nexus

Release v2022.06

NIAC, https://www.nexusformat.org

Jun 24, 2022

CONTENTS

https://www.nexusformat.org/

CHAPTER

ONE

NEXUS: USER MANUAL

1.1 NeXus Introduction

NeXus[1] is an effort by an international group of scientists *motivated* to define a common data exchange format for neutron, X-ray, and muon experiments. NeXus is built on top of the scientific data format HDF5 and adds domain-specific rules for organizing data within HDF5 files in addition to a dictionary of well-defined domain-specific field names. The NeXus data format has three purposes:

1. *raw data*: NeXus defines a format that can serve as a container for all relevant data associated with a scientific instrument or beamline. This is a very important use case. This includes the case of streaming data acquisition, where time stamped data are logged.
2. *processed data*: NeXus also defines standards for processed data. This is data which has underwent some form of data reduction or data analysis. NeXus allows storing the results of such processing together with documentation about how the processed data was generated.
3. *standards*: NeXus defines standards in the form of *application definitions* for the exchange of data between applications. NeXus provides standards for both raw and processed data.

A community of scientists and computer programmers working in neutron and synchrotron facilities around the world came to the conclusion that a common data format would fulfill a valuable function in the scattering community. As instrumentation becomes more complex and data visualization becomes more challenging, individual scientists, or even institutions, find it difficult to keep up with new developments. A common data format makes it easier, both to exchange experimental results and to exchange ideas about how to analyze them. It promotes greater cooperation in software development and stimulates the design of more sophisticated visualization tools. Additional background information is given in the chapter titled *Brief history of NeXus*.

This section is designed to give a brief introduction to NeXus, the data format and tools that have been developed in response to these needs. It explains what a modern data format such as NeXus is and how to write simple programs to read and write NeXus files.

The programmers who produce intermediate files for storing analyzed data should agree on simple interchange rules.

[1] *J. Appl. Cryst.* (2015). **48**, 301-305 (https://doi.org/10.1107/S1600576714027575)

1.1.1 What is NeXus?

The NeXus data format has four components:

A set of design principles
to help people understand what is in the data files.

A set of data storage objects
(base.class.definitions and application.definitions) to allow the development of portable analysis software.

A set of subroutines
(*Utilities* and *examples*) to make it easy to read and write NeXus data files.

A Scientific Community
to provide the scientific data, advice, and continued involvement with the NeXus standard. NeXus provides a forum for the scientific community to exchange ideas in data storage.

In addition, NeXus relies on a set of low-level file formats to actually store NeXus files on physical media. Each of these components are described in more detail in the *Physical File format* section.

The NeXus Application-Programmer Interface (NAPI), which provides the set of subroutines for reading and writing NeXus data files, is described briefly in *NAPI: The NeXus Application Programming Interface*. (Further details are provided in the *NAPI* chapter.)

The principles guiding the design and implementation of the NeXus standard are described in the *NeXus Design* chapter.

Base classes, which comprise the data storage objects used in NeXus data files, are detailed in the base.class.definitions chapter.

Additionally, a brief list describing the set of NeXus Utilities available to browse, validate, translate, and visualise NeXus data files is provided in the *NeXus Utilities* chapter.

A Set of Design Principles

NeXus data files contain four types of entity: groups, fields, attributes, and links.

Groups
Groups are like folders that can contain a number of fields and/or other groups.

Fields
Fields can be scalar values or multidimensional arrays of a variety of sizes (1-byte, 2-byte, 4-byte, 8-byte) and types (characters, integers, floats). Fields are represented as HDF5 *datasets*.

Attributes
Extra information required to describe a particular group or field, such as the data units, can be stored as a data attribute. Attributes can also be given at the file level of an HDF5 file.

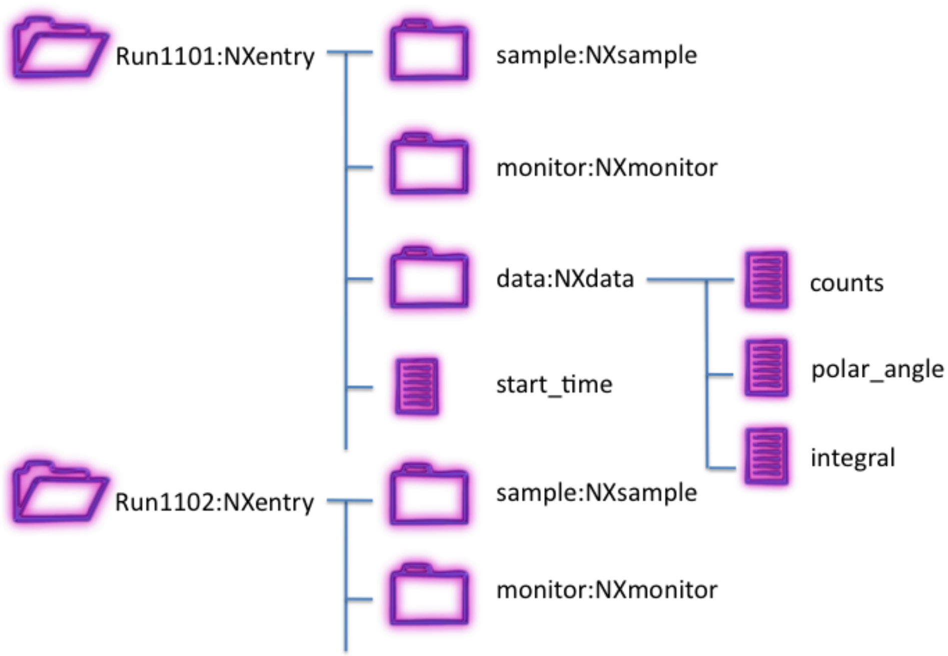
Links
Links are used to represent the same information in different places.

In fact, a NeXus file can be viewed as a computer file system. Just as files are stored in folders (or subdirectories) to make them easy to locate, so NeXus fields are stored in groups. The group hierarchy is designed to make it easy to navigate a NeXus file.

Example of a NeXus File

The following diagram shows an example of a NeXus data file represented as a tree structure.

Example of a NeXus Data File



Note that each field is identified by a name, such as `counts`, but each group is identified both by a name and, after a colon as a delimiter, the class type, e.g., `monitor:NXmonitor`). The class types, which all begin with `NX`, define the sort of fields that the group should contain, in this case, counts from a beamline monitor. The hierarchical design, with data items nested in groups, makes it easy to identify information if you are browsing through a file.

Important Classes

Here are some of the important classes found in nearly all NeXus files. A complete list can be found in the *NeXus Base Classes* chapter. A complete list of *all* NeXus classes may be found in the *NeXus Class Definitions* chapter.

Note: `NXentry` is the only class required in a valid NeXus data file.

NXentry
: *Required:* The top level of any NeXus file contains one or more groups with the class `NXentry`. These contain all the data that is required to describe an experimental run or scan. Each `NXentry` typically contains a number of groups describing sample information (class `NXsample`), instrument details (class `NXinstrument`), and monitor counts (class `NXmonitor`).

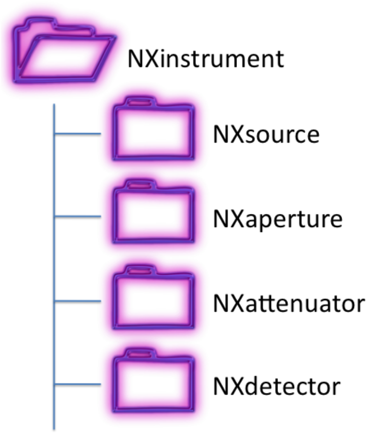
NXdata
: Each `NXentry` group may contain one or more `NXdata` groups. These groups contain the experimental results

in a self-contained way, i.e., it should be possible to generate a sensible plot of the data from the information contained in each NXdata group. That means it should contain the axis labels and titles as well as the data.

NXsample
A NXentry group will often contain a group with class NXsample. This group contains information pertaining to the sample, such as its chemical composition, mass, and environment variables (temperature, pressure, magnetic field, etc.).

NXinstrument
There might also be a group with class NXinstrument. This is designed to encapsulate all the instrumental information that might be relevant to a measurement, such as flight paths, collimation, chopper frequencies, etc.

NXinstrument excerpt



Since an instrument can include several beamline components each defined by several parameters, the components are each specified by a separate group. This hides the complexity from generic file browsers, but makes the information available in an intuitively obvious way if it is required.

Simple Example

NeXus data files do not need to be complicated. In fact, the following diagram shows an extremely simple NeXus file (in fact, the simple example shows the minimum information necessary for a NeXus data file) that could be used to transfer data between programs. (Later in this section, we show how to write and read this simple example.)

Example structure of a simple data file

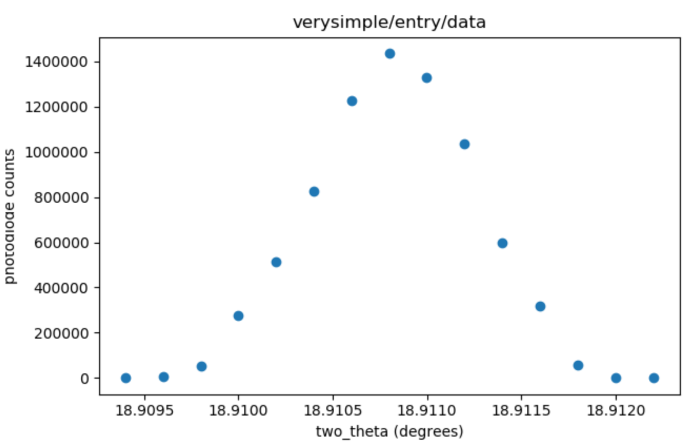
This illustrates the fact that the structure of NeXus files is extremely flexible. It can accommodate very complex instrumental information, if required, but it can also be used to store very simple data sets. Here is the structure of a very simple NeXus data file (`examples/verysimple.nx5`):

Structure of a very simple NeXus Data file

```
verysimple.nx5 : NeXus data file
  @default = "entry"
  entry:NXentry
    @NX_class = NXentry
    @default = "data"
    data:NXdata
      @NX_class = NXdata
      @signal = "counts"
      @axes = "two_theta"
      @two_theta_indices = [0]
      counts:int32[15] = [1193, 4474, 53220, '...', 1000]
        @units = "counts"
        @long_name = photodiode counts
      two_theta:float64[15] = [18.9094, 18.9096, '...', 18.9122]
        @units = "degrees"
        @long_name = "two_theta (degrees)"
```

NeXus files are easy to visualize. Here, this data is plotted using *NeXPy* simply by opening the NeXus data file and double-clicking the file name in the list:

Plot of a very simple NeXus HDF5 Data file



NeXus files are easy to create. This example NeXus file was created using a short Python program and the *h5py* package:

Using Python to write a very simple NeXus HDF5 Data file

```python
#!/usr/bin/env python
'''uses h5py to build the verysimple.nx5 data file'''

import h5py

angle = [18.9094, 18.9096, 18.9098, 18.91,  18.9102,
         18.9104, 18.9106, 18.9108, 18.911, 18.9112,
         18.9114, 18.9116, 18.9118, 18.912, 18.9122]
diode = [1193, 4474, 53220, 274310, 515430, 827880,
         1227100, 1434640, 1330280, 1037070, 598720,
         316460, 56677, 1000, 1000]

with h5py.File('verysimple.nx5', 'w') as f:
    f.attrs['default'] = 'entry'

    nxentry = f.create_group('entry')
    nxentry.attrs["NX_class"] = 'NXentry'
    nxentry.attrs['default'] = 'data'

    nxdata = nxentry.create_group('data')
    nxdata.attrs["NX_class"] = 'NXdata'
    nxdata.attrs['signal'] = 'counts'
    nxdata.attrs['axes'] = 'two_theta'
    nxdata.attrs['two_theta_indices'] = [0,]

    tth = nxdata.create_dataset('two_theta', data=angle)
    tth.attrs['units'] = 'degrees'
    tth.attrs['long_name'] = 'two_theta (degrees)'

    counts = nxdata.create_dataset('counts', data=diode)
    counts.attrs['units'] = 'counts'
    counts.attrs['long_name'] = 'photodiode counts'
```

A Set of Data Storage Objects

If the design principles are followed, it will be easy for anyone browsing a NeXus file to understand what it contains, without any prior information. However, if you are writing specialized visualization or analysis software, you will need to know precisely what specific information is contained in advance. For that reason, NeXus provides a way of defining the format for particular instrument types, such as time-of-flight small angle neutron scattering. This requires some agreement by the relevant communities, but enables the development of much more portable software.

The set of data storage objects is divided into three parts: base classes, application definitions, and contributed definitions. The base classes represent a set of components that define the dictionary of all possible terms to be used with that component. The application definitions specify the minimum required information to satisfy a particular scientific or data analysis software interest. The contributed definitions have been submitted by the scientific community for incubation before they are adopted by the NIAC or for availability to the community.

These instrument definitions are formalized as XML files, using *NXDL*, to specify the names of fields, and other NeXus data objects. The following is an example of such a file for the simple NeXus file shown above.

A very simple NeXus Definition Language (NXDL) file

```xml
<?xml version="1.0" ?>
<definition
  xmlns="http://definition.nexusformat.org/nxdl/3.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://definition.nexusformat.org/nxdl/3.1 ../nxdl.xsd"
  category="base"
  name="verysimple"
  version="1.0"
  type="group" extends="NXobject">

  <doc>
    A very simple NeXus NXDL file
  </doc>
  <group type="NXentry">
    <group type="NXdata">
      <field name="counts" type="NX_INT"  units="NX_UNITLESS">
        <doc>counts recorded by detector</doc>
      </field>
      <field name="two_theta" type="NX_FLOAT" units="NX_ANGLE">
        <doc>rotation angle of detector arm</doc>
      </field>
    </group>
  </group>
</definition>
```

Complete examples of reading and writing NeXus data files are provided *later*. This chapter has several examples of writing and reading NeXus data files. If you want to define the format of a particular type of NeXus file for your own use, e.g. as the standard output from a program, you are encouraged to *publish* the format using this XML format. An example of how to do this is shown in the *Creating a NXDL Specification* section.

A Set of Subroutines

NeXus data files are high-level so the user only needs to know how the data are referenced in the file but does not need to be concerned where the data are stored in the file. Thus, the data are most easily accessed using a subroutine library tuned to the specifics of the data format.

In the past, a data format was defined by a document describing the precise location of every item in the data file, either as row and column numbers in an ASCII file, or as record and byte numbers in a binary file. It is the job of the subroutine library to retrieve the data. This subroutine library is commonly called an application-programmer interface or API.

For example, in NeXus, a program to read in the wavelength of an experiment would contain lines similar to the following:

Simple example of reading data using the NeXus API

```
NXopendata (fileID, "wavelength");
NXgetdata (fileID, lambda);
NXclosedata (fileID);
```

In this example, the program requests the value of the data that has the label `wavelength`, storing the result in the variable lambda. `fileID` is a file identifier that is provided by NeXus when the file is opened.

We shall provide a more complete example when we have discussed the contents of the NeXus files.

Scientific Community

NeXus began as a group of scientists with the goal of defining a common data storage format to exchange experimental results and to exchange ideas about how to analyze them.

The *NeXus Community* provides the scientific data, advice, and continued involvement with the NeXus standard. NeXus provides a forum for the scientific community to exchange ideas in data storage.

The NeXus International Advisory Committee (NIAC) supervises the development and maintenance of the NeXus common data format for neutron, X-ray, and muon science through the NeXus class definitions and oversees the maintenance of the NeXus Application Programmer Interface (NAPI) as well as the technical infrastructure.

Representation of data examples

Most of the examples of data files have been written in a format intended to show the structure of the file rather than the data content. In some cases, where it is useful, some of the data is shown. Consider this prototype example:

example of NeXus data file structure

```
entry:NXentry
  instrument:NXinstrument
    detector:NXdetector
      data:[]
        @long_name = "strip detector 1-D array"
      bins:[0, 1, 2, ... 1023]
        @long_name = "bin index numbers"
  sample:NXsample
    name = "zeolite"
  data:NXdata
    @signal = "data"
    @axes = ["bins", "bins"]
    @bins_indices = [0, 1]
    data --> /entry/instrument/detector/data
    bins --> /entry/instrument/detector/bins
```

Some words on the notation:

- Hierarchy is represented by indentation. Objects on the same indentation level are in the same group
- The combination `name:NXclass` denotes a NeXus group with name `name` and class `NXclass`.
- A simple name (no following class) denotes a field. An equal sign is used to show the value, where this is important to the example.

- Sometimes, a data type is specified and possibly a set of dimensions. For example, `energy:NX_NUMBER[NE]` says *energy* is a 1-D array of numbers (either integer or floating point) of length `NE`.
- Attributes are noted as @name="value" pairs. The `@` symbol only indicates this is an attribute and is not part of the attribute name.
- Links are shown with a text arrow `-->` indicating the source of the link (using HDF5 notation listing the sequence of *names*).

Line 1 shows that there is one group at the root level of the file named `entry`. This group is of type `NXentry` which means it conforms to the specification of the `NXentry` NeXus base class. Using the HDF5 nomenclature, we would refer to this as the `/entry` group.

Lines 2, 8, and 10: The `/entry` group contains three subgroups: `instrument`, `sample`, and `data`. These groups are of type `NXinstrument`, `NXsample`, and `NXdata`, respectively.

Line 4: The data of this example is stored in the `/entry/instrument/detector` group in the dataset called `data` (HDF5 path is `/entry/instrument/detector/data`). The indication of `data:\[]` says that `data` is an array of unspecified dimension(s).

Line 5: There is one attribute of `/entry/instrument/detector/data`: `long_name`. This attribute *might* be used by a plotting program as the axis title.

Line 6 (reading `bins:\[0, 1, 2, ... 1023]`) shows that `bins` is a 1-D array of length presumably 1024. A small, representative selection of values are shown.

Line 7: an attribute that shows a descriptive name of `/entry/instrument/detector/bins`. This attribute might be used by a NeXus client while plotting the data.

Line 9 (reading `name = "zeolite"`) shows how a string value is represented.

Line 11 says that the default data to be plotted is called `data`.

Line 12 says that each axis *dimension scale* of `data` is described by the field called `bins`.

Line 13 says that `bins` will be used for axis 0 and axis 1 of `data`.

Lines 14-15: The `/entry/data`) group has two datasets that are actually linked as shown to data sets in a different group. (As you will see later, the `NXdata` group enables NeXus clients to easily determine what to offer for display on a default plot.)

Class path specification

In some places in this documentation, a path may be shown using the class types rather than names. For example:

```
/NXentry/NXinstrument/NXcrystal/wavelength
```

identifies a dataset called `wavelength` that is inside a group of type `NXcrystal` ...

As it turns out, this syntax is the syntax used in NXDL link specifications. This syntax is also used when the exact name of each group is either unimportant or not specified.

If default names are taken for each class, then the above class path is expressed as this equivalent HDF5 path:

```
/entry/instrument/crystal/wavelength
```

In some places in this documentation, where clarity is needed to specify both the path and class name, you may find this equivalent path:

```
/entry:NXentry/instrument:NXinstrument/crystal:NXcrystal/wavelength
```

Motivations for the NeXus standard in the Scientific Community

By the early 1990s, several groups of scientists in the fields of neutron and X-ray science had recognized a common and troublesome pattern in the data acquired at various scientific instruments and user facilities. Each of these instruments and facilities had a locally defined format for recording experimental data. With lots of different formats, much of the scientists' time was being wasted in the task of writing import readers for processing and analysis programs. As is common, the exact information to be documented from each instrument in a data file evolves, such as the implementation of new high-throughput detectors. Many of these formats lacked the generality to extend to the new data to be stored, thus another new format was devised. In such environments, the documentation of each generation of data format is often lacking.

Three parallel developments have led to NeXus:

1. *June 1994*: Mark Könnecke (Paul Scherer Institute, Switzerland) made a proposal using netCDF for the European neutron scattering community while working at the ISIS pulsed neutron facility.
2. *August 1994*: Jon Tischler and Mitch Nelson (Oak Ridge National Laboratory, USA) proposed an HDF-based format as a standard for data storage at the Advanced Photon Source (Argonne National Laboratory, USA).
3. *October 1996*: Przemek Klosowski (National Institute of Standards and Technology, USA) produced a first draft of the NeXus proposal drawing on ideas from both sources.

These scientists proposed methods to store data using a self-describing, extensible format that was already in broad use in other scientific disciplines. Their proposals formed the basis for the current design of the NeXus standard which was developed across three workshops organized by Ray Osborn (ANL), *SoftNeSS'94* (Argonne Oct. 1994), *SoftNeSS'95* (NIST Sept. 1995), and *SoftNeSS'96* (Argonne Oct. 1996), attended by representatives of a range of neutron and X-ray facilities. The NeXus API was released in late 1997. Basic motivations for this standard were:

1. *Simple plotting*
2. *Unified format for reduction and analysis*
3. *Defined dictionary of terms*

Simple plotting

An important motivation for the design of NeXus was to simplify the creation of a default plot view. While the best representation of a set of observations will vary depending on various conditions, a good suggestion is often known *a priori*. This suggestion is described in the NXdata group so that any program that is used to browse NeXus data files can provide a *best representation* without request for user input. A description of how simple plotting is facilitated in NeXus is shown in the section titled *Find the plottable data*.

NeXus is about how to find and annotate the data to be plotted but not to describe how the data is to be plotted. (https://www.nexusformat.org/NIAC2018Minutes.html#nxdata-plottype–attribute)

Unified format for reduction and analysis

Another important motivation for NeXus, indeed the *raison d'etre*, was the community need to analyze data from different user facilities. A single data format that is in use at a variety of facilities would provide a major benefit to the scientific community. This should be capable of describing any type of data from the scientific experiments, at any step of the process from data acquisition to data reduction and analysis. This unified format also needs to allow data to be written to storage as efficiently as possible to enable use with high-speed data acquisition.

Self-description, combined with a reliance on a *multi-platform* (and thereby *portable*) data storage format, are valued components of a data storage format where the longevity of the data is expected to be longer than the lifetime of the facility at which it is acquired. As the name implies, self-description within data files is the practice where the structure of the information contained within the file is evident from the file itself. A multi-platform data storage format must faithfully represent the data identically on a variety of computer systems, regardless of the bit order or byte order or word size native to the computer.

The scientific community continues to grow the various types of data to be expressed in data files. This practice is expected to continue as part of the investigative process. To gain broad acceptance in the scientific user community, any data storage format proposed as a standard would need to be *extendable* and continue to provide a means to express the latest notions of scientific data.

The maintenance cost of common data structures meeting the motivations above (self-describing, portable, and extendable) is not insurmountable but is often well-beyond the research funding of individual members of the muon, neutron, and X-ray science communities. Since it is these members that drive the selection of a data storage format, it is necessary for the user cost to be as minimal as possible. In this case, experience has shown that the format must be in the *public-domain* for it to be commonly accepted as a standard. A benefit of the public-domain aspect is that the source code for the API is open and accessible, a point which has received notable comment in the scientific literature.

More recently, NeXus has recognized that many facilities face increased performance requirements and support for writing HDF5 directly in high level languages has become better (for example with h5py for Python). For that reason HDF5 has become the default recommended storage format for NeXus and the use of the NeXus API for new projects is no longer encouraged. In NeXus has recently defined encoding of information in ways that are not compatible with the existing HDF4 and XML container formats (using attribute arrays). The move to HDF5 is strongly advised.

For cases where legacy support of the XML or HDF4 storage backends is required the NeXus API will still be maintained though and provide an upgrade path via the utilities to convert between the different backends.

