointer(domain) to the file object which contains it.

* 1. **HDF5-JSON mappings**

Files generated according to the HDF5-JSON production rules are readable by the JSON VOL. The rules specify that the file object is a JSON hashtable object and contains the following items: id, created, lastModified, root, groups, datasets, datatypes, userblock, userblockSize, creationProperties, driverInfo, apiVersion. In the JSON VOL, *H5Fopen()* maps to *H5VL\_json\_file\_open()* verifies that a minimum set of fields is included: **id, root, groups, apiVersion.**

* + 1. **id**

The id field maps to an ‘identifier’ value. The ‘identifier’ type is defined in the ‘Miscellaneous’ types section of the production rules. It contains a uuid. This uuid object can be represented in memory as type char[].

* + 1. **created**

The created field maps to a ‘utc\_datetime’ value. The exact format for this value is not specified (listed as \*\*TBD\*\* in spec) however it can be assumed to be representable in memory as a 64-bit unsigned integer and in JSON text as e.g., ‘Wed Oct 25 11:41:21 PHT 2017’

* + 1. **lastModified**

The lastModified field maps to a ‘utc\_datetime’ value. The exact format for this value is not specified (listed as \*\*TBD\*\* in spec) however it can be assumed to be representable in memory as a 64-bit unsigned integer and in JSON text as e.g., ‘Wed Oct 25 11:41:21 PHT 2017’

* + 1. **root**

The root field contains an ‘id\_reference’ which is an ‘identifier’ and which is the a that maps to the root group for the file object. This identifier is a uuid object and can be represented in memory as type char[].

* + 1. **groups**

The groups field contains a ‘group\_hashtable’ object. The JSON representation of the group\_hashtable type is a JSON hashtable object, containing zero or more ‘group’ objects. The in-memory representation of this ‘group\_hashtable’ object will be an instance of the JSON parser/render’s hashtable type.

* + 1. **datasets**

The datasets field contains a ‘dataset\_hashtable’ object. The JSON representation of the dataset\_hashtable type is a JSON hashtable object, containing zero or more ‘dataset’ objects. The in-memory representation of this ‘dataset\_hashtable’ object will be an instance of the JSON parser/render’s hashtable type.

* + 1. **datatypes**

The datatypes field contains a ‘datatypes\_hashtable’ object. The JSON representation of the datatypes\_hashtable type is a JSON hashtable object, containing zero or more ‘datatype’ objects. The in-memory representation of this ‘datatype\_hashtable’ object will be an instance of the JSON parser/render’s hashtable type.

* + 1. **userblock**

The userblock field contains a ‘byte\_array’ object. This ‘byte\_array’ object maps through the ‘byte\_list’ object, containing zero or more ‘byte\_value’ objects, which are ultimately represented as hexadecimals. It can be represented in memory as char\*.

* + 1. **userblockSize**

The userblockSize field contains a ‘non\_negative\_integer’ which is a power of two and is greater than or equal to 512. It can be represented in memory as an unsigned 64-bit integer.

* + 1. **creationProperties**

The creationProperties field contains a ‘fcpl’ object. This object can be represented as an instance of the JSON parser/render’s hashtable type.

* + 1. **driverInfo**

The driverInfo field contains a ‘file\_driver\_info’ object, which maps to either a ‘family\_driver\_info’ or ‘multi\_driver\_info’ object. A ‘family\_driver\_info’ object can be represented in memory as a struct with the field memberSize, a 64-bit unsigned integer. A ‘multi\_driver\_info’ object contains a ‘data\_distribution\_list’ object, which contains zero or more ‘data\_item’ objects. See production rules for details.

* + 1. **apiVersion**

The apiVersion field contains a simple string, e.g. “1.0.0” which can be represented in memory by a char\*.

The above mappings are drawn directly from the HDF5-JSON production rules. Some of the components defined by the production rules for these objects are specific to the HDF5 file format and have no intrinsic meaning above the VOL layer/outside of native VOL. For the remaining HDF5 object types (group, dataset, datatype, etc.) the reader is referred to the production rules.

* 1. **JSON VOL API callbacks**

The JSON VOL will contain the same set of callback functions as other VOLs, with their usual interpretations. Non-implemented/unnecessary functions are either NULL (prototype shown in strikethrough) or exist as stubs (prototype in gray) below.

