LA-UR-22-28306

Approved for public release; distribution is unlimited.

Title:	LANL Nuclear Data Manager Format Specification v1.0
Author(s):	Josey, Colin James Conlin, Jeremy Lloyd
Intended for:	Report
Issued:	2022-08-09

Los Alamos
NATIONAL LABORATORY

LANL Nuclear Data Manager Format Specification v1.0

Colin Josey, cjosey@lanl.gov, XCP-3,
Jeremy Lloyd Conlin, XCP-5
Los Alamos National Laboratory

August 8, 2022

1 Introduction

The LANL nuclear data manager tool is a Python utility that downloads nuclear data from the Nuclear Data Team's website at https://nucleardata.lanl.gov, arranges it, and configures directory listings for nuclear codes such as the MCNP code to consume. In this document, the Version 1.0 API that this tool relies on will be discussed, which will show how an external provider of nuclear data could set up their website to provide a compatible implementation.

2 The API

There are three main components in the API. First, there is a `libraries.json` file that provides a directory listing of libraries available on the server. Second, each library file has a particular layout. Finally, there is a `code_support.json` format that can be used to generate directory listings of a particular format for other codes.

Within this document, JSON files [1] will be shown. Text shown within C-style inline comments (`/* */`) are comments. Within these comments, the format of the data will be described, as well as a note on whether or not the object is a required element. In all JSON files, the dictionary entry `comment` is always reserved for annotation purposes, and the nuclear data tool will never process such a token. Further, all dates follow ISO 8601 [2].

2.1 `libraries.json`

The outermost format is described by Listing 1. There is a header with a `libraries.json` format version, a date of publication, and an optional root object to point users at where more information can be found. The `ACE Libraries` dictionary then contains each library available on the website, by name. Note that while the token specifies ACE libraries, other files can be supported, as will be discussed in Section 2.2. It is important to note that library name entries must be unique keys after converted to lower-case. As a result, two `libraries.json` cannot describe a `foo` library at the same time, nor can there be a `foo` and a `Foo`.

The next format component is the library description block, as shown in Listing 2. The header includes a description, the date, whether or not this library counts as production for the command line argument `--all production` in the data downloader, and optionally a URL. The date is used to sort the libraries in the default sorting. Newer libraries will be placed before older libraries in the default directory listing generation order, under the assumption that newer libraries were only released due to being superior.

Within the `files` entry, there are one or more file descriptors. The data downloader will only download one file for each library, so the contents of each compressed archive must be complete. Multiple files are allowed to allow a client to select a preferred compression format. Within a file

Listing 1: Libraries JSON Outer Format

```
{
  "version": "/* Semantic version, currently 1.0.0, required */",
  "date": "/* Date updated, YYYY-MM-DD, required */",
  "root": "/* URL to website, optional */",
  "ACE Libraries": {
    "/* Library name 1 */": /* ACE Library Desc., See Listing 2 */,
    "/* Library name 2 */": /* ACE Library Desc., See Listing 2 */,
    /* More Libraries ... */
  }
}
```

Listing 2: ACE Library Descriptor

```
{
  "description": "/* Short description of library, required */",
  "production": /* bool, library suitable in production?, required */,
  "url": "/* URL to website for library, optional */",
  "date": "/* Date produced, YYYY-MM-DD, required */",
  "files": [
    {
      "name": "/* filename for user systems, required */",
      "path": "/* URL to file, required */",
      "sha512": "/* SHA512 sum of file, required */"
    },
    /* More file options ... */
  ],
  "errata": [
    /* Errata Block, See Listing 3, optional */,
    /* More errata ... */
  ]
}
```

Listing 3: Library Errata Descriptor

```
{
  "description": "/* Short description of errata, required */",
  "date": "/* Date produced, YYYY-MM-DD, required */",
  "name": "/* Errata name, follows library name rules, required */",
  "files": [
    {
      "name": "/* filename for user systems, required */",
      "path": "/* URL to file, required */",
      "sha512": "/* SHA512 sum of file, required */"
    },
    /* More file options ... */
  ]
}
```

descriptor, the `path` is an absolute URL pointing to where the file exists on the server. The `name` is what the file should be saved as on the user's computer. The reason `name` is not extracted from the `path` is that there are some circumstances in which a user-friendly filename cannot be used in the URL. The SHA512 sum is used to ensure the downloaded file was received correctly. If the SHA512 does not match, the data downloader will treat it as a failed download and not install the library. The content of the file in `path` must match the rules discussed in Section 2.2.