Defined dictionary of terms

A necessary feature of a standard for the interchange of scientific data is a ` *defined dictionary* (or *lexicography*) of terms. This dictionary declares the expected spelling and meaning of terms when they are present so that it is not necessary to search for all the variant forms of *energy* when it is used to describe data (e.g., `E`, `e`, `keV`, `eV`, `nrg`, . . .).

NeXus recognized that each scientific specialty has developed a unique dictionary and needs to categorize data using those terms. NeXus Application Definitions provide the means to document the lexicography for use in data files of that scientific specialty.

NAPI: The NeXus Application Programming Interface

The NeXus API consists of routines to read and write NeXus data files. It was written to provide a simple to use and consistent common interface for all supported backends (XML, HDF4 and HDF5) to scientific programmers and other users of the NeXus Data Standard.

Note: It is not necessary to use the NAPI to write or read NeXus data files. The intent of the NAPI is to simplify the programming effort to use the HDF programming interface. There are *Examples of writing and reading NeXus data files* to help you understand.

This section will provide a brief overview of the available functionality. Further documentation of the NeXus Application Programming Interface (NAPI) for bindings to specific programming language can be found in the *NAPI* chapter and may be downloaded from the NeXus development site.[1]

For an even more detailed description of the internal workings of NAPI see the NeXus Internals manual, copied from the NeXus code repository. That document is written for programmers who want to work on the NAPI itself. If you are new to NeXus and just want to implement basic file reading or writing you should not start by reading that.

[1] https://github.com/nexusformat/code/releases/

How do I write a NeXus file?

The NeXus Application Program Interface (NAPI) provides a set of subroutines that make it easy to read and write NeXus files. These subroutines are available in C, Fortran 77, Fortran 90, Java, Python, C++, and IDL.

The API uses a very simple *state* model to navigate through a NeXus file. When you open a file, the API provides a file *handle*, which stores the current location, i.e. which group and/or field is currently open. Read and write operations then act on the currently open entity. Following the simple example titled *Example structure of a simple data file*, we walk through a schematic of NeXus program written in C (without any error checking or real data).

Writing a simple NeXus file using NAPI

Note: We assume the program can define the arrays `tth` and `counts`, each length `n`. This part has been omitted from the example code.

```c
#include "napi.h"

 int main()
 {
    /* we start with known arrays tth and counts, each length n */
    NXhandle fileID;
    NXopen ("NXfile.nxs", NXACC_CREATE, &fileID);
      NXmakegroup (fileID, "Scan", "NXentry");
      NXopengroup (fileID, "Scan", "NXentry");
        NXmakegroup (fileID, "data", "NXdata");
        NXopengroup (fileID, "data", "NXdata");
          NXmakedata (fileID, "two_theta", NX_FLOAT32, 1, &n);
          NXopendata (fileID, "two_theta");
            NXputdata (fileID, tth);
            NXputattr (fileID, "units", "degrees", 7, NX_CHAR);
          NXclosedata (fileID);  /* two_theta */
          NXmakedata (fileID, "counts", NX_FLOAT32, 1, &n);
          NXopendata (fileID, "counts");
            NXputdata (fileID, counts);
          NXclosedata (fileID);  /* counts */
        NXclosegroup (fileID);  /* data */
      NXclosegroup (fileID);  /* Scan */
    NXclose (&fileID);
    return;
}
```

program analysis

1. **line 7:**
   Open the file `NXfile.nxs` with *create* access (implying write access). NAPI[2] returns a file identifier of type `NXhandle`.
2. **line 7:**
   Next, we create the NXentry group to contain the scan using `NXmakegroup()` and then open it for access using `NXopengroup()`.[3]
3. **line 10:**
   The plottable data is contained within an NXdata group, which must also be created and opened.
4. **line 12:**
   To create a field, call `NXmakedata()`, specifying the data name, type (`NX_FLOAT32`), rank (in this case, `1`), and length of the array (`n`). Then, it can be opened for writing.[4]
5. **line 14:**
   Write the data using `NXputdata()`.
6. **line 15:**
   With the field still open, we can also add some field attributes, such as the data units,[5][6] which are specified as a character string (`type="NX_CHAR"`[7]) that is 7 bytes long.
7. **line 16:**
   Then we close the field before opening another. In fact, the API will do this automatically if you attempt to open another field, but it is better style to close it yourself.
8. **line 17:**
   The remaining fields in this group are added in a similar fashion. Note that the indentation whenever a new field or group are opened is just intended to make the structure of the NeXus file more transparent.
9. **line 20:**
   Finally, close the groups (`NXdata` and `NXentry`) before closing the file itself.

How do I read a NeXus file?

Reading a NeXus file works in the same way by traversing the tree with the handle.

This schematic C code will read the two-theta array created in the *example above*. (Again, compare this example with *Reading a simple NeXus file using native HDF5 commands in C*.)

[2] *NAPI: NeXus Application Programmer Interface (frozen)*
[3] See the chapter base.class.definitions for more information.
[4] The *NeXus Data Types* section describes the available data types, such as `NX_FLOAT32` and `NX_CHAR`.
[5] *NeXus Data Units*
[6] The NeXus rule about data units is described in the *NeXus Data Units* section.
[7] see *Data Types allowed in NXDL specifications*

Reading a simple NeXus file using NAPI

```
NXopen ('NXfile.nxs', NXACC_READ, &fileID);
  NXopengroup (fileID, "Scan", "NXentry");
    NXopengroup (fileID, "data", "NXdata");
      NXopendata (fileID, "two_theta");
        NXgetinfo (fileID, &rank, dims, &datatype);
        NXmalloc ((void **) &tth, rank, dims, datatype);
        NXgetdata (fileID, tth);
      NXclosedata (fileID);
    NXclosegroup (fileID);
  NXclosegroup (fileID);
NXclose (fileID);
```

How do I browse a NeXus file?

NeXus files can also be viewed by a command-line browser, `nxbrowse`, which is included as a helper tool in the *NeXus API* distribution. The *following* is an example session of nxbrowse `nxbrowse` to view a data file.

Using `nxbrowse`

```
%> nxbrowse lrcs3701.nxs

NXBrowse 3.0.0. Copyright (C) 2000 R. Osborn, M. Koennecke, P. Klosowski
    NeXus_version = 1.3.3
    file_name = lrcs3701.nxs
    file_time = 2001-02-11 00:02:35-0600
    user = EAG/RO
NX> dir
  NX Group : Histogram1 (NXentry)
  NX Group : Histogram2 (NXentry)
NX> open Histogram1
NX/Histogram1> dir
  NX Data  : title[44] (NX_CHAR)
  NX Data  : analysis[7] (NX_CHAR)
  NX Data  : start_time[24] (NX_CHAR)
  NX Data  : end_time[24] (NX_CHAR)
  NX Data  : run_number (NX_INT32)
  NX Group : sample (NXsample)
  NX Group : LRMECS (NXinstrument)
  NX Group : monitor1 (NXmonitor)
  NX Group : monitor2 (NXmonitor)
  NX Group : data (NXdata)
NX/Histogram1> read title
  title[44] (NX_CHAR) = MgB2 PDOS 43.37g 8K 120meV E0@240Hz T0@120Hz
NX/Histogram1> open data
NX/Histogram1/data> dir
  NX Data  : title[44] (NX_CHAR)
  NX Data  : data[148,750] (NX_INT32)
```

(continues on next page)

(continued from previous page)

```
29    NX Data  : time_of_flight[751] (NX_FLOAT32)
30    NX Data  : polar_angle[148] (NX_FLOAT32)
31  NX/Histogram1/data> read time_of_flight
32    time_of_flight[751] (NX_FLOAT32) = [ 1900.000000 1902.000000 1904.000000 ...]
33      units = microseconds
34      long_name = Time-of-Flight [microseconds]
35  NX/Histogram1/data> read data
36    data[148,750] (NX_INT32) = [ 1 1 0 ...]
37      units = counts
38      signal = 1
39      long_name = Neutron Counts
40      axes = polar_angle:time_of_flight
41  NX/Histogram1/data> close
42  NX/Histogram1> close
43  NX> quit
```

program analysis

1. **line 1:**
   Start `nxbrowse` from the UNIX command line and open file `lrcs3701.nxs` from IPNS/LRMECS.
2. **line 8:**
   List the contents of the current group.
3. **line 11:**
   Open the NeXus group `Histogram1`.
4. **line 23:**
   Print the contents of the NeXus data labeled `title`.
5. **line 41:**
   Close the current group.
6. **line 43:**
   Quits `nxbrowse`.

The source code of `nxbrowse`[8] provides an example of how to write a NeXus reader. The test programs included in the *NeXus API* may also be useful to study.

1.2 NeXus Design

This chapter actually defines the rules to use for writing valid NeXus files. An explanation of NeXus objects is followed by the definition of NeXus coordinate systems, the rules for structuring files and the rules for storing single items of data.

The structure of NeXus files is extremely flexible, allowing the storage both of simple data sets, such as a single data array and its axes, and also of highly complex data, such as the simulation results or an entire multi-component instrument. This flexibility is a necessity as NeXus strives to capture data from a wild variety of applications in X-ray, muSR and neutron scattering. The flexibility is achieved through a hierarchical structure, with related *fields* collected together into *groups*, making NeXus files easy to navigate, even without any documentation. NeXus files are self-describing, and should be easy to understand, at least by those familiar with the experimental technique.

[8] https://github.com/nexusformat/code/blob/master/applications/NXbrowse/NXbrowse.c

1.2.1 NeXus Objects and Terms

Before discussing the design of NeXus in greater detail it is necessary to define the objects and terms used by NeXus. These are:

Groups
    Levels in the NeXus hierarchy. May contain fields and other groups.

Fields
    Multidimensional arrays and scalars representing the actual data to be stored

Attributes
    Attributes containing additional metadata can be assigned to groups, fields, or *files*.

Links
    Elements which point to data stored in another place in the file hierarchy

NeXus Base Classes
    Dictionaries of names possible in the various types of NeXus groups

NeXus Application Definitions
    Describe the minimum content of a NeXus file for a particular usage case

In the following sections these elements of NeXus files will be defined in more detail.

Note: Notation used to describe a NeXus data file

In various places in the NeXus manual, contents of a NeXus data file are described using a tree structure, such as in the *Introduction*.

The tree syntax is a very condensed version (with high information density) meant to convey the structure of the HDF file.

- Groups have a `/` appended to their name (with NeXus class name shown)
- Indentation shows membership in the lesser indented parent above.
- Fields have a data type and value appended (for arrays, this may be an abbreviated view)
- Attributes (of groups or fields) are prefixed with `@`.
- NeXus-style links are described with some sort of arrow notation such as `-->`.

Groups

NeXus files consist of data groups, which contain fields and/or other groups to form a hierarchical structure. This hierarchy is designed to make it easy to navigate a NeXus file by storing related fields together. Data groups are identified both by a name, which must be unique within a particular group, and a class. There can be multiple groups with the same class but they must have different names (based on the HDF rules).

For the class names used with NeXus data groups the prefix NX is reserved. Thus all NeXus class names start with NX.

Fields

Fields (also called data fields, data items or data sets) contain the essential information stored in a NeXus file. They can be scalar values or multidimensional arrays of a variety of sizes (1-byte, 2-byte, 4-byte, 8-byte) and types (integers, floats, characters). The fields may store both experimental results (counts, detector angles, etc), and other information associated with the experiment (start and end times, user names, etc). Fields are identified by their names, which must be unique within the group in which they are stored. Some fields have engineering units to be specified. In some cases, such as /NXdata/DATA, a field is expected to have be an array of several dimensions.

Examples of fields

`variable` (*NX_NUMBER*)
Dimension scale defining an axis of the data.

`variable_errors` (*NX_NUMBER*)
Errors (uncertainties) associated with axis `variable`.

`wavelength` (*NX_FLOAT*)
wavelength of radiation, `units="NX_FLOAT"`

`chemical_formula` (*NX_CHAR*)
The chemical formula specified using CIF conventions.

`name` (*NX_CHAR*)
Name of user responsible for this entry.

`data` (*NX_NUMBER*)
Data values from the detector, `units="NX_ANY"`

See the sections *Data Types allowed in NXDL specifications* and *Unit Categories allowed in NXDL specifications* for complete lists of the data types and engineering units types, respectively.

In the case of streaming data acquisition, when time-stamped values of data are collected, fields can be replaced with NXlog structures of the same name. For example, if time stamped data for wavelength is being streamed, wavelength would not be an array but a NXlog structure.

Attributes

Attributes are extra (meta-)information that are associated with particular groups or fields. They are used to annotate data, e.g. with physical units or calibration offsets, and may be scalar numbers or character strings. In addition, NeXus uses attributes to identify plottable data and their axes, etc. In a *tree structure*, an attribute is usually shown with a `@` prefix, such as `@units`. A description of some of the many possible attributes can be found in the next table:

Examples of attributes

`units` (*NX_CHAR*)
Data units given as character strings, must conform to the NeXus units standard. See the *NeXus Data Units* section for details.

`signal` (*NX_CHAR*)
Defines which data set contains the signal to be plotted. Use `signal="{dataset_name}"` where `{dataset_name}` is the name of a field (or link to a field) in the NXdata group. The field referred to by the *signal* attribute might be referred to as the "signal data".

`long_name` (*NX_CHAR*)
Defines title of signal data or axis label of dimension scale

calibration_status (*NX_CHAR*)
Defines status of data value - set to `Nominal` or `Measured`

data_offset (*NX_INT*)
Rank values of offsets to use for each dimension if the data is not in C storage order

interpretation (*NX_CHAR*)
Describes how to display the data. `rgba`, `hsla` and `cmyk` are (*n x m x 4*) arrays, where the 4 channels are the colour channels appropriately. If the image data does not contain an alpha channel, then the array should simply be (*n x m x 3*). Allowed values include:

- `scalar` (0-D data)
- `scaler` DEPRECATED, use `scalar`
- `spectrum` (1-D data)
- `image` (2-D data)
- `rgb-image` (3-D data)
- `rgba-image` (3-D data)
- `hsl-image` (3-D data)
- `hsla-image` (3-D data)
- `cmyk-image` (3-D data)
- `vertex` (3-D data)

File attributes

Finally, some attributes are defined at file level. They are specified in the base class NXroot.

Links

Python h5py code to make NeXus links

The section titled *HDF5 in Python with h5py* provides example python code to create links (both internal and external) in NeXus data files. See the routines:

- **{hdf5_object}._id.link()**
- **h5py.ExternalLink()**

Links are pointers to existing data somewhere else. The concept is very much like symbolic links in a unix filesystem. The NeXus definition sometimes requires to have access to the same data in different groups in the same file. For example: detector data is stored in the NXinstrument/NXdetector group but may be needed in NXdata for automatic plotting. Rather then replicating the data, NeXus uses links in such situations. See the *figure* for a more descriptive representation of the concept of linking.

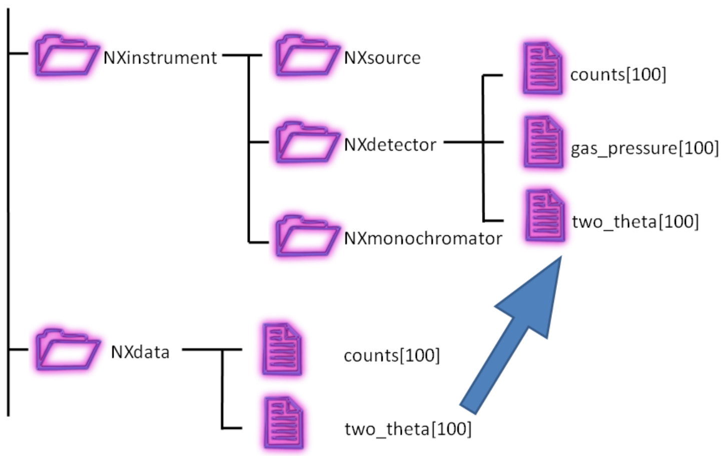


Fig. 1: Linking in a NeXus file

NeXus links are HDF5 hard links with an additional `target` attribute. The `target` attribute is added[1] for NeXus to distinguish the HDF5 path to the *original*[2] dataset. The value of the `target` attribute is the HDF5 path[3] to the *original* dataset.

NeXus links are best understood with an example. The canonical location (expressed as a NeXus class path) to store wavelength (see *Strategies: The wavelength*) has been:

```
/NXentry/NXinstrument/NXcrystal/wavelength
```

An alternative location for this field makes sense to many, especially those not using a crystal to create monochromatic radiation:

```
/NXentry/NXinstrument/NXmonochromator/wavelength
```

These two fields might be hard linked together in a NeXus data file (using HDF5 paths such `/entry/instrument`):

[1] When using the NAPI, the `target` attribute is added automatically. When the NAPI is not used to write NeXus/HDF5 files, this attribute must be added. Here are the steps to follow:

1. Get the HDF5 reference ID of the source item (*field*, *group*, or *link*) to be linked.
2. If the ID does not have a `target` attribute defined: #. Get the absolute HDF5 address[Page 21, 3] of the ID. #. Create a `target` attribute for the ID. #. Set the `target` attribute's value to the absolute HDF5 address of the ID.
3. Create an HDF5 hard link[4] to the ID at the desired (new) HDF5 address.

[3] When using the `target` attribute, **always** specify the HDF5 address as an *absolute** address (starts from the HDF5 root, such as: `/entry/instrument/detector/polar_angle`) rather than a **relative** address (starting from the current group, such as: `detector/polar_angle`).

Note: The `target` attribute does not work for *external file links*. The NIAC is working at resolving the technical limitations

[4] HDF5 hard link: https://portal.hdfgroup.org/display/HDF5/H5L_CREATE_HARD

[2] The notion of an *original* dataset with regard to links is a NeXus abstraction. In truth, HDF5 makes no distinction which is the *original* dataset. But, when the file is viewed with a tool such as *h5dump*, confusion often occurs over which dataset is original and which is a link to the original. Actually, both HDF5 paths point to the exact, same dataset which exists at a specific offset in the HDF5 file.

See the *Frequently Asked Questions* question: **I'm using links to place data in two places. Which one should be the data and which one is the link?**

```
entry:NXentry
    ...
    instrument:NXinstrument
        ...
        crystal:NXcrystal
            ...
            wavelength:NX_FLOAT = 154.
                @target="/entry/instrument/crystal/wavelength"
                @units="pm"
        ...
        monochromator:NXmonochromator
            ...
            wavelength --> "/entry/instrument/crystal/wavelength"
```

It is possible that the linked field or group has a different name than the original. One obvious use of this capability is to adapt to a specific requirement of an application definition. For example, suppose some application definition required the specification of wavelength as a field named *lambda* in the entry group. This requirement can be satisifed easily:

```
entry:NXentry
    ...
    instrument:NXinstrument
        ...
        crystal:NXcrystal
            ...
            wavelength:NX_FLOAT = 154.
                @target="/entry/instrument/crystal/wavelength"
                @units="pm"
        ...
        monochromator:NXmonochromator
            ...
            wavelength --> "/entry/instrument/crystal/wavelength"
    ...
    lambda --> "/entry/instrument/crystal/wavelength"
```

External File Links

NeXus also allows for links to external files. Consider the case where an instrument uses a detector with a closed-system software support provided by a commercial vendor. This system writes its images into a NeXus HDF5 file. The instrument's data acquisition system writes instrument metadata into another NeXus HDF5 file. In this case, the instrument metadata file might link to the data in the detector image file. Here is an example (from Diamond Light Source) showing an external file link in HDF5:

Example of linking to data in an external HDF5 file

```
EXTERNAL_LINK "data" {
	TARGETFILE "/dls/i22/data/2012/sm7594-1/i22-69201-Pilatus2M.h5"
	TARGETPATH "entry/instrument/detector/data"
}
```

Note: The NAPI code[5] makes no `target` attribute assignment for links to external files. It is best to avoid using the `target` attribute with external file links. The NIAC is working at resolving the technical limitations

The NAPI maintains a group attribute `@napimount` that provides a URL to a group in another file. More information about the `@napimount` attribute is described in the *NeXus Programmers Reference*.[6]

Combining NeXus links and External File Links

Consider the case described in *Links to Data in External HDF5 Files*, where numerical data are provided in two different HDF5 files and a *master* NeXus HDF5 file links to the data through external file links. HDF5 will not allow hard links to be constructed with these data objects in the master file. An error such as *Interfile hard links are not allowed* (as generated from h5py) will arise. This makes sense since there is no such data object in the file.

Instead, it is necessary to make an external file link at each place in the master where external data is to be represented.

NeXus Base Classes

Data groups often describe objects in the experiment (monitors, detectors, monochromators, etc.), so that the contents (both fields and/or other groups) comprise the properties of that object. NeXus has defined a set of standard objects, or base classes, out of which a NeXus file can be constructed. This is each data group is identified by a name and a class. The group class, defines the type of object and the properties that it can contain, whereas the group name defines a unique instance of that class. These classes are defined in XML using the NeXus Definition Language (NXDL) format. All NeXus class types adopted by the NIAC *must* begin with `NX`. Classes not adopted by the NIAC *must not* start with `NX`.

Note: NeXus base classes are the components used to build the NeXus data structure.

Not all classes define physical objects. Some refer to logical groupings of experimental information, such as plottable data, sample environment logs, beam profiles, etc. There can be multiple instances of each class. On the other hand, a typical NeXus file will only contain a small subset of the possible classes.

Note: The groups, fields, links, and attributes of a base class definition are all **optional**, with a few particular exceptions in `NXentry` and `NXdata`. They are named in the specification to describe the exact spelling and usage of the term when it appears.

NeXus base classes are not proper classes in the same sense as used in object oriented programming languages. In fact the use of the term classes is actually misleading but has established itself during the development of NeXus. NeXus base classes are rather dictionaries of field names and their meanings which are permitted in a particular NeXus group implementing the NeXus class. This sounds complicated but becomes easy if you consider that most NeXus groups

[5] `NX5nativeexternallink()`: https://github.com/nexusformat/code/blob/fe8ddd287ee33961982931e2016cc25f76f95edd/src/napi5.c#L2248

[6] https://manual.nexusformat.org/_static/NeXusIntern.pdf

describe instrument components. Then for example, a NXmonochromator base class describes all the possible field names which NeXus allows to be used to describe a monochromator.

Most NeXus base classes represent instrument components. Some are used as containers to structure information in a file (`NXentry`, `NXcollection`, `NXinstrument`, `NXprocess`, `NXparameters`). But there are some base classes which have special uses which need to be mentioned here:

NXdata
`NXdata` is used to identify the default plottable data. The notion of a default plot of data is a basic motivation of NeXus. (see *Simple plotting*)

NXlog
`NXlog` is used to store time stamped data like the log of a temperature controller. Basically you give a start time, and arrays with a difference in seconds to the start time and the values read.

NXcollection
`NXcollection` is used to gather together any set of terms. Anything (groups, fields, or attributes) placed in an `NXcollection` group will not be validated. One use is to use this as a container class for the various control system variables from a beamline or instrument.

NXnote
This group provides a place to store general notes, images, video or whatever. A mime type is stored together with a binary blob of data. Please use this only for auxiliary information, for example an image of your sample, or a photo of your boss.

NXtransformations

`NXtransformations` is used to gather together any set of movable or fixed
elements positioning the device described by the class that contains this. Supercedes `NXgeometry`.

NXgeometry (superceded by NXtransformations,[Page 24, 7]**)**

`NXgeometry` and its subgroups `NXtranslation`, `NXorientation`, `NXshape` are used to store absolute positions in the laboratory coordinate system or to define shapes.