## Public init and terminate functions

static herr\_t H5VL\_json\_init(void);

static herr\_t H5VL\_json\_term(hid\_t vtpl\_id);

## Attributes

static void \*H5VL\_json\_attr\_create(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*attr\_name, hid\_t acpl\_id, hid\_t aapl\_id, hid\_t dxpl\_id, void \*\*req);

static void \*H5VL\_json\_attr\_open(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*attr\_name, hid\_t aapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_attr\_read(void \*attr, hid\_t dtype\_id, void \*buf, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_attr\_write(void \*attr, hid\_t dtype\_id, const void \*buf, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_attr\_get(void \*obj, H5VL\_attr\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_attr\_specific(void \*obj, H5VL\_loc\_params\_t loc\_params, H5VL\_attr\_specific\_t specific\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

~~static void \*H5VL\_json\_attr\_optional();~~

static herr\_t H5VL\_json\_attr\_close(void \*attr, hid\_t dxpl\_id, void \*\*req);

## Datasets

static void \*H5VL\_json\_dataset\_create(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*name, hid\_t dcpl\_id, hid\_t dapl\_id, hid\_t dxpl\_id, void \*\*req);

static void \*H5VL\_json\_dataset\_open(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*name, hid\_t dapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_dataset\_read(void \*dset, hid\_t mem\_type\_id, hid\_t mem\_space\_id,

hid\_t file\_space\_id, hid\_t dxpl\_id, void \*buf, void \*\*req);

static herr\_t H5VL\_json\_dataset\_write(void \*dset, hid\_t mem\_type\_id, hid\_t mem\_space\_id,

hid\_t file\_space\_id, hid\_t dxpl\_id, const void \*buf, void \*\*req);

static herr\_t H5VL\_json\_dataset\_get(void \*dset, H5VL\_dataset\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_dataset\_specific(void \*dset, H5VL\_dataset\_specific\_t specific\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

~~static herr\_t H5VL\_json\_dataset\_optional();~~

static herr\_t H5VL\_json\_dataset\_close(void \*dset, hid\_t dxpl\_id, void \*\*req);

## Datatypes

static void \*H5VL\_json\_datatype\_commit(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*name, hid\_t type\_id, hid\_t lcpl\_id, hid\_t tcpl\_id, hid\_t tapl\_id, hid\_t dxpl\_id, void \*\*req);

static void \*H5VL\_json\_datatype\_open(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*name, hid\_t tapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_datatype\_get(void \*dt, H5VL\_datatype\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

~~static herr\_t H5VL\_json\_datatype\_specific(),~~

~~static herr\_t H5VL\_json\_datatype\_optional(),~~

static herr\_t H5VL\_json\_datatype\_close(void \*dt, hid\_t dxpl\_id, void \*\*req);

## Files

static void \*H5VL\_json\_file\_create(const char \*name, unsigned flags, hid\_t fcpl\_id, hid\_t fapl\_id, hid\_t dxpl\_id, void \*\*req);

static void \*H5VL\_json\_file\_open(const char \*name, unsigned flags, hid\_t fapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_file\_get(void \*file, H5VL\_file\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_file\_specific(void \*file, H5VL\_file\_specific\_t specific\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_file\_optional(void \*file, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_file\_close(void \*file, hid\_t dxpl\_id, void \*\*req);

## Groups

static void \*H5VL\_json\_group\_create(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*name, hid\_t gcpl\_id, hid\_t gapl\_id, hid\_t dxpl\_id, void \*\*req);

static void \*H5VL\_json\_group\_open(void \*obj, H5VL\_loc\_params\_t loc\_params, const char \*name, hid\_t gapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_group\_get(void \*obj, H5VL\_group\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

~~static herr\_t H5VL\_json\_group\_specific();~~

~~static herr\_t H5VL\_json\_group\_optional();~~

static herr\_t H5VL\_json\_group\_close(void \*grp, hid\_t dxpl\_id, void \*\*req);

## Links

static herr\_t H5VL\_json\_link\_create(H5VL\_link\_create\_type\_t create\_type, void \*obj,

H5VL\_loc\_params\_t loc\_params, hid\_t lcpl\_id, hid\_t lapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_link\_copy(void \*src\_obj, H5VL\_loc\_params\_t loc\_params1,

void \*dst\_obj, H5VL\_loc\_params\_t loc\_params2,

hid\_t lcpl\_id, hid\_t lapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_link\_move(void \*src\_obj, H5VL\_loc\_params\_t loc\_params1,

void \*dst\_obj, H5VL\_loc\_params\_t loc\_params2,

hid\_t lcpl\_id, hid\_t lapl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_link\_get(void \*obj, H5VL\_loc\_params\_t loc\_params, H5VL\_link\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_link\_specific(void \*obj, H5VL\_loc\_params\_t loc\_params, H5VL\_link\_specific\_t specific\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

