RFC: A Simple VFD Configuration Language

John Mainzer john.mainzer@lifeboat.llc

As discussed in "RFC: A Plugin Interface for HDF5 Virtual File Drivers", a text based configuration language for VFDs would allow VFD agnostic configuration of VFD stacks via either environment variable or command line parameter.

This note on that RFC is essentially a simplified rewrite of Appendix A, combined with an outline of how the proposed configuration language would be parsed and employed.

---

1 Overview

Consider the following grammar:

```
<name_value_pair> ::= ‘(‘ <identifier> <value> ’)’

<value> ::= <integer> | <float> | <quote_string> | <binary_blob> | <name_value_pair_list>

<name_value_pair_list> ::= ‘(‘ (<name_value_pair>)* ‘)’
```

where the non-terminals not defined above are loosely defined below:

`<identifier>`	a valid C identifier.
`<integer>`	a C integer constant
`<float>`	a C floating point constant
`<quote_string>`	a C quote string
`<binary_blob>`	a hex representation of an arbitrary sequence of binary bytes

As the astute reader will note, this is just a restricted definition of a LISP form.

To turn this into a workable configuration language, we need to add some semantic constraints.

In the context of a VFD configuration, the top level must be a single `<name_value_pair>`, where the identifier is the name of the VFD, and the value must be a (possibly empty) `<name_value_pair_list>`[1]. This list of name value pairs must contain the configuration data required by the target VFD, with the list of identifiers that may appear, and the range and type of each associated value specified by the target VFD. As a result, the maximum length of this list of name value pairs is also specified by the target VFD – a point that shall become important shortly.

Leaving aside the matter of underlying VFDs, the values associated with each of these name-value pairs will be either integer, float, quote string, or binary blob. While this selection of types is sufficient for all VFDs that we are aware of, new types can be added as necessary.

For underlying VFDs, the value must be a single name value pair, where the name is the name of the underlying VFD, and the value is another (possibly empty) list of name value pairs containing the necessary configuration information.

While it should be obvious that this format allows definition of an arbitrary, possibly tree structured, VFD stacks, how strings in this language are parsed and used to setup the desired VFD stack requires some discussion.

First, observe that every name value pair can be described by the following structure:

```
typedef enum nv_val_t { int_const, float_const, qstr_const, bblob_const, nv_pair };
```

1 This is slightly different from the grammar and examples shown in appendices A and B of “RFC: A Plugin Interface for HDF5 Virtual File Drivers”. The list of name value pairs is enclosed in parentheses to simplify the grammar and parsing.

```
typedef struct nv_pair_t {
    char * name;
    nv_val_t val_type;
    int int_val;
    float float_val;
    void * vlen_val;
} nv_pair_t;
```

where:

- the name is stored in a C string pointed to by the `name` field,
- the type of the associated value is indicated by the `val_type` field, and
- the `int_val`, `float_val`, and `vlen_val` fields are used as follows depending on the `val_type` field:
  - `int_const`: Value of integer constant is stored in the `int_val` field.
  - `float_const`: Value of floating-point constant is stored in the `float_val` field.
  - `qstr_const`: Pointer to a buffer containing the C string representation of the quote string is stored in the `vlen_val` field.
  - `bblob_const`: Pointer to a buffer containing the decoded binary blob is stored in the `vlen_val` field, with the length of this buffer stored in the `int_val` field.
  - `nv_pair`: Pointer to a C string containing the text representation of a (possibly empty) list of name value pairs.

Second, observe that the above grammar is easily parsed with a recursive descent parser.

Recursive descent parsers have largely gone out of style today, as there are many parser generators available. However, they are easy to write and maintain for simple grammars such as the one presented above. Further, they lend themselves to performing operations as each production is satisfied – which is particularly convenient in this use case.

In the context of VFD stack configuration, the recursive descent parser will consist of two function – one to parse a `<name_value_pair>` and load its name and value into an instance of `nv_val_t`, and one to parse a `<name_value_pair_list>`, and load the names and values of its contents into an array of `nv_val_t`. The signatures of these functions will look something like:

```
bool parse_name_val_pair(char *conf_string, nv_val_t * nv_val_ptr);
bool parse_name_val_pair_list(char *conf_string,  nv_val_t nv_vals[],  int nv_count);
```

where:

`conf_string` is a pointer to the string to be parsed,

`nv_val_ptr` is a pointer to a single instance of `nv_val_t`, and

`nv_vals[]` is an array of `nv_val_t` of length `nv_count`

Both functions return `true` on success, and `false` if any error is detected. In particular, `parse_name_val_pair_list()` must fail if more than `nv_count` name value pairs are encountered. It will

be the responsibility of the caller to generate an error if required name value pairs are omitted, unknown name value pairs appear, or if the supplied values in any way invalid.

With these functions in hand, we can describe the use of the configuration string as follows:

The configuration string version of `H5FD_open()` calls `parse_name_val_pair()` to obtain the name of the specified VFD and a string containing the list of name value pairs required to configure it. It then loads the indicated VFD if necessary, and calls its open routine with a pointer to the string containing the list name value pairs.

The open routine of the target VFD, calls `parse_name_val_pair_list()` to load its configuration data into an array of `nv_val_t`. As mentioned above, the names, types, and ranges of the fields used for the VFDs configuration are specified by the target VFD. This data is then used to configure the VFD. If there is an underlying VFD, a string containing a name value pair specifying it must have been loaded into an appropriately named instance of `nv_val_t`. This string is passed to the configuration string version of `H5FD_open()`, and the process repeats until the VFD stack is completely configured.

This approach of allows the definition and use of new plugin VFDs without any change to the VFD layer in HDF5 once the necessary utilities are implemented, since the exact details of the configuration string for any such VFD can be defined by it as long as it conforms to the above grammar.

For clarity, the following example configuration string for the encryption VFD is offered. Note that the key used in this example is not an actual key – although the binary blob that expresses it matches the key size. The formatting is for readability, and not required by the configuration language.

2 Example

Configuration string for Encryption VFD that sets up parameters for Pge Buffer VFD, Encryption VFD and default terminal VFD SEC2.

```
( page_buffer
  ( ( page_size 4096 )
    ( max_num_pages 16 )
    ( replacement_policy 0 )
    ( underlying_VFD
       ( encryption_VFD
         ( ( plaintext_page_size  4096 )
            ( ciphertext_page_size 4112 )
            ( encryption_buffer_size 65792 )
            ( cipher  0 )
            ( cipher_block_size 16 )
            ( key_size  32 )
            ( key
              0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
            )
            ( iv_size 16 )
```

```
            ( mode 0 )
            ( underlying_VFD ( sec2 () ) )
          )
        )
      )
    )
  )
)
```

Observe that the proposed grammar is sufficiently general that it should be applicable to other applications in the HDF5 library, albeit with different semantic constraints.

Acknowledgement

This work is supported by the U.S. Department of Energy, Office of Science under award number DE-SC0024823 for Phase I SBIR project “Protecting the confidentiality and integrity of data stored in HDF5"

Revision History

October 17, 2024:	Version 1 was created from original document from 9/19/2024.