These groups can appear anywhere in the NeXus hierarchy, where needed. Preferably close to the component they annotate or in a `NXcollection`. All of the base classes are documented in the reference manual.

`NXdata` Facilitates Automatic Plotting

The most notable special base class (or *group* in NeXus) is `NXdata`. `NXdata` is the answer to a basic motivation of NeXus to facilitate automatic plotting of data. `NXdata` is designed to contain the main dataset and its associated dimension scales (axes) of a NeXus data file. The usage scenario is that an automatic data plotting program just opens a `NXentry` and then continues to search for any `NXdata` groups. These `NXdata` groups represent the plottable data. An algorithm for identifying the default plottable data is *presented* in the chapter titled *Rules for Storing Data Items in NeXus Files*.

[7] see: https://github.com/nexusformat/definitions/issues/397

Where to Store Metadata

There are many ways to store metadata about your experiments. Already there are many fields in the various base classes to store the more common or general metadata, such as wavelength. (For wavelength, see the *Strategies: The wavelength* section.)

One common scheme is to store the metadata all in one group. If the group is to be validated for content, then there are several possibilities, as shown in the next table:

base class	intent
NXnote	to store additional information
NXlog	information that is time-stamped
NXparameters	parameters for processing or analysis
NXcollection	to store *any* unvalidated content

If the content of the metadata group is to be excluded from validation, then store it in a NXcollection group.

NeXus Application Definitions

The objects described so far provide us with the means to store data from a wide variety of instruments, simulations, or processed data as resulting from data analysis. But NeXus strives to express strict standards for certain applications of NeXus, too. The tool which NeXus uses for the expression of such strict standards is the NeXus Application Definition. A NeXus Application Definition describes which groups and data items have to be present in a file in order to properly describe an application of NeXus. For example for describing a powder diffraction experiment. An application definition may also declare terms which are optional in the data file. Typically an application definition will contain only a small subset of the many groups and fields defined in NeXus. NeXus application definitions are also expressed in the NeXus Definition Language (NXDL). A tool exists which allows one to validate a NeXus file against a given application definition.

Note: NeXus application definitions define the *minimum required* information necessary to satisfy data analysis or other data processing.

Another way to look at a NeXus application definition is as a contract between a file producer (writer) and a file consumer (reader).

> The contract reads: *If you write your files following a particular NeXus application definition, I can process these files with my software.*

Yet another way to look at a NeXus application definition is to understand it as an interface definition between data files and the software which uses this file. Much like an interface in the Java or other modern object oriented programming languages.

In contrast to NeXus base classes, NeXus supports inheritance in application definitions.

Please note that a NeXus Application Definition will only define the bare minimum of data necessary to perform common analysis with data. Practical files will nearly always contain more data. One of the beauties of NeXus is that it is always possible to add more data to a file without breaking its compliance with its application definition.

1.2.2 NeXus Geometry

NeXus supports description of the shape, position and orientation of objects in *The NeXus Coordinate System*. Position and orientation can be defined as *Coordinate Transformations* using the NXtransformations class. *Shape Descriptions* use the NXoff_geometry or NXcylindrical_geometry class.

You may come across old files which use *Legacy Geometry Descriptions*.

The NeXus Coordinate System

The NeXus coordinate system is shown *below*. Note that it is the same as that used by *McStas* (http://mcstas.org). This choice is arbitrary and any other choice should be possible as long as it is used consistently and application code that reads NeXus files does not assume any prior knowledge of the chosen coordinate system.

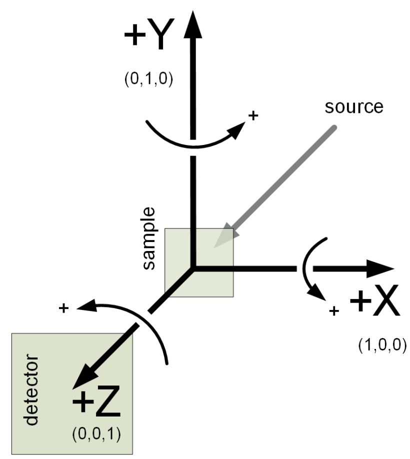


Fig. 2: NeXus coordinate system, as viewed from detector

Note: The NeXus definition of $+z$ is opposite to that in the IUCr International Tables for Crystallography, volume G.

Coordinate Transformations

In the recommended way of dealing with geometry NeXus uses a series of transformations to place objects in space. In this world view, the absolute position of a component or a detector pixel with respect to the laboratory coordinate system is calculated by applying a series of translations and rotations. Thus a rotation or translation operation transforms the whole coordinate system and gives rise to a new local coordinate system. These transformations between coordinate systems are mathematical operations and can be expressed as matrices and their combination as matrix multiplication. A very important aspect is that the order of application of the individual operations *does* matter. The mathematics behind this is well known and used in such applications such as industrial robot control, flight dynamics and computer games. The beauty in this comes from the fact that the operations to apply map easily to instrument settings and constants. It is also easy to analyze the contribution of each individual operation: this can be studied under the condition that all other operations are at a zero setting.

In order to use coordinate transformations, several pieces of information need to be known:

Type
: The type of operation: rotation or translation

Direction
The direction of the translation or the direction of the rotation axis

Value
The angle of rotation or the length of the translation

Order
The order of operations to apply to move a component into its place.

Coordinate Transformation Field And Attributes

NeXus chooses to encode information about each transformation as a field in an `NXtransformations` group in the following way:

`value`
This is represented in the actual data of the field or the **value** of the transformation. Its actual name should relate to the physical device used to effect the transformation.

The coordinate transformation attributes are:

`transformation_type`
This specifies the **type** of transformation and is either *rotation* or *translation* and describes the kind of operation performed

`vector` (*NX_NUMBER*)
This is a set of 3 values forming a unit vector for **direction** that describes the components of either the direction of the rotation axis or the direction along which the translation happens.

`offset` (*NX_NUMBER*)
This is a set of 3 values forming the offset vector for a translation to apply before applying the operation of the actual transformation. Without this offset attribute, additional virtual translations would need to be introduced in order to encode mechanical offsets in the axis.

`depends_on`
The **order** is encoded through this attribute. The value is the name of the transformation upon which the current transformation depends on.

As each transformation represents possible motion by a physical device, this dependency expresses the attachment order; thus, the current device is attached to (or mounted on) the next device referred to by the attribute.

Allowed values for depends_on are:

`.`
A dot ends the depends_on chain

`name`
The name of a field within the enclosing group

`dir/name`
The name of a field further along the path

`/dir/dir/name`
An absolute path to a field in another group

In addition, for each beamline component, there is a `depends_on` attribute that points to the field at the head of the axis dependency chain. For example, consider an eulerian cradle as used on a four-circle diffractometer. Such a cradle has a dependency chain of `phi:chi:rotation_angle`. Then the `depends_on` field in NXsample would have the value `phi`.

NeXus Transformation encoding

Transformation encoding for an eulerian cradle on a four-circle diffractometer

```
sample:NXsample
  transforms:NXtransformations
    rotation_angle
      @transformation_type=rotation
      @vector=0,1,0
      @offset=0,0,0
      @depends_on=.
    chi
      @transformation_type=rotation
      @vector=0,0,1
      @offset=0,0,0
      @depends_on=rotation_angle
    phi
      @transformation_type=rotation
      @vector=0,1,0
      @offset=0,0,0
      @depends_on=chi
  depends_on
    transforms/phi
```

The type and direction of the NeXus standard operations is documented below in the table: *Actions of standard NeXus fields*. The rule is to always give the attributes to make perfectly clear how the axes work. The CIF scheme also allows to store and use arbitrarily named axes in a NeXus file.

The CIF scheme (see NXtransformations) is the preferred method for expressing geometry in NeXus.

Actions of standard NeXus fields

Transformation Actions

Field Name	transformation_type	vector
polar_angle	rotation	0 1 0
azimuthal_angle	rotation	0 0 1
meridional_angle	rotation	1 0 0
distance	translation	0 0 1
height	translation	0 1 0
x_translation	translation	1 0 0
chi	rotation	0 0 1
phi	rotation	0 1 0

For the NeXus spherical coordinate system (described in the legacy section below), the order is implicit and is given in the next example.

implicit order of NeXus spherical coordinate system

```
azimuthal_angle:polar_angle:distance
```

This is also a nice example of the application of transformation matrices:

1. You first apply `azimuthal_angle` as a rotation around z. This rotates the whole coordinate out of the plane.
2. Then you apply `polar_angle` as a rotation around y in the tilted coordinate system.
3. This also moves the direction of the z vector. Along which you translate the component to place by distance.

Shape Descriptions

NXoff_geometry

The shape of instrument components can be described using the NXoff_geometry class. `NXoff_geometry` is a polygon-based description, based on the open OFF format. Conversion between OFF files and the NeXus description is straightforward. This is beneficial as existing tools can use, view or manipulate the geometry in OFF files. CAD software, for example FreeCAD, can be used to define the geometry. 3D rendering tools such as Geomview can be used to view the geometry. McStas can use OFF files to define the shape of components for scattering simulations.

The example OFF file shown below defines a cube. The first line containing numbers defines: the number of vertices, the number of faces (polygons) making up the model's surface, and the number of edges in the mesh. Note, the number of edges must be present but does not need to be correct (http://www.geomview.org/docs/html/OFF.html).

```
OFF
#  cube.off
#  A cube

8 6 12
1.0   0.0   1.0
0.0   1.0   1.0
-1.0   0.0   1.0
0.0  -1.0   1.0
1.0   0.0  0.0
0.0   1.0  0.0
-1.0   0.0  0.0
0.0  -1.0  0.0
4  0 1 2 3
4  7 4 0 3
4  4 5 1 0
4  5 6 2 1
4  3 2 6 7
4  6 5 4 7
```

Following the initial line are the xyz coordinates of each vertex. Proceeding which is the list of faces. Each line defining a face starts with the number of vertices in that face followed by the sequence number of the composing vertices, indexed from zero. The vertex indices form a winding order by defining the face normal by the right-hand rule. The number of vertices in each face need not be constant; a mesh can comprise of polygons of many different orders.

The list of vertices in an OFF file maps directly to the `vertices` dataset in the NXoff_geometry class. The vertex indices of the face list in the OFF file occupy the `winding_order` dataset of the NeXus class, however the list is flattened to 1D in order to avoid a ragged-edged dataset, which are not easy to work with using HDF libraries. A `faces` dataset contains the position of the first entry in `winding_order` for each face. The `NXoff_geometry` equivalent of the OFF cube example is shown below.

```
shape : NXoff_geometry
  @NX_class = "NXoff_geometry"
  vertices =
    1.0,   0.0,   1.0
    0.0,   1.0 ,  1.0
    -1.0,  0.0,   1.0
    0.0,   -1.0,  1.0
    1.0,   0.0,   0.0
    0.0,   1.0,   0.0
    -1.0,  0.0,   0.0
    0.0,   -1.0,  0.0
  faces =
    0, 4, 8, 12, 16, 20
  winding_order =
    0, 1, 2, 3, 7, 4, 0, 3, 4, 5, 1, 0, 5, 6, 2, 1, 3, 2, 6, 7, 6, 5, 4, 7
```

`NXcylindrical_geometry`

Although the polygon-based description of NXoff_geometry is very flexible, it is not ideal for curved shapes when high precision is required since a very large number of vertices may be necessary. A common example of this is when describing helium tube, neutron detectors. NXcylindrical_geometry provides a more concise method of defining shape for such cases.

Like `NXoff_geometry`, `NXcylindrical_geometry` contains a `vertices` dataset. The indices of three vertices (**A**, **B**, **C** in *Cylinder definition with three vertices*) in the `vertices` dataset are used to define each cylinder in the `cylinders` dataset.

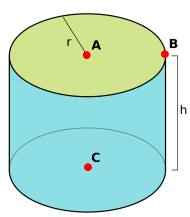


Fig. 3: Cylinder definition with three vertices

Detector Shape Descriptions

An `NXoff_geometry` or `NXcylindrical_geometry` group named `detector_shape` can be placed in an `NXdetector` or `NXdetector_module` to define the complete shape of the detector. Alternatively, the group can be named `pixel_shape` and define the shape of a single pixel. In this case, `x_pixel_offset`, `y_pixel_offset` and `z_pixel_offset` datasets of the `NXdetector` define how the pixel shape is tiled to form the geometry of the complete detector.

Legacy Geometry Descriptions

The above system of chained transformations is the recommended way of encoding geometry going forward. This section describes the traditional way this was handled in NeXus, which you may find occasionally in old files.

Coordinate systems in NeXus have undergone significant development. Initially, only motor positions of the relevant motors were stored without further standardization. This soon proved to be too little and the *NeXus polar coordinate* system was developed. This system still is very close to angles that are meaningful to an instrument scientist but allows to define general positions of components easily. Then users from the simulation community approached the NeXus team and asked for a means to store absolute coordinates. This was implemented through the use of the *NXgeometry* class on top of the *McStas* system. We soon learned that all the things we do can be expressed through the McStas coordinate system. So it became the reference coordinate system for NeXus. `NXgeometry` was expanded to allow the description of shapes when the demand came up. Later, members of the CIF team convinced the NeXus team of the beauty of transformation matrices and NeXus was enhanced to store the necessary information to fully map CIF concepts. Not much had to be changed though as we choose to document the existing angles in CIF terms. The CIF system allows to store arbitrary operations and nevertheless calculate absolute coordinates in the laboratory coordinate system. It also allows to convert from local, for example detector coordinate systems, to absolute coordinates in the laboratory system.

McStas and `NXgeometry` System

As stated above, NeXus uses the *McStas coordinate system* (http://mcstas.org) as its laboratory coordinate system. The instrument is given a global, absolute coordinate system where the z axis points in the direction of the incident beam, the x axis is perpendicular to the beam in the horizontal plane pointing left as seen from the source, and the y axis points upwards. See below for a drawing of the McStas coordinate system. The origin of this coordinate system is the sample position or, if this is ambiguous, the center of the sample holder with all angles and translations set to zero. The McStas coordinate system is illustrated in the next figure:

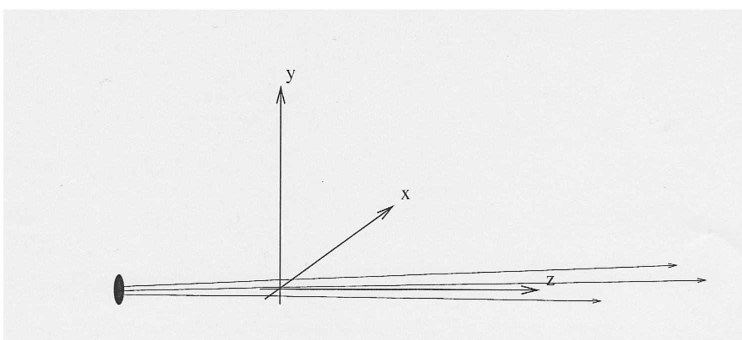


Fig. 4: The McStas Coordinate System

The NeXus NXgeometry class directly uses the McStas coordinate system. `NXgeometry` classes can appear in any component in order to specify its position. The suggested name to use is geometry. In `NXgeometry` the `NXtranslation/values` field defines the absolute position of the component in the McStas coordinate system. The `NXorientation/value` field describes the orientation of the component as a vector of in the McStas coordinate system.

Simple (Spherical Polar) Coordinate System

In this system, the instrument is considered as a set of components through which the incident beam passes. The variable *distance* is assigned to each component and represents the effective beam flight path length between this component and the sample. A sign convention is used where negative numbers represent components pre-sample and positive numbers components post-sample. At each component there is local spherical coordinate system with the angles *polar_angle* and *azimuthal_angle*. The size of the sphere is the distance to the previous component.

In order to understand this spherical polar coordinate system it is helpful to look initially at the common condition that *azimuthal_angle* is zero. This corresponds to working directly in the horizontal scattering plane of the instrument. In this case *polar_angle* maps directly to the setting commonly known as *two theta*. Now, there are instruments where components live outside of the scattering plane. Most notably detectors. In order to describe such components we first apply the tilt out of the horizontal scattering plane as the *azimuthal_angle*. Then, in this tilted plane, we rotate to the component. The beauty of this is that *polar_angle* is always *two theta*. Which, in the case of a component out of the horizontal scattering plane, is not identical to the value read from the motor responsible for rotating the component. This situation is shown in *Polar Coordinate System*.

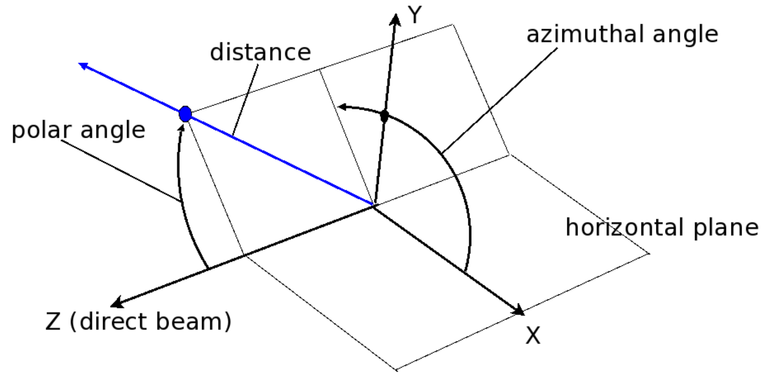


Fig. 5: NeXus Simple (Spherical Polar) Coordinate System

1.2.3 Rules and Underlying File Formats

Rules for Structuring Information in NeXus Files

All NeXus files contain one or many groups of type `NXentry` at root level. Many files contain only one `NXentry` group, then the name is `entry`. The NXentry level of hierarchy is there to support the storage of multiple related experiments in one file. Or to allow the NeXus file to serve as a container for storing a whole scientific workflow from data acquisition to publication ready data. Also, `NXentry` class groups can contain raw data or processed data. For files with more than one `NXentry` group, since HDF requires that no two items at the same level in an HDF file may have the same name, the NeXus fashion is to assign names with an incrementing index appended, such as `entry1`, `entry2`, `entry3`, etc.

In order to illustrate what is written in the text, example hierarchies like the one in figure *Raw Data* are provided.

Content of a Raw Data `NXentry` Group

An example raw data hierarchy is shown in figure *Raw Data* (only showing the relevant parts of the data hierarchy). In the example shown, the `data` field in the `NXdata` group is linked to the 2-D detector data (a 512x512 array of 32-bit integers). The attribute `signal = data` on the NXdata group marks this field as the default plottable data of the `data:NXdata` group. The NXdata group attribute `axes = . .` declares that both dimensions of the `data` field do not have associated dimension scales (plotting routines should use integer scaling for each axis). Note that `[,]` represents a 2D array.

NeXus Raw Data Hierarchy

```
entry:NXentry
    @default = data
    instrument:NXinstrument
        source:NXsource
        ....
        detector:NXdetector
            data:NX_INT32[512,512]
    sample:NXsample
    control:NXmonitor
    data:NXdata
        @signal = data
        @axes = [".", "."]
        data --> /entry/instrument/detector/data
```

An `NXentry` describing raw data contains at least a `NXsample`, one `NXmonitor`, one `NXdata` and a `NXinstrument` group. It is good practice to use the names `sample` for the `NXsample` group, `control` for the `NXmonitor` group holding the experiment controlling monitor and `instrument` for the `NXinstrument` group. The `NXinstrument` group contains further groups describing the individual components of the instrument as appropriate.

The `NXdata` group contains links to all those data items in the `NXentry` hierarchy which are required to put up a default plot of the data. As an example consider a SAXS instrument with a 2D detector. The `NXdata` will then hold a link to the detector image. If there is only one `NXdata` group, it is good practice to name it `data`. Otherwise, the name of the detector bank represented is a good selection.

Content of a processed data `NXentry` group

Processed data, see figure *Processed Data*, in this context means the results of a data reduction or data analysis program. Note that `[]` represents a 1D array.

NeXus Processed Data Hierarchy

```
entry:NXentry
    @default = data
    reduction:NXprocess
        program_name = "pyDataProc2010"
        version = "1.0a"
        input:NXparameters
            filename = "sn2013287.nxs"
    sample:NXsample
```

(continues on next page)

(continued from previous page)

```
        data:NXdata
            @signal = data
            @axes = "."
            data
```

NeXus stores such data in a simplified NXentry structure. A processed data NXentry has at minimum a NXsample, a NXdata and a NXprocess group. Again the preferred name for the NXsample group is sample. In the case of processed data, the NXdata group holds the result of the processing together with the associated axis data. The NXprocess group holds the name and version of the program used for this processing step and further NXparameters groups. These groups ought to contain the parameters used for this data processing step in suitable detail so that the processing step can be reproduced.

Optionally a processed data NXentry can hold a NXinstrument group with further groups holding relevant information about the instrument. The preferred name is again instrument. Whereas for a raw data file, NeXus strives to capture as much data as possible, a NXinstrument group for processed data may contain a much-reduced subset.

NXsubentry or Multi-Method Data

Especially at synchrotron facilities, there are experiments which perform several different methods on the sample at the same time. For example, combine a powder diffraction experiment with XAS. This may happen in the same scan, so the data needs to be grouped together. A suitable NXentry would need to adhere to two different application definitions. This leads to name clashes which cannot be resolved easily. In order to solve this issue, the following scheme was implemented in NeXus:

- The complete beamline (all data) is stored in an appropriate hierarchy in an NXentry.
- The NXentry group contains further NXsubentry groups, one for each method.
- Each NXsubentry group is constructed like a NXentry group. It contains links to all those data items required to fulfill the application definition for the particular. method it represents.
- The name of the application definition is stored in the definition field of the NXsubentry group
- Each NXsubentry group contains a NXdata group describing the default plottable data for that experimental method. To satisfy the NeXus requirement of finding the default plottable data from a NXentry group, the NXdata group from one of these NXsubentry groups (the fluoresence data) was linked.

See figure *NeXus Multi Method Hierarchy* for an example hierarchy. Note that [,] represents a 2D array.

NeXus Multi Method Hierarchy

```
    entry:NXentry
        @default = data
        user:NXuser
        sample:NXsample
        instrument:NXinstument
            SASdet:NXdetector
                data:[,]
            fluordet:NXdetector
                data:[,]
            large_area:NXdetector
                data:[,]
        SAS:NXsubentry
```

(continues on next page)

(continued from previous page)

```
            definition = "NXsas"
            instrument:NXinstrument
                detector:NXdetector
                    data --> /entry/instrument/SASdet/data
            data:NXdata
                data --> /entry/instrument/SASdet/data
        Fluo:NXsubentry
            definition = "NXfluo"
            instrument:NXinstrument
                detector --> /entry/instrument/fluordet/data
                detector2 --> /entry/instrument/large_area/data
            data:NXdata
                @signal = detector
                @axes = [".", "."]
                detector --> /entry/instrument/fluordet/data
        data:NXdata --> /entry/Fluo/data
```

Rules for Special Cases

Scans

Scans are difficult to capture because they have great variety. Basically, any variable can be scanned. Such behaviour cannot be captured in application definitions. Therefore NeXus solves this difficulty with a set of rules. In this section, `NP` is used as a symbol for the number of scan points.

- The scan dimension `NP` is always the first dimension of any multi-dimensional dataset. The reason for this is that HDF allows the first dimension of a dataset to be unlimited. Which means, that data can be appended to the dataset during the scan.
- All data is stored as arrays of dimensions `NP`, original dimensions of the data at the appropriate position in the `NXentry` hierarchy.
- The `NXdata` group has to contain links to all variables varied during the scan and the detector data. Thus the `NXdata` group mimics the usual tabular representation of a scan.
- The `NXdata` group has attributes to enable the default plotting, as described in the section titled *NXdata Facilitates Automatic Plotting*.