~~static herr\_t H5VL\_json\_link\_optional();~~

## Objects

static void \*H5VL\_json\_object\_open(void \*obj, H5VL\_loc\_params\_t loc\_params, H5I\_type\_t \*opened\_type, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_object\_copy(void \*src\_obj, H5VL\_loc\_params\_t loc\_params1, const char \*src\_name, void \*dst\_obj, H5VL\_loc\_params\_t loc\_params2, const char \*dst\_name, hid\_t ocpypl\_id, hid\_t lcpl\_id, hid\_t dxpl\_id, void \*\*req);

static herr\_t H5VL\_json\_object\_get(void \*obj, H5VL\_loc\_params\_t loc\_params, H5VL\_object\_get\_t get\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_object\_specific(void \*obj, H5VL\_loc\_params\_t loc\_params, H5VL\_object\_specific\_t specific\_type, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

static herr\_t H5VL\_json\_object\_optional(void \*obj, hid\_t dxpl\_id, void \*\*req, va\_list arguments);

# Example program and output

* 1. This simple example program illustrates the use of the JSON VOL to generate a reversible encoding of a simple hdf5 file into JSON:

1 #include "hdf5.h"

2 #include <stdio.h>

3

4 void main(){

5

6 hid\_t fapl\_id = H5Pcreate(H5P\_FILE\_ACCESS);

7 H5VLjson\_init();

8 H5Pset\_fapl\_json\_vol(fapl\_id);

9

10 const char\* name = "new\_file.h5j";

11 hid\_t file\_id = H5Fcreate(name, H5F\_ACC\_TRUNC, H5P\_DEFAULT, fapl\_id );

12 hid\_t group\_id = H5Gcreate(file\_id, "/new\_group/inside\_group/deep\_group",

13 H5P\_DEFAULT, H5P\_DEFAULT, H5P\_DEFAULT);

14

15 H5Gclose(group\_id);

16 H5Fclose(file\_id);

17 H5Pclose(fapl\_id);

18 H5VLjson\_term();

19 }

* 1. The above example program was linked against a prototype version of the JSON VOL and was used to generate the following output:

{

"id": "164b4a11-fc92-d128-ea97-90a532cd1a63",

"root": "3f02d876-0650-67a4-7017-7e21bdfa65f7",

"userblock": [],

"userblockSize": 0,

"fcpl": null,

"created": "Mon Nov 20 1:52:45 UTC 2017",

"lastModified": "Mon Nov 20 1:52:45 UTC 2017",

"groups": {

"3f02d876-0650-67a4-7017-7e21bdfa65f7": {

"alias": [],

"attributes": [],

"links": [

{

"class": "H5L\_TYPE\_HARD",

"title": "new\_group",

"collection": "groups",

"id": "d773dd6b-b6cf-6a16-1c15-2efdde6cbb58"

}

],

"created": "Mon Nov 20 1:52:45 UTC 2017",

"lastModified": "Mon Nov 20 1:52:45 UTC 2017",

"creationProperties": {}

},

"d773dd6b-b6cf-6a16-1c15-2efdde6cbb58": {

"alias": [],

"attributes": [],

"links": [

{

"class": "H5L\_TYPE\_HARD",

"title": "inside\_group",

"collection": "groups",

"id": "4b1f42fb-a551-67cf-e25c-6c14f121d13f"

}

],

"created": "Mon Nov 20 1:52:45 UTC 2017",

"lastModified": "Mon Nov 20 1:52:45 UTC 2017",

"creationProperties": {}

},

"4b1f42fb-a551-67cf-e25c-6c14f121d13f": {

"alias": [],

"attributes": [],

"links": [

{

"class": "H5L\_TYPE\_HARD",

"title": "deep\_group",

"collection": "groups",

"id": "c6b5e603-e7e4-0a81-9c24-de54727c455a"

}

],

"created": "Mon Nov 20 1:52:45 UTC 2017",

"lastModified": "Mon Nov 20 1:52:45 UTC 2017",

"creationProperties": {}

},

"c6b5e603-e7e4-0a81-9c24-de54727c455a": {

"alias": [],

"attributes": [],

"links": [],

"created": "Mon Nov 20 1:52:45 UTC 2017",

"lastModified": "Mon Nov 20 1:52:45 UTC 2017",

"creationProperties": {}

}

},

"datasets": {},

"datatypes": {},

"driverInfo": {},

"apiVersion": "1.0.0"

}

* 1. Running the same example with the native VOL would generate the standard file format; the h5dump of it yields:

HDF5 "new\_file.h5j" {

GROUP "/" {

GROUP "new\_group" {

GROUP "inside\_group" {

GROUP "deep\_group" {

}

}

}

}

}

While it may at first appear more cumbersome than the h5dump format, JSON-formatted HDF5 content is also more standard, complete, and extensible.