The `errata` section contains zero or more errata blocks, and is optional. These blocks, shown in Listing 3, contain additional files that may be needed to introduce corrections for any reason. Note that the no-file-collision rule in Section 2.2 still applies to errata, so an errata cannot replace a file in the original library, only append. See 3.2 for a discussion for how errata sets are used.

2.2 Library File Layout

For practical reasons, the actual library files need to follow a few rules. First, each file must be an archive. At this time, the supported file formats are, in descending default priority:

1. `.tar.xz` or `.txz`
2. `.tar.bz2` or `.tbz2`
3. `.tar.gz` or `.tgz`
4. `.zip`

This priority was chosen due to compression ratios in normal use, and is constrained by the formats supported within the `tarfile` and `zipfile` packages in Python[1]. When extracted, there must be only one base folder. It is recommended but not required that this folder's name matches the library `name` entry.

The following rules apply to the files within that folder:

1. Regular files not listed below will be copied with the same heirarchy as in the base folder. So `[base folder]/foo/bar` will be installed at `[install folder]/foo/bar`.

[1]The low placement of `.zip` is due to its default DEFLATE algorithm. While LZMA-compressed `.zip` files should perform similarly to `.tar.xz`, this type of compression is rare and not supported by Microsoft Windows' built-in zip support, which is the primary target for `.zip` files. The absence of `.tar.zst`, which would be at the top of the list, is due to a lack of support in current implementations of Python's `tarfile`.

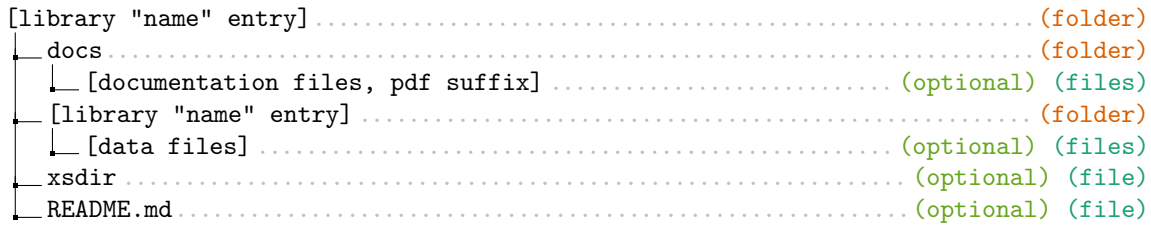

Figure 1: Recommended Library Layout

2. During installation, the process will roll back if a file is already installed at the location the tool wishes to copy to. This prevents two libraries from generating the same file with different contents and putting things in an inconsistent state.

3. PDF files are assumed to not be data, and are exempt from name collision checking. This allows a set of libraries to share a single reference document. It is still recommended to give the files a descriptive name, such as an LA-UR number for Los Alamos publications.

4. Any file named `xsdir` will be installed to `[install folder]/database/xsdir_[library ‘‘name’’]` and so only one file by that name can be present in the archive. This file is assumed to be an XSDIR fragment, containing only directory entries, and is used to create the full XSDIR. See Section 3.1 for more details.

5. Any file named `xsdir_2.0` will be installed to `[install folder]/database/xsdir_2.0_[library ‘‘name’’]` and so only one file by that name can be present in the archive.

6. Any file named `README.md` will be installed to `[install folder]/docs/README_[library ‘‘name’’].md` and so only one file by that name can be present in the archive.

7. Any file prefixed with a period (such as `.DS_Store`) will be ignored and not processed. This prevents accidental inclusion of hidden files from causing problems.

With these rules, the layout in Figure 1 is recommended. The only files in the root directory are those that are handled in a special way. Documentation is recommended to be placed in the `docs` folder for consistency with currently published libraries. As the library `name` entry must be unique, placing all other files in a folder using that name should minimize the risk of any further collisions.

2.3 `code_support.json`

The final file in the API is the code support file. This file contains information on how a given code would like directory files (such as XSDIR) to be formatted, as well as nuclear data that is fundamentally not supported. The layout is shown in Listing 4.