Simple scan

Examples may be in order here. Let us start with a simple case, the sample is rotated around its rotation axis and data is collected in a single point detector. See figure *Simple Scan* for an overview. Then we have:

- A dataset at `NXentry/NXinstrument/NXdetector/data` of length `NP` containing the count data.
- A dataset at `NXentry/NXsample/rotation_angle` of length `NP` containing the positions of `rotation_angle` at the various steps of the scan.
- `NXdata` contains links to:
  - `NXentry/NXinstrument/NXdetector/data`
  - `NXentry/NXsample/rotation_angle`
- All other fields have their normal dimensions.

NeXus Simple Scan Example

```
entry:NXentry
    @default = data
    instrument:NXinstrument
        detector:NXdetector
            data[NP]
    sample:NXsample
        rotation_angle[NP]
    control:NXmonitor
        data[NP]
    data:NXdata
        @signal = "data"
        @axes = "rotation_angle"
        @rotation_angle_indices = 0
        data --> /entry/instrument/detector/data
        rotation_angle --> /entry/sample/rotation_angle
```

Simple scan with area detector

The next example is the same scan but with an area detector with `xsize` times `ysize` pixels. The only thing which changes is that `/NXentry/NXinstrument/NXdetector/data` will have the dimensions `NP, xsize, ysize`. See figure *Simple Scan with Area Detector* for an overview.

NeXus Simple Scan Example with Area Detector

```
entry:NXentry
    instrument:NXinstrument
        detector:NXdetector
            data:[NP,xsize,ysize]
    sample:NXsample
        rotation_angle[NP]
    control:NXmonitor
        data[NP]
    data:NXdata
        @signal = "data"
        @axes = ["rotation_angle", ".", "."]
        @rotation_angle_indices = 0
        data --> /entry/instrument/detector/data
        rotation_angle --> /entry/sample/rotation_angle
```

The `NXdata` group attribute `axes = rotation_angle . .` declares that only the first dimension of the plottable `data` has a dimension scale (by name, `rotation_angle`). The other two dimensions have no associated dimension scales and should be plotted against integer bin numbers.

Complex *hkl* scan

The next example involves a complex movement along the h axis in reciprocal space which requires mutiple motors of a four-circle diffractometer to be varied during the scan. We then have:

- A dataset at `NXentry/NXinstrument/NXdetector/data` of length `NP` containing the count data.
- A dataset at `NXentry/NXinstrument/NXdetector/polar_angle` of length `NP` containing the positions of the detector's polar_angle at the various steps of the scan.
- A dataset at `NXentry/NXsample/rotation_angle` of length `NP` containing the positions of `rotation_angle` at the various steps of the scan.
- A dataset at `NXentry/NXsample/chi` of length `NP` containing the positions of chi at the various steps of the scan.
- A dataset at `NXentry/NXsample/phi` of length `NP` containing the positions of phi at the various steps of the scan.
- A dataset at `NXentry/NXsample/h` of length `NP` containing the positions of the reciprocal coordinate h at the various steps of the scan.
- A dataset at `NXentry/NXsample/k` of length `NP` containing the positions of the reciprocal coordinate k at the various steps of the scan.
- A dataset at `NXentry/NXsample/l` of length `NP` containing the positions of the reciprocal coordinate l at the various steps of the scan.
- `NXdata` contains links to:
  - `NXentry/NXinstrument/NXdetector/data`
  - `NXentry/NXinstrument/NXdetector/polar_angle`
  - `NXentry/NXsample/rotation_angle`
  - `NXentry/NXsample/chi`
  - `NXentry/NXsample/phi`
  - `NXentry/NXsample/h`
  - `NXentry/NXsample/k`
  - `NXentry/NXsample/l`

  The `NXdata` also contains appropriate attributes as described in *Associating plottable data using attributes applied to the NXdata group*.
- All other fields have their normal dimensions.

NeXus Complex *hkl* Scan

```
entry:NXentry
    @default = data
    instrument:NXinstrument
        detector:NXdetector
            data[NP]
            polar_angle[NP]
            name
    sample:NXsample
```

(continues on next page)

(continued from previous page)

```
            name
            rotation_angle[NP]
            chi[NP]
            phi[NP]
            h[NP]
            k[NP]
            l[NP]
        control:NXmonitor
            data[NP]
        data:NXdata
            @signal = data
            @axes = "h"
            @h_indices = 0
            @k_indices = 0
            @l_indices = 0
            @chi_indices = 0
            @phi_indices = 0
            @polar_angle_indices = 0
            @rotation_angle_indices = 0
            data --> /entry/instrument/detector/data
            rotation_angle --> /entry/sample/rotation_angle
            chi --> /entry/sample/chi
            phi --> /entry/sample/phi
            polar_angle --> /entry/instrument/detector/polar_angle
            h --> /entry/sample/h
            k --> /entry/sample/k
            l --> /entry/sample/l
```

Multi-parameter scan: XAS

Data can be stored almost anywhere in the NeXus tree. While the previous examples showed data arrays in either `NXdetector` or `NXsample`, this example demonstrates that data can be stored in other places. Links are used to reference the data.

The example is for X-ray Absorption Spectroscopy (XAS) data where the monochromator energy is step-scanned and counts are read back from detectors before (`I0`) and after (`I`) the sample. These energy scans are repeated at a sequence of sample temperatures to map out, for example, a phase transition. While it is customary in XAS to plot *log(I0/I)*, we show them separately here in two different `NXdata` groups to demonstrate that such things are possible. Note that the length of the 1-D energy array is `NE` while the length of the 1-D temperature array is `NT`

NeXus Multi-parameter scan: XAS

```
entry:NXentry
    @default = "I_data"
    instrument:NXinstrument
        I:NXdetector
            data:NX_NUMBER[NE,NT]
            energy --> /entry/monochromator/energy
            temperature --> /entry/sample/temperature
        I0:NXdetector
```

(continues on next page)

(continued from previous page)

```
                data:NX_NUMBER[NE,NT]
                energy --> /entry/monochromator/energy
                temperature --> /entry/sample/temperature
        sample:NXsample
            temperature:NX_NUMBER[NT]
        monochromator:NXmonochromator
            energy:NX_NUMBER[NE]
        I_data:NXdata
            @signal = "data"
            @axes = ["energy", "temperature"]
            @energy_indices = 0
            @temperature_indices = 0
            data --> /entry/instrument/I/data
            energy --> /entry/monochromator/energy
            temperature --> /entry/sample/temperature
        I0_data:NXdata
            @signal = data
            @axes = ["energy", "temperature"]
            @energy_indices = 0
            @temperature_indices = 0
            data --> /entry/instrument/I00/data
            energy --> /entry/monochromator/energy
            temperature --> /entry/sample/temperature
```

Rastering

Rastering is the process of making experiments at various locations in the sample volume. Again, rasterisation experiments can be variable. Some people even raster on spirals! Rasterisation experiments are treated the same way as described above for scans. Just replace `NP` with `P`, the number of raster points.

Special rules apply if a rasterisation happens on a regular grid of size `xraster, yraster`. Then the variables varied in the rasterisation will be of dimensions `xraster, yraster` and the detector data of dimensions `xraster, yraster, (orginal dimensions)` of the detector. For example, an area detector of size `xsize, ysize` then it is stored with dimensions `xraster, yraster, xsize, ysize`.

> **Warning:** Be warned: if you use the 2D rasterisation method with `xraster, yraster` you may end up with invalid data if the scan is aborted prematurely. This cannot happen if the first method is used.

Streaming Data Acquisition And Logging

More and more data is collected in streaming mode. This means that time stamped data is logged for one or more inputs, possibly together with detector data. Another use case is the logging of parameters, for example temperature, while a long running data collection is in progress. NeXus covers this case too. There is one simple rule for structuring such files:

Just use the standard NeXus raw data file structure, but replace the corresponding data object with an NXlog or NXevent_data structure of the same name.

For example, consider your instrument is streaming detector images against a magnetic_field on the sample. In this case both NXsample/magnetic_field and NXdetector/data would become NXlog structures instead of simple arrays i.e.

the NXlog structure will have the same name as the NeXus field involved.

NXcollection

On demand from the community, NeXus introduced a more informal method of storing information in a NeXus file. This is the `NXcollection` class which can appear anywhere underneath `NXentry`. `NXcollection` is a container for holding other data. The foreseen use is to document collections of similar data which do not otherwise fit easily into the `NXinstrument` or `NXsample` hierarchy, such as the intent to record *all* motor positions on a synchrotron beamline. Thus, `NXcollection` serves as a quick point of access to data for an instrument scientist or another expert. NXcollection is also a feature for those who are too lazy to build up the complete NeXus hierarchy. An example usage case is documented in figure *NXcollection example*.

`NXcollection` Example

```
entry:NXentry
    positioners:NXcollection
        mxx:NXpositioner
        mzz:NXpositioner
        sgu:NXpositioner
        ttv:NXpositioner
        hugo:NXpositioner
        ....
    scalars:NXcollection
        title (dataset)
        lieselotte (dataset)
        ...
    detectors:NXcollection
        Pilatus:NXdata
        MXX-45:NXdata
        ....
```

Rules for Storing Data Items in NeXus Files

This section describes the rules which apply for storing single data items.

Naming Conventions

Group and field names used within NeXus follow a naming convention described by the following rules:

- The names of NeXus *group* and *field* items must only contain a restricted set of characters.

  This set is described by a regular expression syntax regular expression *regular expression syntax*, as described below.

- For the class names[1] of NeXus *group* items, the prefix *NX* is reserved as shown in the *table* below. Thus all NeXus class names start with NX. The chapter titled *NeXus: Reference Documentation* lists the available NeXus class names as either *base classes*, *application definitions*, or *contributed definitions*.

[1] The *class name* is the value assigned to the *NX_class* attribute of an HDF5 group in the NeXus data file. This *class name* is different than the *name* of the HDF5 group. This is important when not using the NAPI to either read or write the HDF5 data file.

NXDL group and field names

The names of NeXus *group* and *field* items are validated according to these boundaries:

- *Recommended* names[3]
  - lower case words separated by underscores and, if needed, with a trailing number
  - NOTE: this is used by the NeXus base classes
- *Allowed* names
  - any combination of upper and lower case letter, numbers, underscores and periods, except that periods cannot be at the start or end of the string
  - NOTE: this matches the *validItemName* regular expression *below*
- *Invalid* names
  - NOTE: does not match the *validItemName* regular expression *below*

Regular expression pattern for NXDL group and field names

The NIAC recognises that the majority of the world uses characters outside of the basic latin (a.k.a. US-ASCII, 7-bit ASCII) set currently included in the allowed names. The restriction given here reflects current technical issues and we expect to revisit the issue and relax such restrictions in future.

The names of NeXus *group* and *field* items must match this regular expression (named *validItemName* in the XML Schema file: *nxdl.xsd*):

```
^[a-zA-Z0-9_]([a-zA-Z0-9_.]*[a-zA-Z0-9_])?$
```

The length should be limited to no more than 63 characters (imposed by the HDF5 rules for names).

It is recognized that some facilities will construct data files with group and field names with upper case letters or start names with a number or include a period in a name.[Page 41, 3]

Use of underscore in descriptive names

Sometimes it is necessary to combine words in order to build a descriptive name for a field or a group. In such cases lowercase words are connected by underscores.

```
number_of_lenses
```

For all fields, only names from the NeXus base class dictionaries should be used. If a field name or even a complete component is missing, please suggest the addition to the *NIAC: The NeXus International Advisory Committee*. The addition will usually be accepted provided it is not a duplication of an existing field and adequately documented.

Note: The NeXus base classes provide a comprehensive dictionary of terms that can be used for each class. The expected spelling and definition of each term is specified in the base classes. It is not required to provide all the terms specified in a base class. Terms with other names are permitted but might not be recognized by standard software. Rather than persist in using names not specified in the standard, please suggest additions to the *NIAC: The NeXus International Advisory Committee*.

[3] NeXus data files with group or field names that match the regular expression but contain upper case characters, start with a digit, or include a period in the group or field names might not be accepted by all software that reads NeXus data files. These names will be flagged as a warning during data file validation.

The data stored in NeXus fields must be *readback* values. This means values as read from the detector, other hardware, etc. There are occasions where it is sensible to store the target value the variable was supposed to have. In such cases, the *target* value is stored with a name built by appending `_set` to the NeXus (readback) field name.

Consider this example:

```
temperature
temperature_set
```

The `temperature` field will hold the readback from the cryostat/furnace/whatever. The field `temperature_set` will hold the target value for the temperature as set by the experiment control software.

Some fields share a common part of their name and an additional part name that makes the whole name specific. For example, a `unit_cell` might have parts named `abc`, `alphabetagamma`, and `volume`. It is recommended to write them with the common part first, an underscore (_), and then the specific part. In this way, the fields will sort alphabetically on the common name. So, in this example:

```
unit_cell_abc
unit_cell_alphabetagamma
unit_cell_volume
```

Reserved prefixes

When naming an attribute, field, or group, NeXus has reserved certain prefixes to the names to ensure that names written in NeXus files will not conflict with future releases as the NeXus standard evolves. Prefixes should follow a naming scheme of uppercase letters followed by an underscore, but exceptions will be made for cases already in wide use. The following table lists the prefixes reserved by NeXus.

prefix	use	meaning	URL
BLUESKY_	attributes	reserved for use by Bluesky project	https://blueskyproject.io
DECTRIS_	attributes, fields	reserved for use by Dectris	https://www.dectris.com
IDF_	attributes	reserved for use by pulsedTD Muon definition	https://www.isis.stfc.ac.uk/Pages/nexus-definition-v27924.pdf
NDAttr	attributes	reserved for use by EPICS area detector	https://github.com/areaDetector
NX	NXDL class	for the class names used with NeXus groups	https://www.nexusformat.org
NX_	attributes	reserved for use by NeXus	https://www.nexusformat.org
PDBX_	attributes	reserved for the US protein data bank	https://www.rcsb.org
SAS_	attributes	reserved for use by canSAS	https://www.cansas.org
SILX_	attributes	reserved for use by silx	https://www.silx.org

Reserved suffixes

When naming a field, NeXus has reserved certain suffixes to the names so that a specific meaning may be attached. Consider a field named DATASET, the following table lists the suffixes reserved by NeXus.

suffix	reference	meaning
_end	NXtransformations	end points of the motions that start with DATASET
_errors	NXdata	uncertainties (a.k.a., errors)
_increment_set	NXtransformations	intended average range through which the corresponding axis moves during the exposure of a frame
_indices	NXdata	Integer array that defines the indices of the signal field which need to be used in the DATASET in order to reference the corresponding axis value
_mask		Field containing a signal mask, where 0 means the pixel is not masked. If required, bit masks are defined in NXdetector pixel_mask.
_set	*target values*	Target value of DATASET
_weights		divide DATASET by these weights[4]

Variants

Sometimes it is necessary to store alternate values of a NeXus field in a NeXus file. A common example may be the beam center of which a rough value is available at data acquisition. But later on, a better beam center is calculated as part of the data reduction. In order to store this without losing the historical information, the original field can be given a variant attribute that points to a new field containing the obsolete value. If even better values become available, further fields can be inserted into the chain of variant attributes pointing to the preceeding value for the field. A reader can thus keep the best value in the pre-defined field, and also be able to follow the variant chain and locate older variants.

A little example is in order to illustrate the scheme:

```
beam_center_x
        @variant=beam_center_x_refined
beam_center_x_refined
        @variant=beam_center_x_initial_guess
beam_center_x_initial_guess
```

NeXus borrowed this scheme from CIF. In this way all the different variants of a field can be preserved. The expectation is that variants will be rarely used and NXprocess groups with the results of data reduction will be written instead.

Uncertainties or Errors

It is desirable to store experimental errors (also known as *uncertainties*) together with the data. NeXus supports this through a convention: uncertainties or experimental errors on data are stored in a separate field which has a name consisting of the original name of the data with _errors appended to it. These uncertainties fields have the same shape as the original data field.

An example, from NXdetector:

[4] If DATASET_weights exists and has the same shape as the field, you are supposed to divide DATASET by the weights.

```
data
data_errors
beam_center_x
beam_center_x_errors
```

Where data errors would contain the errors on data, and beam_center_x_errors the error on the beam center for x.

NeXus Array Storage Order

NeXus stores multi-dimensional arrays of physical values in C language storage order, where the first dimension has the slowest varying index when iterating through the array in storage order, and the last dimension is the fastest varying. This is the rule. *Good reasons are required to deviate from this rule.*

Where the array contains data from a detector, the array dimensions may correspond to physical directions or axes. The slowest, slow, fast, fastest qualifiers can then apply to these axes too.

It is possible to store data in storage orders other than C language order.

As well it is possible to specify that the data needs to be converted first before being useful. Consider one situation, when data must be streamed to disk as fast as possible and conversion to C language storage order causes unnecessary latency. This case presents a good reason to make an exception to the standard rule.

Non C Storage Order

In order to indicate that the storage order is different from C storage order two additional data set attributes, offset and stride, have to be stored which together define the storage layout of the data. Offset and stride contain rank numbers according to the rank of the multidimensional data set. Offset describes the step to make when the dimension is multiplied by 1. Stride defines the step to make when incrementing the dimension. This is best explained by some examples.

Offset and Stride for 1 D data:

```
   * raw data = 0 1 2 3 4 5 6 7 8 9
      size[1] = { 10 }  // assume uniform overall array dimensions

   * default stride:
      stride[1] = { 1 }
      offset[1] = { 0 }
      for i:
         result[i]:
            0 1 2 3 4 5 6 7 8 9

   * reverse stride:
      stride[1] = { -1 }
      offset[1] = { 9 }
      for i:
         result[i]:
            9 8 7 6 5 4 3 2 1 0
```

Offset and Stride for 2D Data

```
   * raw data = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
      size[2] = { 4, 5 }  // assume uniform overall array dimensions

   * row major (C) stride:
      stride[2] = { 5, 1 }
      offset[2] = { 0, 0 }
      for i:
         for j:
            result[i][j]:
               0 1 2 3 4
               5 6 7 8 9
               10 11 12 13 14
               15 16 17 18 19

   * column major (Fortran) stride:
      stride[2] = { 1, 4 }
      offset[2] = { 0, 0 }
      for i:
         for j:
            result[i][j]:
               0 4 8 12 16
               1 5 9 13 17
               2 6 10 14 18
               3 7 11 15 19

   * "crazy reverse" row major (C) stride:
      stride[2] = { -5, -1 }
      offset[2] = { 4, 5 }
      for i:
         for j:
            result[i][j]:
               19 18 17 16 15
               14 13 12 11 10
               9 8 7 6 5
               4 3 2 1 0
```

Offset and Stride for 3D Data

```
  * raw data = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
         20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
         40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
      size[3] = { 3, 4, 5 }  // assume uniform overall array dimensions

   * row major (C) stride:
      stride[3] = { 20, 5, 1 }
      offset[3] = { 0, 0, 0 }
      for i:
         for j:
            for k:
```

(continues on next page)

(continued from previous page)

```
               result[i][j][k]:
                  0 1 2 3 4
                  5 6 7 8 9
                  10 11 12 13 14
                  15 16 17 18 19

                  20 21 22 23 24
                  25 26 27 28 29
                  30 31 32 33 34
                  35 36 37 38 39

                  40 41 42 43 44
                  45 46 47 48 49
                  50 51 52 53 54
                  55 56 57 58 59

   * column major (Fortran) stride:
      stride[3] = { 1, 3, 12 }
      offset[3] = { 0, 0, 0 }
      for i:
         for j:
            for k:
               result[i][j][k]:
                  0 12 24 36 48
                  3 15 27 39 51
                  6 18 30 42 54
                  9 21 33 45 57

                  1 13 25 37 49
                  4 16 28 40 52
                  7 19 31 43 55
                  10 22 34 46 58

                  2 14 26 38 50
                  5 17 29 41 53
                  8 20 32 44 56
                  11 23 35 47 59
```

NeXus Data Types

description	matching regular expression
integer	`NX_INT(8|16|32|64)`
floating-point	`NX_FLOAT(32|64)`
array	`(\\[0-9\\])?`
valid item name	`^[a-zA-Z0-9_]([a-zA-Z0-9_.]*[a-zA-Z0-9_])?$`
valid class name	`^NX[A-Za-z0-9_]*$`

NeXus supports numeric data as either integer or floating-point numbers. A number follows that indicates the number of bits in the word. The table above shows the regular expressions that match the data type specifier.

integers
: NX_INT8, NX_INT16, NX_INT32, or NX_INT64

floating-point numbers
: NX_FLOAT32 or NX_FLOAT64

date / time stamps
: NX_DATE_TIME or ISO8601: Dates and times are specified using ISO-8601 standard definitions. Refer to *NeXus dates and times*.

strings
: NX_CHAR: The preferred string representation is UTF-8. Both fixed-length strings and variable-length strings are valid. String arrays cannot be used where only a string is expected (title, start_time, end_time, NX_class attribute,...). Fields or attributes requiring the use of string arrays will be clearly marked as such (like the NXdata attribute auxiliary_signals).

binary data
: Binary data is to be written as UINT8.

images
: Binary image data is to be written using UINT8, the same as binary data, but with an accompanying image mime-type. If the data is text, the line terminator is [CR][LF].

NeXus dates and times

NeXus dates and times should be stored using the ISO 8601[5] format, e.g. 1996-07-31T21:15:22+0600 (which includes a time zone offset of +0600). Note: The time zone offset is always numeric or Z (which means UTC). The standard also allows for time intervals in fractional seconds with *1 or more digits of precision*. This avoids confusion, e.g. between U.S. and European conventions, and is appropriate for machine sorting. It is recommended to add an explicit time zone, otherwise the local time zone is assumed per ISO8601. The norm is that if there is no time zone, it is assumed local time, however, when a file moves from one country to another it is undefined. If the local time zone is written, the ambiguity is gone.

strftime() format specifiers for ISO-8601 time

```
%Y-%m-%dT%H:%M:%S%z
```

Note: Note that the T appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601. It is common to use a space in place of the T, such as 1996-07-31 21:15:22+0600. While human-readable (and later allowed in a relaxed revision of the standard), compatibility with libraries supporting the ISO 8601 standard is not assured with this substitution. The strftime() format specifier for this is "%Y-%m-%d %H:%M:%S%z".

[5] ISO 8601: https://www.w3.org/TR/NOTE-datetime

NeXus Data Units

Given the plethora of possible applications of NeXus, it is difficult to define units to use. Therefore, the general rule is that you are free to store data in any unit you find fit. However, any field must have a units attribute which describes the units. Wherever possible, SI units are preferred. NeXus units are written as a string attribute (`NX_CHAR`) and describe the engineering units. The string should be appropriate for the value. Values for the NeXus units must be specified in a format compatible with Unidata UDunits[6] Application definitions may specify units to be used for fields using an `enumeration`.

Storing Detectors

There are very different types of detectors out there. Storing their data can be a challenge. As a general guide line: if the detector has some well defined form, this should be reflected in the data file. A linear detector becomes a linear array, a rectangular detector becomes an array of size `xsize` times `ysize`. Some detectors are so irregular that this does not work. Then the detector data is stored as a linear array, with the index being detector number till `ndet`. Such detectors must be accompanied by further arrays of length `ndet` which give `azimuthal_angle, polar_angle and distance` for each detector.

If data from a time of flight (TOF) instrument must be described, then the TOF dimension becomes the last dimension, for example an area detector of `xsize` *vs.* `ysize` is stored with TOF as an array with dimensions `xsize, ysize, ntof`.