# Testing and acceptance

# Early test cases should include obvious examples, such as the ‘empty’ HDF5-JSON file, as well as the other examples (“Classic”, Array Datatype, Compound Datatype, etc.) provided with HDF5-JSON documentation.

* 1. Reversibility: Since the rendering and parsing are inverse operations, it should theoretically be possible to apply these transforms ad nauseum without loss or modification of data. In practice, some whitespace may be lost or gained, and some items in lists could be re-ordered. Additionally, numerical precision for floating point values may be ambiguous and the values represented may be anticipated to drift slightly. Test cases which do not experience these effects can certainly be constructed.
  2. Code rendered via this VOL should be interpretable by the python HDF5-JSON code and vice-versa.

**<<<need more testing specifics>>>**

# Recommendations

1. JANSSON can be used for the encoding and decoding of JSON.
2. The HDF5-JSON production rules should be adhered to as they have already been vetted and provide a clear path to implementation.
3. Re-usability should be kept in mind, as there may be future work which seeks to interact with HDF5 data structures *via* JSON. The REST VOL is one such case.
4. **Acknowledgements**

Thanks to:

* Jordan Henderson for making initial endeavors into the JSON-ification of HDF5 structures in the C library *via* related work on REST VOL.
* Dana Robinson for proposing JSON VOL as an alternative bytes-on-disk format.
* Gerd Heber for generating an initial compiled implementation of HDF5-JSON translation in C++.
* Aleksandar Jelenak and John Readey for helpful discussions and implementation of HDF5-JSON in python.
* Dave Pearah for suggesting protocol buffers as a possible intermediate step.
* John Mainzer and Neil Fortner for helpful discussions regarding the in-memory representation.

Thanks to DOE for funding of work under the Exascale Computing Project (ECP). This work was [partially] supported by [grant, contract, cooperative agreement] number [agreement number] [from, with] [organization name (acronym)]. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author[s] and do not necessarily reflect the views of [organization].

1. **Revision History**

|  |  |
| --- | --- |
| *October 10, 2017:*  *November 21, 2017:* | Version 1 circulated for comment within The HDF Group.  Version 2 scope of work reduced to JSON VOL. Some implementation details via JANSSON and production rules included. |

1. **Glossary of Terminology**

|  |  |
| --- | --- |
| **production rules** | The specification that describes how HDF5 objects are to be expressed in JSON and vice-versa |
| **parsing** | reading of a (JSON) string and digesting it into tokens |
| **lexing** | Generation of (JSON) code from a structured object |
| **HDF5 object** | An HDF5 File, Group, Dataset, or Datatype |
| **library object** | The handle (allocated in memory) which allows access to the persistent representation of an HDF5 object. |
| **persistent representation** | The in-memory representation of an HDF5 File object and all of its contained objects *via* a JANSSON. |
| **underlying storage** | A convenience for loading and storing the state of a persistent representation |
| **encoding** | Rendering a set of HDF5 objects as a JSON string |
| **decoding** | Reading an encoded JSON string to generate HDF5 objects |

1. **References**
2. The HDF Group, HDF5-JSON mappings, http://hdf5-json.readthedocs.io/en/latest/index.html
3. D. Crockford, *et. al.*, JSON RFC https://tools.ietf.org/html/rfc4627.html
4. JSON.org <http://www.json.org/>
5. protocol buffers
6. Lloyd Hilaiel, *et. al.,* YAJL (Yet Another JSON library), https://github.com/lloyd/yajl
7. Petri Lehtinen *et al.*, JANSSON software, <https://jansson.readthedocs.io/en/latest/index.html>
8. M. Chaarawi, “User Guide for Developing a Virtual Object Layer Plugin,” 2014, available at: http://svn.hdfgroup.uiuc.edu/hdf5doc/trunk/RFCs/HDF5/VOL/user guide/
9. M. Chaarawi and Q. Koziol, “RFC: Virtual Object Layer,” 2014, available at <http://svn.hdfgroup.uiuc.edu/hdf5doc/trunk/RFCs/HDF5/VOL/RFC/>

1. RFC THG 2011-11-1.v1 [↑](#footnote-ref-0)
2. HDF5-JSON http://hdf5-json.readthedocs.io/en/latest/bnf/index.html [↑](#footnote-ref-1)
3. http://www.json.org/ [↑](#footnote-ref-2)
4. JANNSON [↑](#footnote-ref-3)
5. YAJL [↑](#footnote-ref-4)
6. https://developers.google.com/protocol-buffers/ [↑](#footnote-ref-5)
7. The HDF Group, HDF5-JSON mappings, http://hdf5-json.readthedocs.io/en/latest/index.html [↑](#footnote-ref-6)