The directory name is the name of the file that will be generated. An example would be `xsdir` and `xsdir_mcnp6.3`. At least one is required in each support file, but as many as one wants can be made. The code will always generate an `xsdir_all` file, that contains the verbatim contents of the XSDIR fragments without any removals. The `format` entry controls the resulting format of the directory listing. The `version` entry is the version of said format to generate. At this time, `format` must be `xsdir` and `version` must be `1.0`. The `exclusion list` is an optional entry that allows a code developer to list off libraries that are known to be problematic for the code consuming this directory file. As an example of an exclusion list, see Listing 5. In that example, `1001.80c-1001.87c` and `1002.80c-1002.87c` are excluded from the `foo` library, and all ZAIDs are excluded from the `bar` library. Excluded lines will be entirely removed from the XSDIR file for compatibility reasons. A user can copy lines from the `xsdir_all` file if adjustments need to be made.

Listing 4: Code Support Descriptor

```
{
  "version": "/* Semantic version, currently 1.0.0, required */",
  "date": "/* Date updated, YYYY-MM-DD, required */",
  "directories": {
    "/* Directory name */": {
      "format": "/* Directory format string, required */",
      "version": "/* Directory format version, required */",
      "comment": "/* Description of file purpose, optional */",
      "exclusion list": {
        "/* library name, matches Listing 1 */": [
          {
            "identifier": "/* Python regex for ZAIDs, required */",
            "reason": "/* Rationale for why, required */",
            "reference": "/* Reference for more info, optional */"
          },
          /* more exclusion matches, optional */
        ],
        /* more libraries to exclude files from, optional */
      }
    },
    /* More directory files, optional */
  }
}
```

Listing 5: Exclusion List Example

```
"exclusion list": {
  "foo": [
    {
      "identifier": "1001.8[0-7]c",
      "reason": "Some short reason string.",
      "reference": "[URL]"
    },
    {
      "identifier": "1002.8[0-7]c",
      "reason": "Some short reason string.",
      "reference": "[URL]"
    }
  ],
  "bar": [
    {
      "identifier": ".*",
      "reason": "Some short reason string.",
      "reference": "[URL]"
    }
  ]
}
```

Note that the restrictions each code has on the XSDIR also apply to the reason block. For example, MCNP version 5 cannot have any lines in an XSDIR exceed 80 characters, even if they are comments. For this reason, it is recommended that the `reason` and `identifier` entries do not exceed 65 characters for any information intended for this version of the code. This is also the reason lines from excluded nuclides are removed instead of being commented out.

3 Other Details

The following two parts note how some of the internals of the data manager tool work, to explain some of the quirks of the API.

3.1 Handling XSDIRs

When a user requests an `XSDIR` to be generated, the process is as follows:

1. The contents of the file `awr_data`, included with the code, is added first.
2. For each library the user requests to add:
   (a) Loop through the errata list and add the XSDIR fragment of the errata in order of the `errata` list in Listing 2, if available (after removing excluded identifiers).
   (b) Append the XSDIR fragment of the library, if available (after removing excluded identifiers).
3. Write out to the file given by the directory name in Listing 4.

These XSDIR fragments are read from `[install folder]/database/xsdir_[library ‘‘name’’]` and thus correspond to any `xsdir` file within the library package. These fragments contain the `directory` listing of the `XSDIR` without the initial `directory` line. Extra whitespace at the end of the XSDIR fragment is removed.

3.2 Errata

It is assumed that a user will want to have errata corrections installed for any library they request. As a result, if a user installs a library that has an errata, it will go ahead and install the errata as well. The code will then treat the combination library as a single component under the original library's name. In addition, if a user runs the update command and new errata is available, the tool will encourage the user to install the errata immediately. If you do not wish for this level of interconnectivity, leaving corrections as a separate library is an option.

4 Summary

It is hoped that, by making this API available, other nuclear data providers consider adding support on their websites for the LANL nuclear data manager. In addition, there are several places designed for future expansion, such as the directory listing format, that can be expanded for use by other codes. If you end up using this API, please let us know how it goes, as well as provide any feedback on future enhancements that would be desired.

References

[1] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[2] Date and time – Representations for information interchange – Part 1: Basic rules. ISO 8601-1:2019, International Organization for Standardization, Geneva, CH, February 2019.