Monitors are Special

Monitors, detectors that measure the properties of the experimental probe rather than the probe's interaction with the sample, have a special place in NeXus files. Monitors are crucial to normalize data. To emphasize their role, monitors are not stored in the `NXinstrument` hierarchy but on `NXentry` level in their own groups as there might be multiple monitors. Of special importance is the monitor in a group called `control`. This is the main monitor against which the data has to be normalized. This group also contains the counting control information, i.e. counting mode, times, etc.

Monitor data may be multidimensional. Good examples are scan monitors where a monitor value per scan point is expected or time-of-flight monitors.

Find the plottable data

Simple plotting is one of the motivations for the NeXus standard. To implement *simple plotting*, a mechanism must exist to identify the default data for visualization (plotting) in any NeXus data file. Over its history the NIAC has agreed upon a method of applying metadata to identify the default plottable data. This metadata has always been specified as HDF attributes. With the evolution of the underlying file formats and the NeXus data standard, the method to identify the default plottable data has evolved, undergoing three distinct versions.

version 1
Associating plottable data by dimension number using the axis attribute

version 2
Associating plottable data by name using the axes attribute

version 3
Associating plottable data using attributes applied to the NXdata group

[6] The UDunits specification also includes instructions for derived units. At present, the contents of NeXus `units` attributes are not validated in data files.

Consult the *NeXus API* section, which describes the routines available to program these operations. In the course of time, generic NeXus browsers will provide this functionality automatically.

For programmers who may encounter NeXus data files written using any of these methods, we present the algorithm for each method to find the default plottable data. It is recommended to start with the most recent method, *Version 3*, first.

Version 3

The third (current) method to identify the default plottable data is as follows:

1. Start at the top level of the NeXus data file (the *root* of the HDF5 hierarchy).
2. Pick the default NXentry group.

   If the *root* has an attribute `default`, the attribute's value is the name of the `NXentry` group to be used. (The value of the `default` attribute *names* an existing child of this group. The child group must itself be a NeXus group.) If no *default* attribute exists, pick any `NXentry` group. This is trivial if there is only one `NXentry` group.

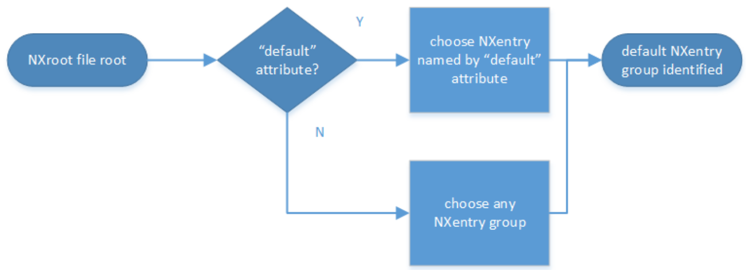


Fig. 6: Find plottable data: select the `NXentry` group

3. Pick the default NXdata group.

   Open the `NXentry` group selected above. If it has an attribute `default`, the attribute's value is the name of the `NXdata` group to be used. (The value of the `default` attribute *names* an existing child of this group. The child group must itself be a NeXus group.) If no *default* attribute exists, pick any `NXdata` group. This is trivial if there is only one `NXdata` group.

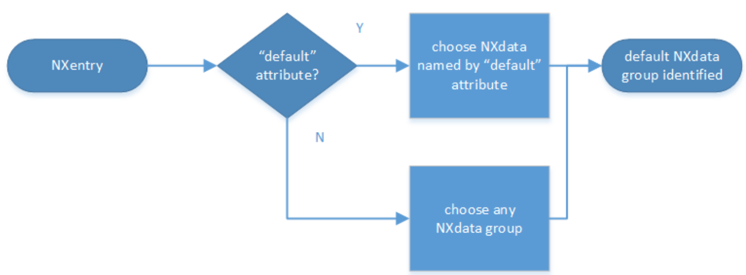


Fig. 7: Find plottable data: select the `NXdata` group

1. Pick the default plottable field (the *signal* data).

   Open the `NXdata` group selected above. If it has a `signal` attribute, the attribute's value is the name of the field to be plotted. (The value of the `signal` attribute *names* an existing child of this group. The child group must itself be a NeXus field.) If no `signal` attribute is present on the `NXdata` group, then proceed to try an *older NeXus method* to find the default plottable data.

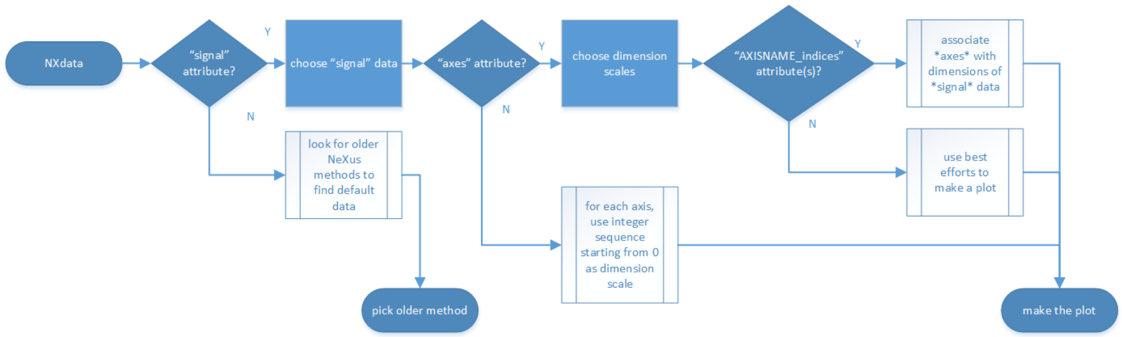


Fig. 8: Find plottable data: select the *signal* data

1. Pick the fields with the dimension scales (the *axes*).

   If the same `NXdata` group has an attribute `axes`, then its value is a string (*signal* data is 1-D) or string array (*signal* data is 2-D or higher rank) naming the field **in this group** to be used as dimension scales of the default plottable data. The number of values given must be equal to the *rank* of the *signal* data. These are the *abscissae* of the plottable *signal* data.

   If no field is available to provide a dimension scale for a given dimension, then a "`.`" will be used in that position. In such cases, programmers are expected to use an integer sequence starting from 0 for each position along that dimension.

2. Associate the dimension scales with each dimension of the plottable data.

   For each field (its name is *AXISNAME*) in `axes` that provides a dimension scale, there will be an `NXdata` group attribute `AXISNAME_indices` which value is an .. integer or integer array with value of the dimensions of the *signal* data to which this dimension scale applies.

   If no `AXISNAME_indices` attribute is provided, a programmer is encouraged to make best efforts assuming the intent of this `NXdata` group to provide a default plot. The `AXISNAME_indices` attribute is only required when necessary to resolve ambiguity.

   It is possible there may be more than one `AXISNAME_indices` attribute with the same value or values. This indicates the possibilty of using alternate abscissae along this (these) dimension(s). The field named in the `axes` attribute indicates the intention of the data file writer as to which field should be used by default.

2. Plot the *signal* data, given *axes* and *AXISNAME_indices*.

When all the `default` and `signal` attributes are present, this Python code example will identify directly the default plottable data (assuming a `plot()` function has been defined by some code:

```python
group = h5py.File(hdf5_file_name, "r")

while "default" in group.attrs:
    child_group_name = group.attrs["default"]
    group = group[child_group_name]

# assumes group.attrs["NX_class"] == "NXdata"
signal_field_name = group.attrs["signal"]
data = group[signal_field_name]

plot(data)
```

Version 2

Tip: Try this method for older NeXus data files and *Version 3* fails..

The second method to identify the default plottable data is as follows:

1. Start at the top level of the NeXus data file.
2. Loop through the groups with class `NXentry` until the next step succeeds.

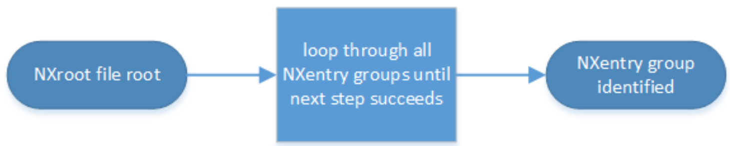


Fig. 9: Find plottable data: pick a `NXentry` group

3. Open the NXentry group and loop through the subgroups with class `NXdata` until the next step succeeds.

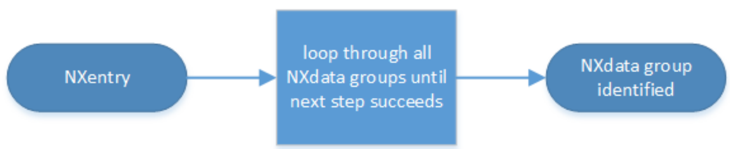


Fig. 10: Find plottable data: pick a `NXdata` group

4. Open the NXdata group and loop through the fields for the one field with attribute `signal="1"`. Note: There should be *only one* field that matches.

   This is the default plottable data.

   If there is no such `signal="1"` field, proceed to try an *older NeXus method* to find the default plottable data.

   1. If this field has an attribute `axes`:
      1. The `axes` attribute value contains a colon (or comma) delimited list (in the C-order of the data array) with the names of the dimension scales associated with the plottable data. Such as: `axes="polar_angle:time_of_flight"`
      2. Parse `axes` and open the fields to describe your dimension scales
   2. If this field has no attribute `axes`:
      1. Search for fields with attributes `axis=1`, `axis=2`, etc.
      2. These are the fields describing your axis. There may be several fields for any axis, i.e. there may be multiple fields with the attribute `axis=1`. Among them the field with the attribute `primary=1` is the preferred one. All others are alternative dimension scales.

5. Having found the default plottable data and its dimension scales: make the plot.

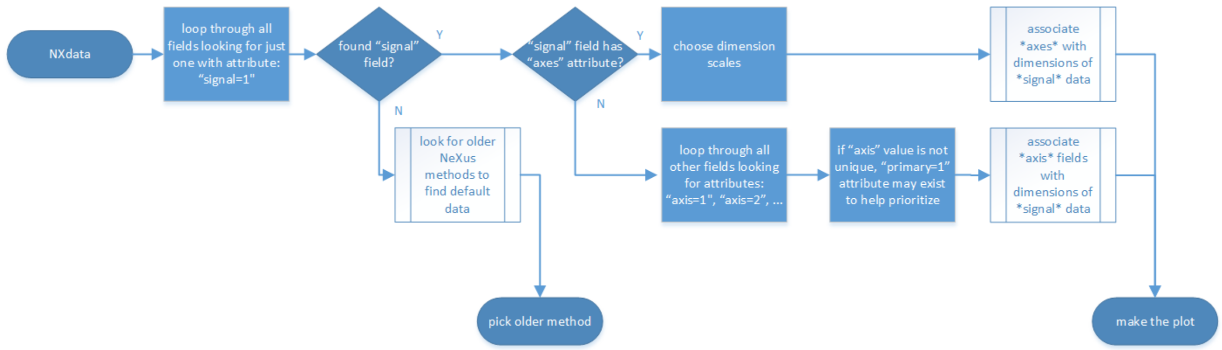


Fig. 11: Find plottable data: select the *signal* data

Version 1

Tip: Try this method for older NeXus data files.

The first method to identify the default plottable data is as follows:

1. Open the first top level NeXus group with class `NXentry`.

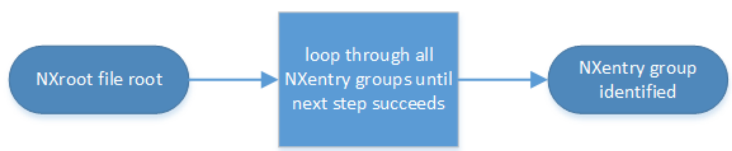


Fig. 12: Find plottable data: pick the first `NXentry` group

2. Open the first NeXus group with class `NXdata`.

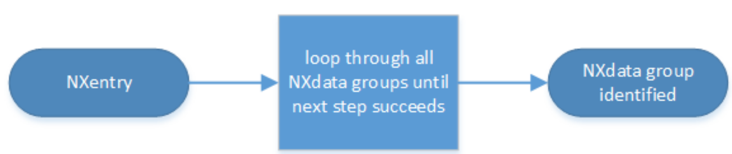


Fig. 13: Find plottable data: pick the first `NXdata` group

3. Loop through NeXus fields in this group searching for the item with attribute `signal="1"` indicating this field has the plottable data.
4. Search for the one-dimensional NeXus fields with attribute `primary=1`. These are the dimension scales to label the axes of each dimension of the data.
5. Link each dimension scale to the respective data dimension by the `axis` attribute (`axis=1`, `axis=2`, … up to the rank of the data).
6. If necessary, close this `NXdata` group, search the next `NXdata` group, repeating steps 3 to 5.
7. If necessary, close the `NXentry` group, search the next `NXentry` group, repeating steps 2 to 6.

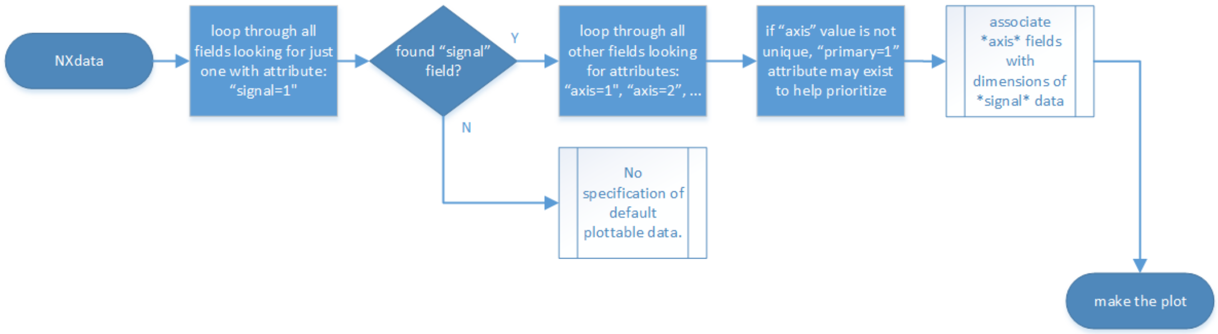


Fig. 14: Find plottable data: select the *signal* data

Associating Multi Dimensional Data with Axis Data

NeXus allows for storage of multi dimensional arrays of data. It is this data that presents the most challenge for description. In most cases it is not sufficient to just have the indices into the array as a label for the dimensions of the data. Usually the information which physical value corresponds to an index into a dimension of the multi dimensional data set. To this purpose a means is needed to locate appropriate data arrays which describe what each dimension of a multi dimensional data set actually corresponds too. There is a standard HDF facility to do this: it is called dimension scales. Unfortunately, when NeXus was first designed, there was only one global namespace for dimension scales. Thus NeXus had to devise its own scheme for locating axis data which is described here. A side effect of the NeXus scheme is that it is possible to have multiple mappings of a given dimension to physical data. For example, a TOF data set can have the TOF dimension as raw TOF or as energy.

There are now three methods of associating each data dimension to its respective dimension scale. Only the first method is recommended now, the other two (older methods) are now discouraged.

1. *Associating plottable data using attributes applied to the NXdata group*
2. *Associating plottable data by name using the axes attribute*
3. *Associating plottable data by dimension number using the axis attribute*

The recommended method uses the `axes` attribute applied to the NXdata group to specify the names of each dimension scale. A prerequisite is that the fields describing the axes of the plottable data are stored together with the plottable data in the same NeXus group. If this leads to data duplication, use *links*.

Associating plottable data using attributes applied to the NXdata group

---

Tip: Recommended: This is the "*NIAC2014*" method recommended for all new NeXus data files.

The default data to be plotted (and any associated axes) is specified using attributes attached to the NXdata group.

`signal`
: Defines the name of the default field *in the NXdata group*. A field of this name *must* exist (either as field or link to field).

  It is recommended to use this attribute rather than adding a signal attribute to the field.[7] The procedure

[7] Summary of the discussion at NIAC2014 to revise how to find default data: https://www.nexusformat.org/2014_How_to_find_default_data.html

to identify the default data to be plotted is quite simple. Given any NeXus data file, any `NXentry`, or any `NXdata`, follow the chain as it is described from that point. Specifically:

- The root of the NeXus file may have a `default` attribute that names the default NXentry group. This attribute may be omitted if there is only one NXentry group. If a second NXentry group is later added, the `default` attribute must be added then.
- Every NXentry group may have a `default` attribute that names the default NXdata group. This attribute may be omitted if there is only one NXdata group or if no NXdata is present. If a second NXdata group is later added, the `default` attribute must be added then.
- Every NXdata group will have a `signal` attribute that names the field name to be plotted by default. This attribute is required.

axes

String array[8] that defines the independent data fields used in the default plot for all of the dimensions of the *signal* field. One entry is provided for every dimension in the *signal* field.

The field(s) named as values (known as "axes") of this attribute *must* exist. An axis slice is specified using a field named `AXISNAME_indices` as described below (where the text shown here as `AXISNAME` is to be replaced by the actual field name).

When no default axis is available for a particular dimension of the plottable data, use a "." in that position.

See examples provided on the NeXus webpage ([9]).

If there are no axes at all (such as with a stack of images), the axes attribute can be omitted.

AXISNAME_indices

Each `AXISNAME_indices` attribute indicates the dependency relationship of the `AXISNAME` field (where `AXISNAME` is the name of a field that exists in this `NXdata` group) with one or more dimensions of the plottable data.

Integer array[Page 54, 8] that defines the indices of the *signal* field (that field will be a multidimensional array) which need to be used in the `AXISNAME` field in order to reference the corresponding axis value.

The first index of an array is `0` (zero).

Here, *AXISNAME* is to be replaced by the name of each field described in the `axes` attribute. An example with 2-D data, $d(t, P)$, will illustrate:

```
data_2d:NXdata
    @signal="data"
    @axes=["time","pressure"]
    @time_indices=0
    @pressure_indices=1
    data: float[1000,20]
    time: float[1000]
    pressure: float[20]
```

This attribute is to be provided in all situations. However, if the indices attributes are missing (such as for data files written before this specification), file readers are encouraged to make their best efforts to plot the data. Thus the implementation of the `AXISNAME_indices` attribute is based on the model of "strict writer, liberal reader".

[8] Note on array attributes: Attributes potentially containing multiple values (axes and _indices) are to be written as string or integer arrays, to avoid string parsing in reading applications.

[9] NIAC2014 proposition: https://www.nexusformat.org/2014_axes_and_uncertainties.html

Examples

Several examples are provided to illustrate this method. More examples are available in the NeXus webpage ([9]).

simple 1-D data example showing how to identify the default data (*counts* vs. *mr*)

In the first example, storage of a 1-D data set (*counts* vs. *mr*) is described.

```
datafile.hdf5:NeXus data file
  @default="entry"
  entry:NXentry
    @default="data"
    data:NXdata
      @signal="counts"
      @axes="mr"
      @mr_indices=0
      counts: float[100]  --> the default dependent data
      mr: float[100]      --> the default independent data
```

2-D data example showing how to identify the default data and associated dimension scales

A 2-D data set, *data* as a function of *time* and *pressure* is described. By default as indicated by the `axes` attribute, *pressure* is to be used. The *temperature* array is described as a substitute for *pressure* (so it replaces dimension `1` of `data` as indicated by the `temperature_indices` attribute).

```
datafile.hdf5:NeXus data file
  @default="entry"
  entry:NXentry
    @default="data_2d"
    data_2d:NXdata
      @signal="data"
      @axes=["time","pressure"]
      @pressure_indices=1
      @temperature_indices=1
      @time_indices=0
      data: float[1000,20]
      pressure: float[20]
      temperature: float[20]
      time: float[1000]
```

Associating plottable data by name using the `axes` attribute

Warning: Discouraged: See this method: *Associating plottable data using attributes applied to the NXdata group*.

This method defines an attribute of the data field called *axes*. The `axes` attribute contains the names of each dimension scale as a colon (or comma) separated list in the order they appear in C. For example:

denoting axes by name

```
data:NXdata
  time_of_flight = 1500.0 1502.0 1504.0 ...
  polar_angle = 15.0 15.6 16.2 ...
  some_other_angle = 0.0 0.0 2.0 ...
  data = 5 7 14 ...
    @axes = ["polar_angle", "time_of_flight"]
    @signal = 1
```

Associating plottable data by dimension number using the `axis` attribute

> **Warning:** Discouraged: See this method: *Associating plottable data by name using the axes attribute*

The original method defines an attribute of each dimension scale field called *axis*. It is an integer whose value is the number of the dimension, in order of fastest varying dimension. That is, if the array being stored is data with elements `data[j][i]` in C and `data(i,j)` in Fortran, where `i` is the time-of-flight index and `j` is the polar angle index, the `NXdata` group would contain:

denoting axes by integer number

```
data:NXdata
  time_of_flight = 1500.0 1502.0 1504.0 ...
    @axis = 1
    @primary = 1
  polar_angle = 15.0 15.6 16.2 ...
    @axis = 2
    @primary = 1
  some_other_angle = 0.0 0.0 2.0 ...
    @axis = 1
  data = 5 7 14 ...
    @signal = 1
```

The `axis` attribute must be defined for each dimension scale. The `primary` attribute is unique to this method.

There are limited circumstances in which more than one dimension scale for the same data dimension can be included in the same `NXdata` group. The most common is when the dimension scales are the three components of an *(hkl)* scan. In order to handle this case, we have defined another attribute of type integer called `primary` whose value determines the order in which the scale is expected to be chosen for plotting, i.e.

- 1st choice: `primary=1`
- 2nd choice: `primary=2`
- etc.

If there is more than one scale with the same value of the `axis` attribute, one of them must have set `primary=1`. Defining the `primary` attribute for the other scales is optional.

Note:

The `primary` attribute can only be
: used with the first method of defining

dimension scales
: discussed above. In addition to the `signal` data, this group could contain a data set of the same rank and dimensions called `errors` containing the standard deviations of the data.

Physical File format

This section describes how NeXus structures are mapped to features of the underlying physical file format. This is a guide for people who wish to create NeXus files without using the NeXus-API.

Choice of HDF as Underlying File Format

At its beginnings, the founders of NeXus identified the Hierarchical Data Format (HDF) as a capable and efficient multi-platform data storage format. HDF was designed for large data sets and already had a substantial user community. HDF was developed and maintained initially by the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC) and later spun off into its own group called The HDF Group (THG: http://www.hdfgroup.org/). Rather then developing its own unique physical file format, the NeXus group choose to build NeXus on top of HDF.

HDF (now HDF5) is provided with software to read and write data (this is the application-programmer interface, or API) using a large number of computing systems in common use for neutron and X-ray science. HDF is a binary data file format that supports compression and structured data.

Mapping NeXus into HDF

NeXus data structures map directly to HDF structures. NeXus *groups* are HDF5 *groups* and NeXus *fields* (or data sets) are HDF5 *datasets*. Attributes map directly to HDF group or dataset attributes. The NeXus class is stored as an attribute to the HDF5 group with the name `NX_class` with value of the NeXus class name. (For legacy NeXus data files using HDF4, groups are HDF4 *vgroups* and fields are HDF4 *SDS (scientific data sets)*. HDF4 does not support group attributes. HDF4 supports a group class which is set with the `Vsetclass()` call and read with `VGetclass()`.)

A NeXus `link` directly maps to the HDF hard link mechanisms.

Note: **Examples** are provided in the *Examples of writing and reading NeXus data files* chapter. These examples include software to write and read NeXus data files using the NAPI, as well as other software examples that use native (non-NAPI) libraries. In some cases the examples show the content of the NeXus data files that are produced. Here are links to some of the examples:

- *How do I write a NeXus file?*
- *How do I read a NeXus file?*
- ***HDF5 in C with NAPI***
  - *HDF5 in Python with NAPI*
- *Writing a simple NeXus file using native HDF5 commands in C*
- *Reading a simple NeXus file using native HDF5 commands in C*
- *Write a NeXus HDF5 File*
- *Read a NeXus HDF5 File*

Perhaps the easiest way to view the implementation of NeXus in HDF5 is to look at the data structure. For this, we use the `h5dump` command-line utility provided with the HDF5 support libraries. Short examples are provided for the basic NeXus data components:

- *group*: created in C NAPI by:

```
NXmakegroup (fileID, "entry", "NXentry");
```

- *field*: created in C NAPI by:

```
NXmakedata (fileID, "two_theta", NX_FLOAT32, 1, &n);
  NXopendata (fileID, "two_theta");
NXputdata (fileID, tth);
```

- *attribute*: created in C NAPI by:

```
NXputattr (fileID, "units", "degrees", 7, NX_CHAR);
```

- *link* created in C NAPI by:

```
NXmakelink (fileid, &itemid);
# -or-
NXmakenamedlink (fileid, "linked_name", &itemid);
```

`h5dump` of a NeXus `NXentry` group

```
GROUP "entry" {
  ATTRIBUTE "NX_class" {
     DATATYPE  H5T_STRING {
           STRSIZE 7;
           STRPAD H5T_STR_NULLPAD;
           CSET H5T_CSET_ASCII;
           CTYPE H5T_C_S1;
        }
     DATASPACE  SCALAR
     DATA {
     (0): "NXentry"
     }
  }
  # ... group contents
}
```

h5dump of a NeXus field (HDF5 dataset)

```
DATASET "two_theta" {
   DATATYPE  H5T_IEEE_F64LE
   DATASPACE  SIMPLE { ( 31 ) / ( 31 ) }
   DATA {
   (0): 17.9261, 17.9259, 17.9258, 17.9256, 17.9254, 17.9252,
   (6): 17.9251, 17.9249, 17.9247, 17.9246, 17.9244, 17.9243,
   (12): 17.9241, 17.9239, 17.9237, 17.9236, 17.9234, 17.9232,
   (18): 17.9231, 17.9229, 17.9228, 17.9226, 17.9224, 17.9222,
   (24): 17.9221, 17.9219, 17.9217, 17.9216, 17.9214, 17.9213,
   (30): 17.9211
   }
   ATTRIBUTE "units" {
      DATATYPE  H5T_STRING {
            STRSIZE 7;
            STRPAD H5T_STR_NULLPAD;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SCALAR
      DATA {
      (0): "degrees"
      }
   }
   # ... other attributes
}
```

h5dump of a NeXus attribute

```
ATTRIBUTE "axes" {
   DATATYPE  H5T_STRING {
         STRSIZE 9;
         STRPAD H5T_STR_NULLPAD;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
   DATASPACE  SCALAR
   DATA {
   (0): "two_theta"
   }
}
```

h5dump of a NeXus link

```
# NeXus links have two parts in HDF5 files.

# The dataset is created in some group.
# A "target" attribute is added to indicate the HDF5 path to this dataset.

ATTRIBUTE "target" {
   DATATYPE  H5T_STRING {
         STRSIZE 21;
         STRPAD H5T_STR_NULLPAD;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
   DATASPACE  SCALAR
   DATA {
   (0): "/entry/data/two_theta"
   }
}

# then, the hard link is created that refers to the original dataset
# (Since the name is "two_theta" in this example, it is understood that
# this link is created in a different HDF5 group than "/entry/data".)

DATASET "two_theta" {
   HARDLINK "/entry/data/two_theta"
}
```

1.3 Constructing NeXus Files and Application Definitions

In *NeXus Design*, we discussed the design of the NeXus format in general terms. In this section a more tutorial style introduction in how to construct a NeXus file is given. As an example a hypothetical instrument named WONI will be used.

Note: If you are looking for a tutorial on reading or writing NeXus data files using the NeXus API, consult the *NAPI: NeXus Application Programmer Interface (frozen)* chapter. For code examples (with or without NAPI), refer to the *Code Examples in Various Languages* chapter.

1.3.1 The WOnderful New Instrument (WONI)

Consider yourself to be responsible for some hypothetical WOnderful New Instrument (WONI). You are tasked to ensure that WONI will record data according to the NeXus standard. For the sake of simplicity, WONI bears a strong resemblance to a simple powder diffractometer, but let's pretend that WONI cannot use any of the existing NXDL application definitions.

WONI uses collimators and a monochromator to illuminate the sample with neutrons of a selected wavelength as described in *The (fictional) WONI example powder diffractometer*. The diffracted beam is collected in a large, banana-shaped, position sensitive detector. Typical data looks like *Example Powder Diffraction Plot from (fictional) WONI at HYNES*. There is a generous background to the data plus quite a number of diffraction peaks.

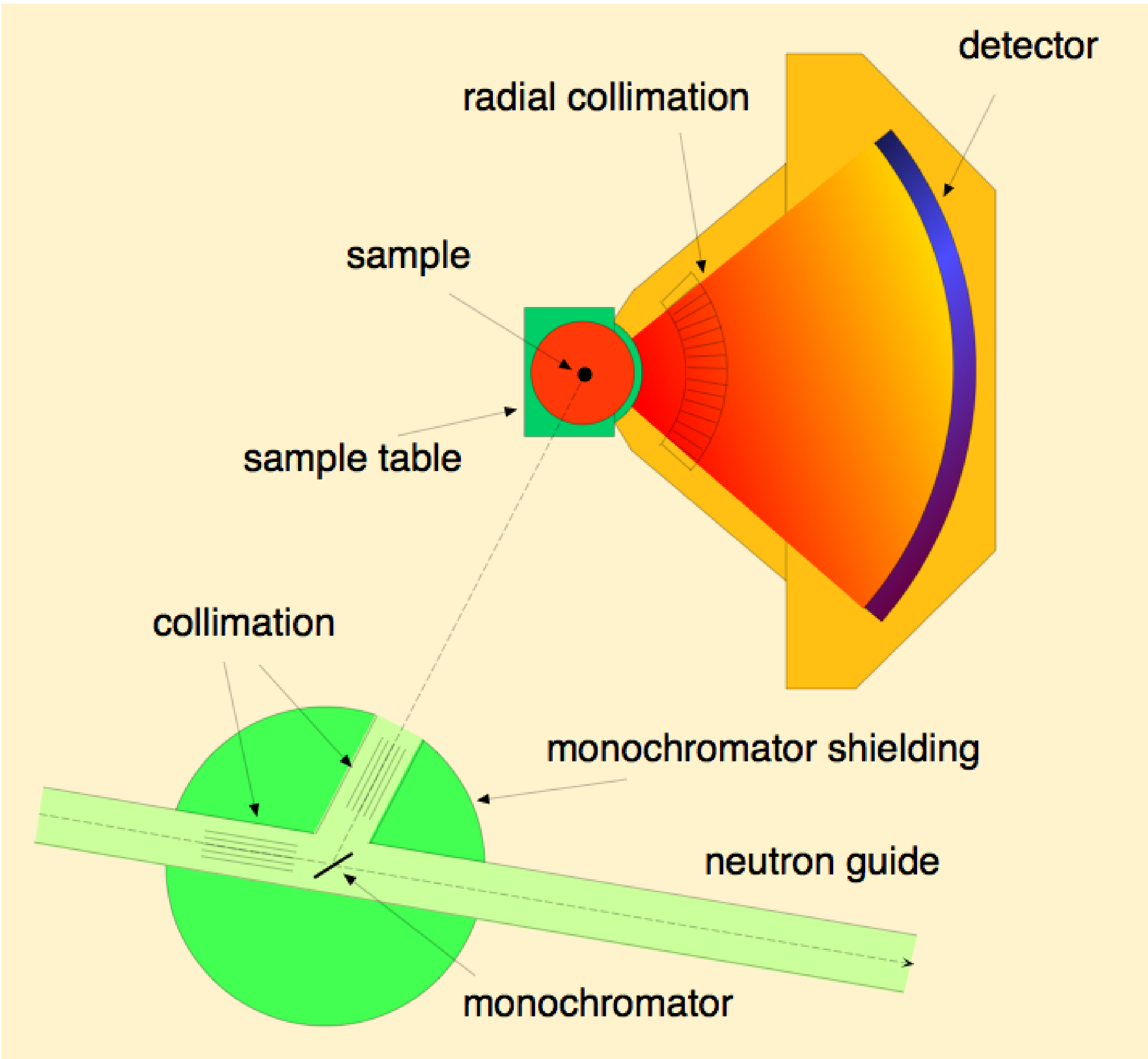


Fig. 15: The (fictional) WONI example powder diffractometer

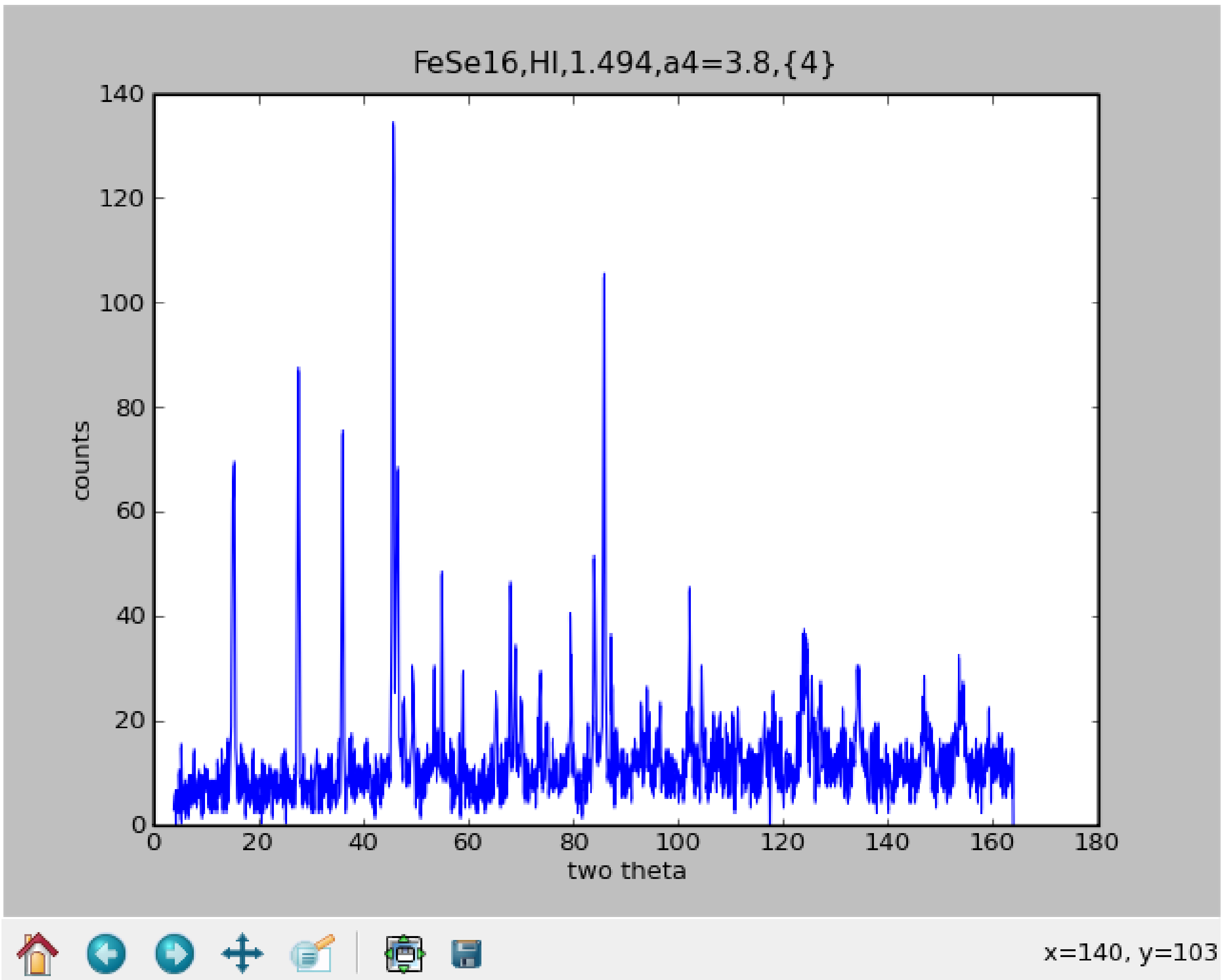


Fig. 16: Example Powder Diffraction Plot from (fictional) WONI at HYNES

1.3.2 Constructing a NeXus file for WONI

The starting point for a NeXus file for WONI will be an empty basic NeXus file hierarchy as documented in the next figure. In order to arrive at a full NeXus file, the following steps are required:

1. For each instrument component, decide which parameters need to be stored
2. Map the component parameters to NeXus groups and parameters and add the components to the `NXinstrument` hierarchy
3. Decide what needs to go into `NXdata`. While this group is optional, you are urged strongly to provide an `NXdata` group to support default plotting.
4. Fill the `NXsample` and `NXmonitor` groups

Basic structure of a NeXus file

```
entry:NXentry
  NXdata
  NXinstrument
  NXmonitor
  NXsample
```

Decide which parameters need to be stored

Now the various groups of this empty NeXus file shell need to be filled. The next step is to look at a design drawing of WONI. Identify all the instrument components like collimators, detectors, monochromators etc. For each component decide which values need to be stored. As NeXus aims to describe the experiment as good as possible, strive to capture as much information as practical.

Mapping parameters to NeXus

With the list of parameters to store for each component, consult the reference manual section on the NeXus base classes. You will find that for each of your instruments components there will be a suitable NeXus base class. Add this base class together with a name as a group under NXinstrument in your NeXus file hierarchy. Then consult the possible parameter names in the NeXus base class and match them with the parameters you wish to store for your instruments components.

As an example, consider the monochromator. You may wish to store: the wavelength, the d-value of the reflection used, the type of the monochromator and its angle towards the incoming beam. The reference manual tells you that NXcrystal is the right base class to use. Suitable fields for your parameters can be found in there to. After adding them to the basic NeXus file, the file looks like in the next figure:

Basic structure of a NeXus file with a monochromator added

```
entry:NXentry
  NXdata
  NXinstrument
     monochromator:Nxcrystal
        wavelength
        d_spacing
        rotation_angle
```

(continues on next page)

(continued from previous page)

```
          reflection
          type
    NXmonitor
    NXsample
```

If a parameter or even a whole group is missing in order to describe your experiment, do not despair! Contact the NIAC and suggest to add the group or parameter. Give a little documentation what it is for. The NIAC will check that your suggestion is no duplicate and sufficiently documented and will then proceed to enhance the base classes with your suggestion.

A more elaborate example of the mapping process is given in the section *Creating a NXDL Specification*.

Decide on `NXdata`

The `NXdata/` group is supposed to contain the data required to put up a quick plot. For WONI this is a plot of counts versus two theta (polar_angle in NeXus) as can be seen in *Example Powder Diffraction Plot from (fictional) WONI at HYNES*. Now, in `NXdata`, create links to the appropriate data items in the `NXinstrument` hierarchy. In the case of WONI, both parameters live in the `detector:NXdetector` group.

Fill in auxiliary Information

Look at the section on `NXsample` in the NeXus reference manual. Choose appropriate parameters to store for your samples. Probably at least the name will be needed.

In order to normalize various experimental runs against each other it is necessary to know about the counting conditions and especially the monitor counts of the monitor used for normalization. The NeXus convention is to store such information in a `control:NXmonitor` group at `NXentry` level. Consult the reference for `NXmonitor` for field names. If additional monitors exist within your experiment, they will be stored as additional `NXmonitor` groups at entry level.

Consult the documentation for `NXentry` in order to find out under which names to store information such as titles, user names, experiment times etc.

A more elaborate example of this process can be found in the following section on creating an application definition.

1.3.3 Creating a NXDL Specification

An NXDL specification for a NeXus file is required if you desire to standardize NeXus files from various sources. Another name for a NXDL description is application definition. A NXDL specification can be used to verify NeXus files to conform to the standard encapsulated in the application definition. The process for constructing a NXDL specification is similar to the one described above for the construction of NeXus files.

One easy way to describe how to store data in the NeXus class structure and to create a NXDL specification is to work through an example. Along the way, we will describe some key decisions that influence our particular choices of metadata selection and data organization. So, on with the example …

Application Definition Steps

With all this introductory stuff out of the way, let us look at the process required to define an application definition:

1. *Think!* hard about what has to go into the data file.
2. *Map* the required fields into the NeXus hierarchy
3. *Describe* this map in a NXDL file
4. *Standardize* your definition through communication with the NIAC

Step 1: *Think!* hard about data

This is actually the hard bit. There are two things to consider:

1. What has to go into the data file?
2. What is the normal plot for this type of data?

For the first part, one of the NeXus guiding principles gives us - Guidance! "A NeXus file must contain all the data necessary for standard data analysis."

Not more and not less for an application definition. Of course the definition of *standard* data for analysis or a *standard* plot depends on the science and the type of data being described. Consult senior scientists in the field about this is if you are unsure. Perhaps you must call an international meeting with domain experts to haggle that out. When considering this, people tend to put in everything which might come up. This is not the way to go.

A key test question is: Is this data item necessary for common data analysis? Only these necessary data items belong in an application definition.

The purpose of an application definition is that an author of upstream software who consumes the file can expect certain data items to be there at well defined places. On the other hand if there is a development in your field which analyzes data in a novel way and requires more data to do it, then it is better to err towards the side of more data.

Now for the case of WONI, the standard data analysis is either Rietveld refinement or profile analysis. For both purposes, the kind of radiation used to probe the sample (for WONI, neutrons), the wavelength of the radiation, the monitor (which tells us how long we counted) used to normalize the data, the counts and the two theta angle of each detector element are all required. Usually, it is desirable to know what is being analyzed, so some metadata would be nice: a title, the sample name and the sample temperature. The data typically being plotted is two theta against counts, as shown in *Example Powder Diffraction Plot from (fictional) WONI at HYNES* above. Summarizing, the basic information required from WONI is given next.

- *title* of measurement
- sample *name*
- sample *temperature*
- counts from the incident beam *monitor*
- type of radiation *probe*
- *wavelength* (λ) of radiation incident on sample
- angle (2θ or *two theta*) of detector elements
- *counts* for each detector element

If you start to worry that this is too little information, hold on, the section on Using an Application Definition (*Using an Application Definition*) will reveal the secret how to go from an application definition to a practical file.

Step 2: *Map* Data into the NeXus Hierarchy

This step is actually easier then the first one. We need to map the data items which were collected in Step 1 into the NeXus hierarchy. A NeXus file hierarchy starts with an `NXentry` group. At this stage it is advisable to pull up the base class definition for `NXentry` and study it. The first thing you might notice is that `NXentry` contains a field named `title`. Reading the documentation, you quickly realize that this is a good place to store our title. So the first mapping has been found.

```
title = /NXentry/title
```

Note: In this example, the mapping descriptions just contain the path strings into the NeXus file hierarchy with the class names of the groups to use. As it turns out, this is the syntax used in NXDL link specifications. How convenient!

Another thing to notice in the `NXentry` base class is the existence of a group of class `NXsample`. This looks like a great place to store information about the sample. Studying the `NXsample` base class confirms this view and there are two new mappings:

```
sample name = /NXentry/NXsample/name
sample temperature = /NXentry/NXsample/temperature
```

Scanning the `NXentry` base class further reveals there can be a `NXmonitor` group at this level. Looking up the base class for `NXmonitor` reveals that this is the place to store our monitor information.

```
monitor = /NXentry/NXmonitor/data
```

For the other data items, there seem to be no solutions in `NXentry`. But each of these data items describe the instrument in more detail. NeXus stores instrument descriptions in the `/NXentry/NXinstrument` branch of the hierarchy. Thus, we continue by looking at the definition of the `NXinstrument` base class. In there we find further groups for all possible instrument components. Looking at the schematic of WONI (*The (fictional) WONI example powder diffractometer*), we realize that there is a source, a monochromator and a detector. Suitable groups can be found for these components in `NXinstrument` and further inspection of the appropriate base classes reveals the following further mappings:

```
probe = /NXentry/NXinstrument/NXsource/probe
wavelength = /NXentry/NXinstrument/NXcrystal/wavelength
two theta of detector elements = /NXentry/NXinstrument/NXdetector/polar angle
counts for each detector element = /NXentry/NXinstrument/NXdetector/data
```

Thus we mapped all our data items into the NeXus hierarchy! What still needs to be done is to decide upon the content of the `NXdata` group in `NXentry`. This group describes the data necessary to make a quick plot of the data. For WONI this is `counts` versus `two theta`. Thus we add this mapping:

```
two theta of detector elements = /NXentry/NXdata/polar angle
counts for each detector element = /NXentry/NXdata/data
```

The full mapping of WONI data into NeXus is documented in the next table:

WONI data	NeXus path
title of measurement	`/NXentry/title`
sample *name*	`/NXentry/NXsample/name`
sample *temperature*	`/NXentry/NXsample/temperature`
monitor	`/NXentry/NXmonitor/data`
type of radiation *probe*	`/NXentry/MXinstrument/NXsource/probe`
wavelength of radiation incident on sample	`/NXentry/MXinstrument/NXcrystal/wavelength`
two theta of detector elements	`/NXentry/NXinstrument/NXdetector/polar_angle`
counts for each detector element	`/NXentry/NXinstrument/NXdetector/data`
two theta of detector elements	`/NXentry/NXdata/polar_angle`
counts for each detector element	`/NXentry/NXdata/data`

Looking at this table, one might get concerned that the two theta and counts data is stored in two places and thus duplicated. Stop worrying, this problem is solved at the NeXus API level. Typically `NXdata` will only hold links to the corresponding data items in `/NXentry/NXinstrument/NXdetector`.

In this step problems might occur. The first is that the base class definitions contain a bewildering number of parameters. This is on purpose: the base classes serve as dictionaries which define names for most things which possibly can occur. You do not have to give all that information. Keep it simple and only require data that is needed for typical data analysis for this type of application.

Another problem which can occur is that you require to store information for which there is no name in one of the existing base classes or you have a new instrument component for which there is no base class altogether. New fields and base classes can be introduced if necessary.

In any case please feel free to contact the NIAC via the mailing list with questions or suggestions.

Step 3: *Describe* this map in a NXDL file

This is even easier. Some XML editing is necessary. Fire up your XML editor of choice and open a file. If your XML editor supports XML schema while editing XML, it is worth to load `nxdl.xsd`. Now your XML editor can help you to create a proper NXDL file. As always, the start is an empty template file. This looks like the XML code below.

Note: This is just the basic XML for a NXDL definition. It is advisable to change some of the documentation strings.

NXDL template file

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
# NeXus - Neutron and X-ray Common Data Format
#
# Copyright (C) 2008-2022 NeXus International Advisory Committee (NIAC)
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser General Public
# License as published by the Free Software Foundation; either
# version 3 of the License, or (at your option) any later version.
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
```

(continues on next page)

(continued from previous page)

```
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
#
# For further information, see https://www.nexusformat.org/
-->
<definition name="NX__template__" extends="NXobject" type="group"
    category="application"
    xmlns="http://definition.nexusformat.org/nxdl/3.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://definition.nexusformat.org/nxdl/3.1 ../nxdl.xsd"
    version="1.0b"
    >
    <doc>template for a NXDL application definition</doc>
</definition>
```

For example, copy and rename the file to `NXwoni.nxdl.xml`. Then, locate the XML root element `definition` and change the `name attribute` (the XML shorthand for this attribute is `/definition/@name`) to `NXwoni`. Change the `doc` as well.

The next thing which needs to be done is adding groups into the definition. A group is defined by some XML, as in this example:

```
<group type="NXdata">

</group>
```

The type is the actual NeXus base class this group belongs to. Optionally a `name` attribute may be given (default is `data`).

Next, one needs to include data items, too. The XML for such a data item looks similar to this:

```
<field name="polar_angle" type="NX_FLOAT units="NX_ANGLE">
  <doc>Link to polar angle in /NXentry/NXinstrument/NXdetector</doc>
  <dimensions rank="1">
    <dim index="1" value="ndet"/>
  </dimensions>
</field>
```

The meaning of the `name` attribute is intuitive, the `type` can be looked up in the relevant base class definition. A `field` definition can optionally contain a `doc` element which contains a description of the data item. The `dimensions` entry specifies the dimensions of the data set. The `size` attribute in the dimensions tag sets the rank of the data, in this example: `rank="1"`. In the `dimensions` group there must be *rank* `dim` fields. Each `dim` tag holds two attributes: `index` determines to which dimension this tag belongs, the `1` means the first dimension. The `value` attribute then describes the size of the dimension. These can be plain integers, variables, such as in the example `ndet` or even expressions like `tof+1`.

Thus a NXDL file can be constructed. The full NXDL file for the WONI example is given in *Full listing of the WONI Application Definition*. Clever readers may have noticed the strong similarity between our working example `NXwoni` and `NXmonopd` since they are essentially identical. Give yourselves a cookie if you spotted this.

Step 4: *Standardize* with the NIAC

Basically you are done. Your first application definition for NeXus is constructed. In order to make your work a standard for that particular application type, some more steps are required:

- Send your application definition to the NIAC for review
- Correct your definition per the comments of the NIAC
- Cure and use the definition for a year
- After a final review, it becomes the standard

The NIAC must review an application definition before it is accepted as a standard. The one year curation period is in place in order to gain practical experience with the definition and to sort out bugs from Step 1. In this period, data shall be written and analyzed using the new application definition.

Full listing of the WONI Application Definition

Using an Application Definition

The application definition is like an interface for your data file. In practice files will contain far more information. For this, the extendable capability of NeXus comes in handy. More data can be added, and upstream software relying on the interface defined by the application definition can still retrieve the necessary information without any changes to their code.

NeXus application definitions only standardize classes. You are free to decide upon names of groups, subject to them matching regular expression for NeXus name attributes (see the *regular expression pattern for NXDL group and field names* in the *Naming Conventions* section). Note the length limit of 63 characters imposed by HDF5. Please use sensible, descriptive names and separate multi worded names with underscores.

Something most people wish to add is more metadata, for example in order to index files into a database of some sort. Go ahead, do so, if applicable, scan the NeXus base classes for standardized names. For metadata, consider to use the `NXarchive` definition. In this context, it is worth to mention that a practical NeXus file might adhere to more then one application definition. For example, WONI data files may adhere to both the `NXmonopd` and `NXarchive` definitions. The first for data analysis, the second for indexing into the database.

Often, instrument scientists want to store the complete state of their instrument in data files in order to be able to find out what went wrong if the data is unsatisfactory. Go ahead, do so, please use names from the NeXus base classes.

Site policy might require you to store the names of all your bosses up to the current head of state in data files. Go ahead, add as many `NXuser` classes as required to store that information. Knock yourselves silly over this.

Your Scientific Accounting Department (SAD) may ask of you the preposterous; to store billing information into data files. Go ahead, do so if your judgment allows. Just do not expect the NIAC to provide base classes for this and do not use the prefix NX for your classes.

In most cases, NeXus files will just have one `NXentry` class group. But it may be required to store multiple related data sets of the results of data analysis into the same data file. In this case create more entries. Each entry should be interpretable standalone, i.e. contain all the information of a complete `NXentry` class. Please keep in mind that groups or data items which stay constant across entries can always be linked to save space. Application definitions describe only what is included within an `NXentry` and so have no power to enforce any particular usage of `NXentry` groups. However, documentation within and accompanying an application definition can provide guidance and recommendations on situations where the use of multiple `NXentry` groups would be appropriate.

1.3.4 Processed Data

Data reduction and analysis programs are encouraged to store their results in NeXus data files. As far as the necessary, the normal NeXus hierarchy is to be implemented. In addition, processed data files must contain a NXprocess group. This group, that documents and preserves data provenance, contains the name of the data processing program and the parameters used to run this program in order to achieve the results stored in this entry. Multiple processing steps must have a separate entry each.

1.4 Strategies for storing information in NeXus data files

NeXus may appear daunting, at first, to use. The number of base classes is quite large as well as is the number of application definitions. This chapter describes some of the strategies that have been recommended for how to store information in NeXus data files.

When we use the term *storing*, some might be helped if they consider this as descriptions for how to *classify* their data.

It is intended for this chapter to grow, with the addition of different use cases as they are presented for suggestions.

1.4.1 Strategies: The simplest case(s)

Perhaps the simplest case might be either a step scan with two or more columns of data. Another simple case might be a single image acquired by an area detector. In either of these hypothetical cases, the situation is so simple that there is little addition information available to be described (for whatever reason).

Step scan with two or more data columns

Consider the case where we wish to store the data from a step scan. This case may involve two or more *related* 1-D arrays of data to be saved, each having the same length. For our hypothetical case, we'lll have these positioners as arrays and assume that a default plot of *photodiode* vs. *ar*:

positioner arrays	detector arrays
`ar`, `ay`, `dy`	`I0`, `I00`, `time`, `Epoch`, `photodiode`

Data file structure for *Step scan with two or more data columns*

```
file.nxs: NeXus HDF5 data file
   @default = "entry"
   entry: NXentry
      @NX_class = "NXentry"
      @default = "data"
      data: NXdata
         @NX_class = "NXdata"
         @signal = "photodiode"
         @axes = "ar"
         ar: NX_FLOAT[]
         ay: NX_FLOAT[]
         dy: NX_FLOAT[]
         I0: NX_FLOAT[]
         I00: NX_FLOAT[]
```

(continues on next page)

(continued from previous page)

```
        time: NX_FLOAT[]
        Epoch: NX_FLOAT[]
        photodiode: NX_FLOAT[]
```

1.4.2 Strategies: The wavelength

Where should the wavelength of my experiment be written? This is one of the *Frequently Asked Questions*. The canonical location to store wavelength has been:

```
/NXentry/NXinstrument/NXcrystal/wavelength
```

Partial data file structure for *canonical location to store wavelength*

```
entry: NXentry
   @NX_class = NXentry
   instrument: NXinstrument
      @NX_class = NXinstrument
      crystal: NXcrystal
         @NX_class = NXcrystal
         wavelength: NX_FLOAT
```

More recently, this location makes more sense to many:

```
/NXentry/NXinstrument/NXmonochromator/wavelength
```

Partial data file structure for *location which makes more sense to many* to store wavelength

```
entry: NXentry
   @NX_class = NXentry
   instrument: NXinstrument
      @NX_class = NXinstrument
      monochromator: NXmonochromator
         @NX_class = NXmonochromator
         wavelength: NX_FLOAT
```

NXcrystal describes a crystal monochromator or analyzer. Recently, scientists with monochromatic radiation not defined by a crystal, such as from an electron-beam undulator or a neutron helical velocity selector, were not satisfied with creating a fictitious instance of a crystal just to preserve the wavelength from their instrument. Thus, the addition of the NXmonochromator base class to NeXus, which also allows "energy" to be specified if one is so inclined.

Note: See the *Class path specification* section for a short discussion of the difference between the HDF5 path and the NeXus symbolic class path.

1.4.3 Strategies: Time-stamped data

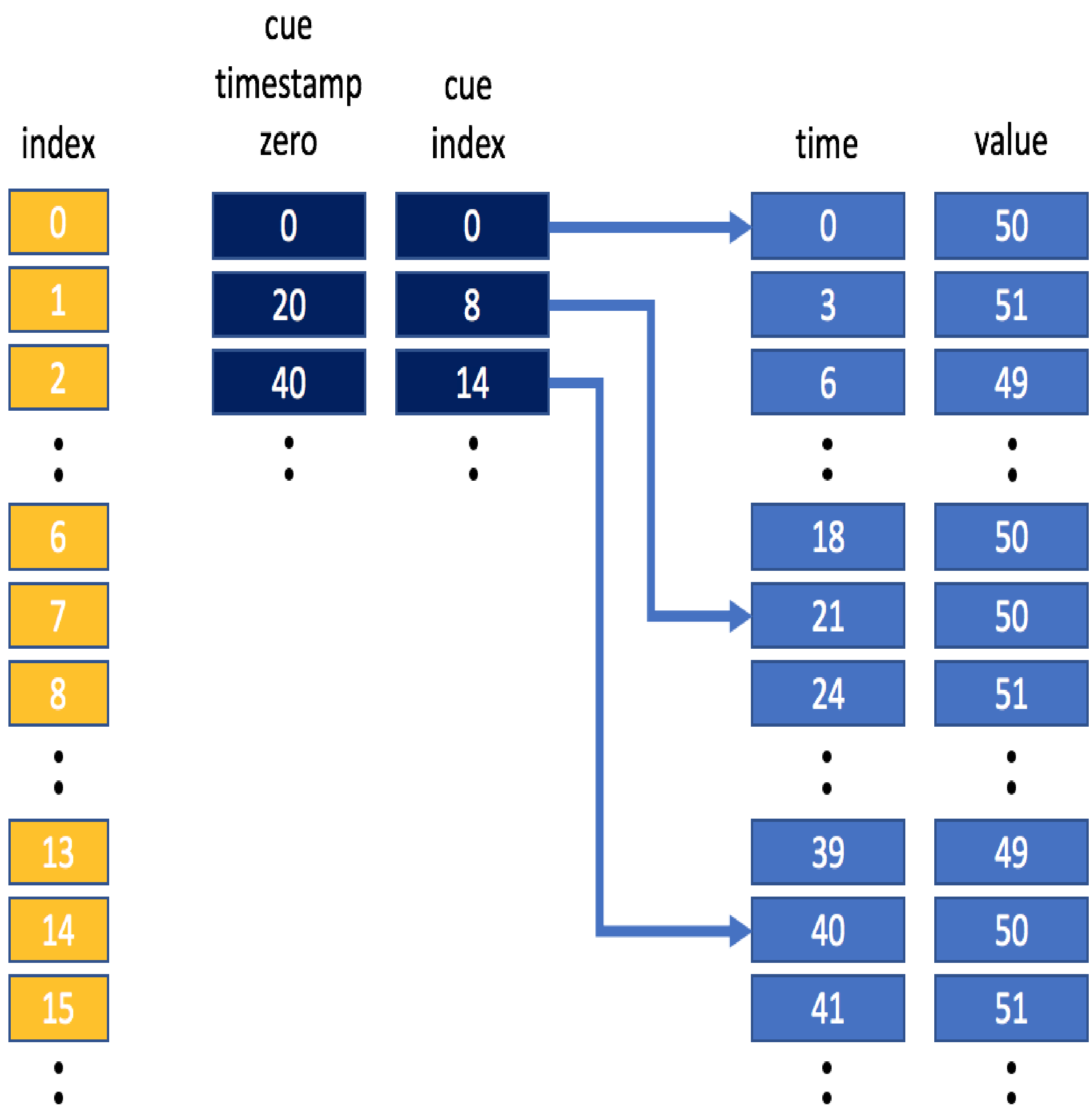
How should I store time-stamped data?

Time-stamped data can be stored in either NXlog and NXevent_data structures. Of the two, NXlog is the most important one, NXevent_data is normally only used for storing detector time of flight event data and NXlog would be used for storing any other time-stamped data, e.g. sample temperature, chopper top-dead-centre, motor position, detector images etc.

Regarding the NeXus file structure to use, there is one simple rule: just use the standard NeXus file structure but insert/replace the fields for streamed data elements through NXlog or NXevent_data structures. For example, consider the collection of detector images against a change in the magnetic field on the sample. Then, both NXsample/magnetic_field and NXdetector/data would be NXlog structures containing the time stamped data.

Both NXlog and NXevent_data have additional support for storing time-stamped data in the form of cues; cues can be used to place markers in the data that allow one to quickly look up coarse time ranges of interest. This coarse range of data can then be manually trimmed to be more selective, if required. The application writing the NeXus file is responsible for writing cues and when they are written. For example, the cue could be written every 10 seconds, every pulse, every 100 datapoints and so on.

Let's consider the case where NXlog is being used to store sample temperature data that has been sampled once every three seconds. The application that wrote the data has added cues every 20 seconds. Pictorially, this may look something like this:



If we wanted to retrieve the mean temperature between 30 and 40 seconds, we would use the cues to grab the data between 20 seconds and 40 seconds, and then trim that data to get the data we want. Obviously in this simple example this does not gain us a lot, but it is easy to see that in a large dataset having appropriately placed cues can save significant computational time when looking up values in a certain time-stamp range. NeXus has actually borrowed the cueing table concept from video file formats where it allows viewing software to quickly access your favourite scene. Correspondingly, cueing in NeXus allows you to quickly access your favourite morsel of time stamped data.

In the NeXus Features repository, the feature ECB064453EDB096D shows example code that uses cues to select time-stamped data.

1.4.4 Strategies: The next case

The *NIAC: The NeXus International Advisory Committee* welcomes suggestions for additional sections in this chapter.

1.5 Verification and validation of files

The intent of verification and validation of files is to ensure, in an unbiased way, that a given file conforms to the relevant specifications. Validation does not check that the data content of the file is sensible; this requires scientific interpretation based on the technique.

Validation is useful to anyone who manipulates or modifies the contents of NeXus files. This includes scientists/users, instrument staff, software developers, and those who might mine the files for metadata. First, the scientist or user of the data must be certain that the information in a file can be located reliably. The instrument staff or software developer must be confident the information they have written to the file has been located and formatted properly. At some time, the content of the NeXus file may contribute to a larger body of work such as a metadata catalog for a scientific instrument, a laboratory, or even an entire user facility.

1.5.1 nxvalidate

NeXus validation tool written in C (not via NAPI).

Its dependencies are libxml2 and the HDF5 libraries, version 1.8.9 or better. Its purpose is to validate HDF5 files against NeXus application definitions.

See the program documentation for more details: https://github.com/nexusformat/cnxvalidate.git

1.5.2 punx

Python Utilities for NeXus HDF5 files

punx can validate both NXDL files and NeXus HDF5 data files, as well as print the structure of any HDF5 file, even non-NeXus files.

NOTE: project is under initial construction, not yet released for public use, but is useful in its present form (version 0.2.5).

punx can show the tree structure of any HDF5 file. The output is more concise than that from *h5dump*.

See the program documentation for more details: https://punx.readthedocs.io

1.6 Frequently Asked Questions

This is a list of commonly asked questions concerning the NeXus data format.

1. Is it Nexus, NeXus or NeXuS?

   NeXus is correct. It is a format for data from **Neutron** and **X-ray** facilities, hence those first letters are capitalised. The format is also used for muon experiments, but there is no *mu* (or m) in NeXus and no s in muon. So the s stays in lower case.

2. How many facilities use NeXus?

This is not easy to say, not all facilities using NeXus actively participate in the committee. Some facilities have reported their adoption status on the Facilities web page. Please have a look at this list. Keep in mind that it is never fully complete or up to date.

3. NeXus files are binary? This is crazy! How am I supposed to see my data?

   Various tools are listed in the *NeXus Utilities* section to inspect NeXus data files. The easiest graphical tool to use is *HDFview* which can open any HDF file. Other tools such as *PyMCA* and *NeXPy* provide visualization of scientific data while *h5dump* and *punx tree* provide text renditions of content and structure. If you want to try, for example `nxbrowse` is a utility provided by the NeXus community that can be very helpful to those who want to inspect their files and avoid graphical applications. For larger data volumes the binary backends used with the appropriate tools are by far superior in terms of efficiency and speed and most users happily accept that after having worked with supersized "human readable" files for a while.

4. What on-disk file format should I choose for my data?

   HDF5 is the default file container to use for NeXus data. It is the recommended format for all applications. HDF4 is still supported as a on disk format for NeXus but for new installations preference should be given to HDF5.

5. Why are the NeXus classes so complicated? I'll never store all that information

   The NeXus classes are essentially glossaries of terms. If you need to store a piece of information, consult the class definitions to see if it has been defined. If so, use it. It is not compulsory to include every item that has been defined in the base class if it is not relevant to your experiment. On the other hand, a NeXus application definition lists a smaller set of compulsory items that should allow other researchers or software to analyze your data. You should really follow the application definition that corresponds to your experiment to take full advantage of NeXus.

6. I don't like NeXus. It seems much faster and simpler to develop my own file format. Why should I even consider NeXus?

   If you consider using an efficient on disk storage format, HDF5 is a better choice than most others. It is fast and efficient and well supported in all mainstream programming languages and a fair share of popular analysis packages. The format is so widely used and backed by a big organisation that it will continue to be supported for the foreseeable future. So if you are going to use HDF5 anyway, why not use the NeXus definition to lay out the data in a standardised way? The NeXus community spent years trying to get the standard right and while you will not agree with every single choice they made in the past, you should be able to store the data you have in a quite reasonable way. If you do not comply with NeXus, chances are most people will perceive your format as different but not necessarily better than NeXus by any large measure. So it may not be worth the effort. Seriously.

   If you encounter any problems because the classes are not sufficient to describe your experiment, please contact the *mailing list*. Pull requests for the defintions repository (for example adding contributed defintions) are also welcome (see next question). The NIAC is always willing to consider new proposals.

7. **I want to contribute an application definition.**
   How do I go about it?

   Read the NXDL Tutorial in *Creating a NXDL Specification* and have a try. You can ask for help on the *mailing lists*. Once you have a definition that is working well for at least your case, you can submit it to the NIAC for acceptance as a standard. The procedures for acceptance are defined in the NIAC constitution.[1]

8. What is the purpose of `NXdata`?

[1] Refer to the most recent version of the NIAC constitution on the NIAC web page: https://www.nexusformat.org/NIAC.html#constitution

NXdata identifies the default plottable data. This is one of the basic motivations (see *Simple plotting*) for the NeXus standard. The choice of the name NXdata is historic and does not really reflect its function. The NXdata group contains data or links to the data stored elsewhere.

9. How do I identify the plottable data?

   See the section: *Find the plottable data*.

10. Why aren't NXsample and NXmonitor groups stored in the NXinstrument group?

    A NeXus file can contain a number of NXentry groups, which may represent different scans in an experiment, or sample and calibration runs, etc. In many cases, though by no means all, the instrument has the same configuration so that it would be possible to save space by storing the NXinstrument group once and using multiple links in the remaining NXentry groups. It is assumed that the sample and monitor information would be more likely to change from run to run, and so should be stored at the top level.

11. Can I use a NXDL specification to parse a NeXus data file?

    This should be possible as there is nothing in the NeXus specifications to prevent this but it is not implemented in NAPI. You would need to implement it for yourself.

12. Do I have to use the NAPI subroutines? Can't I read (or write) the NeXus data files with my own routines?

    You are not required to use the NAPI to write valid NeXus data files. It is possible to avoid the NAPI to write and read valid NeXus data files. But, the programmer who chooses this path must have more understanding of how the NeXus HDF data file is written. Validation of data files written without the NAPI is strongly encouraged.

13. I'm using links to place data in two places. Which one should be the data and which one is the link?

    Note: NeXus uses HDF5 hard links

    In HDF, a hard link points to a data object. A soft link points to a directory entry. Since NeXus uses hard links, there is no need to distinguish between two (or more) directory entries that point to the same data.

    Both places have pointers to the actual data. That is the way hard links work in HDF5. There is no need for a preference to either location. NeXus defines a target attribute to label one directory entry as the source of the data (in this, the link *target*). This has value in only a few situations such as when converting the data from one format to another. By identifying the original in place, duplicate copies of the data are not converted.

14. **If I write my data according to the current specification for NXsas**
    (substitute any other application definition), will other software be able to read my data?

    Yes. NXsas, like other application.definitions, defines and names the *minimum information* required for analysis or data processing. As long as all the information required by the specification is present, analysis software should be able to process the data. If other information is also present, there is no guarantee that small-angle scattering analysis software will notice.

15. Where do I store the wavelength of my experiment?

    See the *Strategies: The wavelength* section.

16. Where do I store metadata about my experiment?

    See the *Where to Store Metadata* section.

17. What file extension should I use when writing a NeXus data file?

Any extension is permitted. Common extensions are *.h5*, *.hdf*, *.hdf5*, and *.nxs* while others are possible. See the many examples in the NeXus exampledata repository. (https://github.com/nexusformat/exampledata)

18. Can instances of classes inside definitions require new fields that were previously optional?

    Yes. That is one of the motivations to have application definitions. By default, all content in an application definition is required.

    For example, the `radiation` field in NXcanSAS requires 1 (and only 1) instance.

19. Can instances of classes inside definitions make optional new fields that were previously not mentioned?

    Yes. To make it optional, set attribute `minOccurs="0"`.

    For example, see the `Idev` field in NXcanSAS.

20. Can instances of classes inside definitions require new fields that were previously not mentioned?

    Yes.

    For example, see the `qx` field in NXiqproc.

21. Can we view the process of defining classes within an application definition as defining a subclass of the original class? That is, all instances of the class within the definition are valid instances of the original class, but not vice-versa?

    Keep in mind that NeXus is not specifically object oriented. The putative super class might be either NXentry (for single-technique data, such as SAXS) or NXsubentry (for multi-technique data such as SAXS/WAXS/USAXS/GIWAXS or SAXS/SANS).

    If you are thinking of a new application definition that uses another as a starting point (like a super class), then there is an `extends` attribute in the definition element of the NXDL file (example here from NXarpes):

```
<definition name="NXarpes" extends="NXobject" type="group"
```

    which describes this relationship. For most (?all?) all NXDL files to date, they extend the NXobject base class (the base object of NeXus).

CHAPTER

TWO

EXAMPLES OF WRITING AND READING NEXUS DATA FILES

Simple examples of reading and writing NeXus data files are provided in the *NeXus Introduction* chapter and also in the *NAPI: NeXus Application Programmer Interface (frozen)* chapter.

2.1 Code Examples in Various Languages

Each example in this section demonstrates writing and reading NeXus compliant files in various languages with different libraries. Most examples are using the HDF5 file format. Note however that other container formats like the legacy format HDF4 or XML can also be used to store NeXus compliant data.

Please be aware that not all examples are up to date with the latest format recommendations.

2.1.1 HDF5 in C with `libhdf5`

C-language code examples are provided for writing and reading NeXus-compliant files using the native HDF5 interfaces. These examples are derived from the simple NAPI examples for *writing* and *reading* given in the *Introduction* chapter.

Writing a simple NeXus file using native HDF5 commands in C

Note: This example uses the new method described in *Associating plottable data using attributes applied to the NXdata group* for indicating plottable data.

```
/**
 * This is an example how to write a valid NeXus file
 * using the HDF-5 API alone. Ths structure which is
 * going to be created is:
 *
 * scan:NXentry
 *      data:NXdata
 *         @signal = "counts"
 *         @axes = "two_theta"
 *         @two_theta_indices = 0
 *         counts[]
 *            @units="counts"
 *         two_theta[]
 *            @units="degrees"
```

(continues on next page)

(continued from previous page)

```
*
 *  WARNING: each of the HDF function below needs to be
 *  wrapped into something like:
 *
 *  if((hdfid = H5function(...)) < 0){
 *     handle error gracefully
 *  }
 *  I left the error checking out in order to keep the
 *  code clearer
 *
 *  This also installs a link from /scan/data/two_theta to /scan/hugo
 *
 *  Mark Koennecke, October 2011
 */
#include <hdf5.h>
#include <stdlib.h>
#include <string.h>

static void write_string_attr(hid_t hid, const char* name, const char* value)
{
  /* HDF-5 handles */
  hid_t atts, atttype, attid;

  atts = H5Screate(H5S_SCALAR);
  atttype = H5Tcopy(H5T_C_S1);
  H5Tset_size(atttype, strlen(value));
  attid = H5Acreate(hid,name, atttype, atts, H5P_DEFAULT, H5P_DEFAULT);
  H5Awrite(attid, atttype, value);
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);
}

static void write_int_attr(hid_t hid, const char* name, int value)
{
  /* HDF-5 handles */
  hid_t atts, atttype, attid;

  atts = H5Screate(H5S_SCALAR);
  atttype = H5Tcopy(H5T_NATIVE_INT);
  H5Tset_size(atttype,1);
  attid = H5Acreate(hid,name, atttype, atts, H5P_DEFAULT, H5P_DEFAULT);
  H5Awrite(attid, atttype, &value);
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);
}

#define LENGTH 400
int main(int argc, char *argv[])
{
  float two_theta[LENGTH];
```

(continues on next page)

(continued from previous page)

```
  int counts[LENGTH], i, rank;

  /* HDF-5 handles */
  hid_t fid, fapl, gid;
  hid_t datatype, dataspace, dataprop, dataid;
  hsize_t dim[1], maxdim[1];


  /* create some data: nothing NeXus or HDF-5 specific */
  for(i = 0; i < LENGTH; i++){
    two_theta[i] = 10. + .1*i;
    counts[i] = (int)(1000 * ((float)random()/(float)RAND_MAX));
  }
  dim[0] = LENGTH;
  maxdim[0] = LENGTH;
  rank = 1;



  /*
   * open the file. The file attribute forces normal file
   * closing behaviour down HDF-5's throat
   */
  fapl = H5Pcreate(H5P_FILE_ACCESS);
  H5Pset_fclose_degree(fapl,H5F_CLOSE_STRONG);
  fid = H5Fcreate("NXfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT,fapl);
  H5Pclose(fapl);


  /*
   * create scan:NXentry
   */
  gid = H5Gcreate(fid, "scan",H5P_DEFAULT,H5P_DEFAULT,H5P_DEFAULT);
  /*
   * store the NX_class attribute. Notice that you
   * have to take care to close those hids after use
   */
  write_string_attr(gid, "NX_class", "NXentry");

  /*
   * same thing for data:Nxdata in scan:NXentry.
   */
  gid = H5Gcreate(fid, "/scan/data",H5P_DEFAULT,H5P_DEFAULT,H5P_DEFAULT);
  write_string_attr(gid, "NX_class", "NXdata");

  /*
   * define axes.
   */
  write_string_attr(gid, "signal", "counts");
  write_string_attr(gid, "axes", "two_theta");
  write_int_attr(gid, "two_theta_indices", 0);

```

(continues on next page)

(continued from previous page)

```
/*
 * store the counts dataset
 */
dataspace = H5Screate_simple(rank,dim,maxdim);
datatype = H5Tcopy(H5T_NATIVE_INT);
dataprop = H5Pcreate(H5P_DATASET_CREATE);
dataid = H5Dcreate(gid,"counts",datatype,dataspace,H5P_DEFAULT,dataprop,H5P_DEFAULT);
H5Dwrite(dataid, datatype, H5S_ALL, H5S_ALL, H5P_DEFAULT, counts);
H5Sclose(dataspace);
H5Tclose(datatype);
H5Pclose(dataprop);
/*
 * set the units attribute
 */
write_string_attr(dataid, "units", "counts");

H5Dclose(dataid);

/*
 * store the two_theta dataset
 */
dataspace = H5Screate_simple(rank,dim,maxdim);
datatype = H5Tcopy(H5T_NATIVE_FLOAT);
dataprop = H5Pcreate(H5P_DATASET_CREATE);
dataid = H5Dcreate(gid,"two_theta",datatype,dataspace,H5P_DEFAULT,dataprop,H5P_
↪DEFAULT);
H5Dwrite(dataid, datatype, H5S_ALL, H5S_ALL, H5P_DEFAULT, two_theta);
H5Sclose(dataspace);
H5Tclose(datatype);
H5Pclose(dataprop);

/*
 * set the units attribute
 */
write_string_attr(dataid, "units", "degrees");

/*
 * set the target attribute for linking
 */
write_string_attr(dataid, "target", "/scan/data/two_theta");

H5Dclose(dataid);

/*
 * make a link in /scan to /scan/data/two_theta, thereby
 * renaming two_theta to hugo
 */
H5Glink(fid,H5G_LINK_HARD,"/scan/data/two_theta","/scan/hugo");

/*
 * close the file
 */
```

(continues on next page)

(continued from previous page)

```
  H5Fclose(fid);
}
```

Reading a simple NeXus file using native HDF5 commands in C

```
/**
 * Reading example for reading NeXus files with plain
 * HDF-5 API calls. This reads out counts and two_theta
 * out of the file generated by nxh5write.
 *
 * WARNING: I left out all error checking in this example.
 * In production code you have to take care of those errors
 *
 * Mark Koennecke, October 2011
 */
#include <hdf5.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  float *two_theta = NULL;
  int *counts = NULL,  rank, i;
  hid_t fid, dataid, fapl;
  hsize_t *dim = NULL;
  hid_t dataspace, memdataspace;

  /*
   * Open file, thereby enforcing proper file close
   * semantics
   */
  fapl = H5Pcreate(H5P_FILE_ACCESS);
  H5Pset_fclose_degree(fapl,H5F_CLOSE_STRONG);
  fid = H5Fopen("NXfile.h5", H5F_ACC_RDONLY,fapl);
  H5Pclose(fapl);

  /*
   * open and read the counts dataset
   */
  dataid = H5Dopen(fid,"/scan/data/counts",H5P_DEFAULT);
  dataspace = H5Dget_space(dataid);
  rank = H5Sget_simple_extent_ndims(dataspace);
  dim = malloc(rank*sizeof(hsize_t));
  H5Sget_simple_extent_dims(dataspace, dim, NULL);
  counts = malloc(dim[0]*sizeof(int));
  memdataspace = H5Tcopy(H5T_NATIVE_INT32);
  H5Dread(dataid,memdataspace,H5S_ALL, H5S_ALL,H5P_DEFAULT, counts);
  H5Dclose(dataid);
  H5Sclose(dataspace);
  H5Tclose(memdataspace);

```

(continues on next page)

(continued from previous page)

```
  /*
   * open and read the two_theta data set
   */
  dataid = H5Dopen(fid,"/scan/data/two_theta",H5P_DEFAULT);
  dataspace = H5Dget_space(dataid);
  rank = H5Sget_simple_extent_ndims(dataspace);
  dim = malloc(rank*sizeof(hsize_t));
  H5Sget_simple_extent_dims(dataspace, dim, NULL);
  two_theta = malloc(dim[0]*sizeof(float));
  memdataspace = H5Tcopy(H5T_NATIVE_FLOAT);
  H5Dread(dataid,memdataspace,H5S_ALL, H5S_ALL,H5P_DEFAULT, two_theta);
  H5Dclose(dataid);
  H5Sclose(dataspace);
  H5Tclose(memdataspace);



  H5Fclose(fid);

  for(i = 0; i < dim[0]; i++){
    printf("%8.2f %10d\n", two_theta[i], counts[i]);
  }

}
```

2.1.2 HDF5 in Python with `h5py`

One way to gain a quick familiarity with NeXus is to start working with some data. For at least the first few examples in this section, we have a simple two-column set of 1-D data, collected as part of a series of alignment scans by the APS USAXS instrument during the time it was stationed at beam line 32ID. We will show how to write this data using the Python language and the `h5py` package[1] (using `h5py` calls directly rather than using the NeXus NAPI). The actual data to be written was extracted (elsewhere) from a `spec`[2] data file and read as a text block from a file by the Python source code. Our examples will start with the simplest case and add only mild complexity with each new case since these examples are meant for those who are unfamiliar with NeXus.

Code examples

Getting started

Write a NeXus HDF5 File

In the main code section of *simple_example_basic_write.py*, the data (`mr` is similar to "two_theta" and `I00` is similar to "counts") is collated into two Python lists. We use the **numpy** package to read the file and parse the two-column format.

The new HDF5 file is opened (and created if not already existing) for writing, setting common NeXus attributes in the same command from our support library. Proper HDF5+NeXus groups are created for `/entry:NXentry/`

[1] *h5py*: https://www.h5py.org/

[2] *SPEC*: http://certif.com/spec.html

mr_scan:NXdata. Since we are not using the NAPI, our support library must create and set the NX_class attribute on each group.

Note: We want to create the desired structure of /entry:NXentry/mr_scan:NXdata/.

1. First, our support library calls f = h5py.File() to create the file and root level NeXus structure.
2. Then, it calls nxentry = f.create_group("entry") to create the NXentry group called entry at the root level.
3. Then, it calls nxdata = nxentry.create_group("mr_scan") to create the NXentry group called entry as a child of the NXentry group.

Next, we create a dataset called title to hold a title string that can appear on the default plot.

Next, we create datasets for mr and I00 using our support library. The data type of each, as represented in numpy, will be recognized by h5py and automatically converted to the proper HDF5 type in the file. A Python dictionary of attributes is given, specifying the engineering units and other values needed by NeXus to provide a default plot of this data. By setting signal="I00" as an attribute on the group, NeXus recognizes I00 as the default *y* axis for the plot. The axes="mr" attribute on the NXdata group connects the dataset to be used as the *x* axis.

Finally, we *must* remember to call f.close() or we might corrupt the file when the program quits.

simple_example_basic_write.py: Write a NeXus HDF5 file using Python with h5py

```python
#!/usr/bin/env python
"""Writes a NeXus HDF5 file using h5py and numpy"""

from pathlib import Path
import datetime
import h5py  # HDF5 support
import numpy

print("Write a NeXus HDF5 file")
fileName = "simple_example_basic.nexus.hdf5"
timestamp = datetime.datetime.now().astimezone().isoformat()

# load data from two column format
data_filename = str(Path(__file__).absolute().parent.parent / "simple_example.dat")
data = numpy.loadtxt(data_filename).T
mr_arr = data[0]
i00_arr = numpy.asarray(data[1], "int32")

# create the HDF5 NeXus file
with h5py.File(fileName, "w") as f:
    # point to the default data to be plotted
    f.attrs["default"] = "entry"
    # give the HDF5 root some more attributes
    f.attrs["file_name"] = fileName
    f.attrs["file_time"] = timestamp
    f.attrs["instrument"] = "APS USAXS at 32ID-B"
    f.attrs["creator"] = "simple_example_basic_write.py"
    f.attrs["NeXus_version"] = "4.3.0"
    f.attrs["HDF5_Version"] = h5py.version.hdf5_version
```

(continues on next page)

(continued from previous page)

```
    f.attrs["h5py_version"] = h5py.version.version

    # create the NXentry group
    nxentry = f.create_group("entry")
    nxentry.attrs["NX_class"] = "NXentry"
    nxentry.attrs["default"] = "mr_scan"
    nxentry.create_dataset("title", data="1-D scan of I00 v. mr")

    # create the NXentry group
    nxdata = nxentry.create_group("mr_scan")
    nxdata.attrs["NX_class"] = "NXdata"
    nxdata.attrs["signal"] = "I00"  # Y axis of default plot
    nxdata.attrs["axes"] = "mr"  # X axis of default plot
    nxdata.attrs["mr_indices"] = [
        0,
    ]  # use "mr" as the first dimension of I00

    # X axis data
    ds = nxdata.create_dataset("mr", data=mr_arr)
    ds.attrs["units"] = "degrees"
    ds.attrs["long_name"] = "USAXS mr (degrees)"  # suggested X axis plot label

    # Y axis data
    ds = nxdata.create_dataset("I00", data=i00_arr)
    ds.attrs["units"] = "counts"
    ds.attrs["long_name"] = "USAXS I00 (counts)"  # suggested Y axis plot label

print("wrote file:", fileName)
```

Read a NeXus HDF5 File

The file reader, *simple_example_basic_read.py*, is very simple since the bulk of the work is done by `h5py`. Our code opens the HDF5 we wrote above, prints the HDF5 attributes from the file, reads the two datasets, and then prints them out as columns. As simple as that. Of course, real code might add some error-handling and extracting other useful stuff from the file.

Note: See that we identified each of the two datasets using HDF5 absolute path references (just using the group and dataset names). Also, while coding this example, we were reminded that HDF5 is sensitive to upper or lowercase. That is, `I00` is not the same is `i00`.

simple_example_basic_read.py: Read a NeXus HDF5 file using Python with h5py

```
#!/usr/bin/env python
"""Reads NeXus HDF5 files using h5py and prints the contents"""

import h5py  # HDF5 support

fileName = "simple_example_basic.nexus.hdf5"
with h5py.File(fileName, "r") as f:
    for item in f.attrs.keys():
        print(item + ":", f.attrs[item])
    mr = f["/entry/mr_scan/mr"]
    i00 = f["/entry/mr_scan/I00"]
    print("%s\t%s\t%s" % ("#", "mr", "I00"))
    for i in range(len(mr)):
        print("%d\t%g\t%d" % (i, mr[i], i00[i]))
```

Output from `simple_example_basic_read.py` is shown next.

Output from `simple_example_basic_read.py`

```
file_name: simple_example_basic.nexus.hdf5
file_time: 2010-10-18T17:17:04-0500
creator: simple_example_basic_write.py
HDF5_Version: 1.8.5
NeXus_version: 4.3.0
h5py_version: 1.2.1
instrument: APS USAXS at 32ID-B
#   mr  I00
0   17.9261 1037
1   17.9259 1318
2   17.9258 1704
3   17.9256 2857
4   17.9254 4516
5   17.9252 9998
6   17.9251 23819
7   17.9249 31662
8   17.9247 40458
9   17.9246 49087
10  17.9244 56514
11  17.9243 63499
12  17.9241 66802
13  17.9239 66863
14  17.9237 66599
15  17.9236 66206
16  17.9234 65747
17  17.9232 65250
18  17.9231 64129
19  17.9229 63044
20  17.9228 60796
21  17.9226 56795
22  17.9224 51550
```

(continues on next page)

(continued from previous page)

```
23  17.9222 43710
24  17.9221 29315
25  17.9219 19782
26  17.9217 12992
27  17.9216 6622
28  17.9214 4198
29  17.9213 2248
30  17.9211 1321
```

downloads

The Python code and files related to this section may be downloaded from the following table.

file	description
`../simple_example.dat`	2-column ASCII data used in this section
`simple_example_basic_read.py`	python code to read example *simple_example_basic.nexus.hdf5*
`simple_example_basic_write.py`	python code to write example *simple_example_basic.nexus.hdf5*
`simple_example_basic.nexus_h5dump.txt`	*h5dump* analysis of the NeXus file
`simple_example_basic.nexus.hdf5`	NeXus file written by *BasicWriter*
`simple_example_basic.nexus_structure.txt`	*punx tree* analysis of the NeXus file

Write a NeXus HDF5 file

In this example, the 1-D scan data will be written into the simplest possible NeXus HDF5 data file, containing only the required NeXus components. NeXus requires at least one NXentry group at the root level of an HDF5 file. The `NXentry` group contains *all the data and associated information that comprise a single measurement*. `NXdata` is used to describe the plottable data in the `NXentry` group. The simplest place to store data in a NeXus file is directly in the `NXdata` group, as shown in the next figure.

In the *above figure*, the data file (`simple_example_write1_h5py.hdf5`) contains a hierarchy of items, starting with an `NXentry` named `entry`. (The full HDF5 path reference, `/entry` in this case, is shown to the right of each component in the data structure.) The next `h5py` code example will show how to build an HDF5 data file with this structure. Starting with the numerical data described above, the only information written to the file is the *absolute* minimum information NeXus requires. In this example, you can see how the HDF5 file is created, how *Groups* and datasets (*Fields*) are created, and how *Attributes* are assigned. Note particularly the `NX_class` attribute on each HDF5 group that describes which of the NeXus base.class.definitions is being used. When the next Python program (`simple_example_write1_h5py.py`) is run from the command line (and there are no problems), the `simple_example_write1_h5py.hdf5` file is generated.

```
#!/usr/bin/env python
"""
Writes the simplest NeXus HDF5 file using h5py

Uses method accepted at 2014NIAC
according to the example from Figure 1.3
in the Introduction chapter
```

(continues on next page)

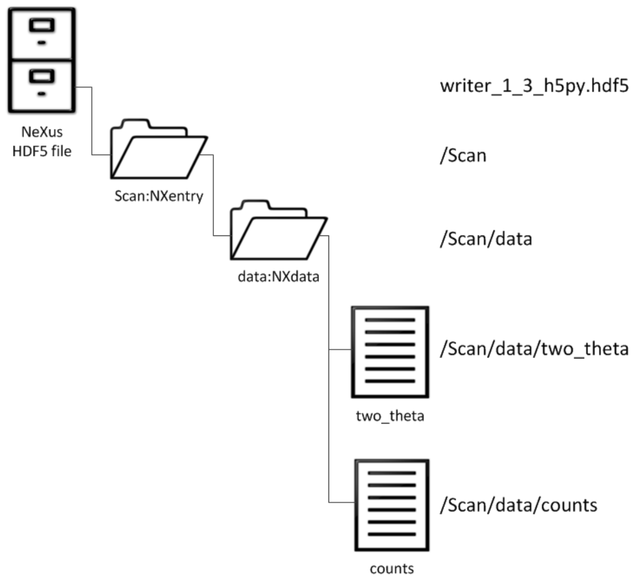


Fig. 1: Simple Example

(continued from previous page)

```
"""

from pathlib import Path
import h5py
import numpy

filename = str(Path(__file__).absolute().parent.parent / "simple_example.dat")
buffer = numpy.loadtxt(filename).T
tthData = buffer[0]  # float[]
countsData = numpy.asarray(buffer[1], "int32")  # int[]

with h5py.File("simple_example_write1.hdf5", "w") as f:  # create the HDF5 NeXus file
    # since this is a simple example, no attributes are used at this point

    nxentry = f.create_group("Scan")
    nxentry.attrs["NX_class"] = "NXentry"

    nxdata = nxentry.create_group("data")
    nxdata.attrs["NX_class"] = "NXdata"
    nxdata.attrs["signal"] = "counts"
    nxdata.attrs["axes"] = "two_theta"
    nxdata.attrs["two_theta_indices"] = [
        0,
    ]

    tth = nxdata.create_dataset("two_theta", data=tthData)
    tth.attrs["units"] = "degrees"

    counts = nxdata.create_dataset("counts", data=countsData)
    counts.attrs["units"] = "counts"
```

One of the tools provided with the HDF5 support libraries is the `h5dump` command, a command-line tool to print out the contents of an HDF5 data file. With no better tool in place (the output is verbose), this is a good tool to investigate what has been written to the HDF5 file. View this output from the command line using `h5dump simple_example_write1.hdf5`. Compare the data contents with the numbers shown above. Note that the various HDF5 data types have all been decided by the `h5py` support package.

Note: The only difference between this file and one written using the NAPI is that the NAPI file will have some additional, optional attributes set at the root level of the file that tells the original file name, time it was written, and some version information about the software involved.

Since the output of `h5dump` is verbose (see the *Downloads* section below), the *punx tree* tool[1] was used to print out the structure of HDF5 data files. This tool provides a simplified view of the NeXus file. Here is the output:

```
Scan:NXentry
  @NX_class = "NXentry"
  data:NXdata
    @NX_class = "NXdata"
    @axes = "two_theta"
    @signal = "counts"
    @two_theta_indices = [0]
    counts:NX_INT32[31] = [1037, 1318, 1704, '...', 1321]
      @units = "counts"
    two_theta:NX_FLOAT64[31] = [17.92608, 17.92591, 17.92575, '...', 17.92108]
      @units = "degrees"
```

As the data files in these examples become more complex, you will appreciate the information density provided by *punx tree*.

downloads

The Python code and files related to this section may be downloaded from the following table.

file	description
`../simple_example.dat`	2-column ASCII data used in this section
`simple_example_write1.py`	python code to write example *simple_example_write1*
`simple_example_write1.hdf5`	NeXus file written by this code
`simple_example_write1_h5dump.txt`	*h5dump* analysis of the NeXus file
`simple_example_write1_structure.txt`	*punx tree* analysis of the NeXus file

Write a NeXus HDF5 file with plottable data

Building on the previous example, we wish to identify our measured data with the detector on the instrument where it was generated. In this hypothetical case, since the detector was positioned at some angle *two_theta*, we choose to store both datasets, `two_theta` and `counts`, in a NeXus group. One appropriate NeXus group is NXdetector. This group is placed in a NXinstrument group which is placed in a NXentry group. To support a default plot, we provide a NXdata group. Rather than duplicate the same data already placed in the detector group, we choose to link to those datasets from the `NXdata` group. (Compare the next figure with *Linking in a NeXus file* in the *NeXus Design* chapter of the NeXus User Manual.) The *NeXus Design* chapter provides a figure (*Linking in a NeXus file*) with a small variation

[1] *punx tree* : https://punx.readthedocs.io/en/latest/source_code/h5tree.html#how-to-use-h5tree

from our previous example, placing the measured data within the `/entry/instrument/detector` group. Links are made from that data to the `/entry/data` group.

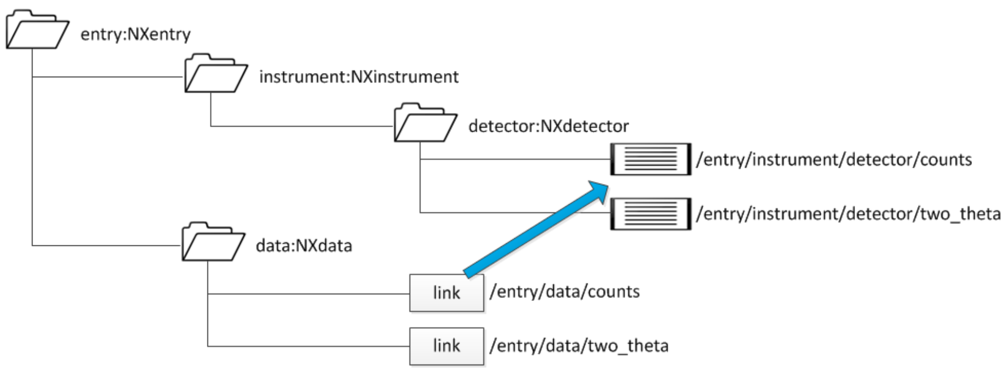


Fig. 2: h5py example showing linking in a NeXus file

The Python code to build an HDF5 data file with that structure (using numerical data from the previous example) is shown below.

```python
#!/usr/bin/env python
"""
Writes a simple NeXus HDF5 file using h5py with links
according to the example from Figure 2.1 in the Design chapter
"""

from pathlib import Path
import h5py
import numpy

filename = str(Path(__file__).absolute().parent.parent / "simple_example.dat")
buffer = numpy.loadtxt(filename).T
tthData = buffer[0]  # float[]
countsData = numpy.asarray(buffer[1], "int32")  # int[]

with h5py.File("simple_example_write2.hdf5", "w") as f:  # create the HDF5 NeXus file
    f.attrs["default"] = "entry"

    nxentry = f.create_group("entry")
    nxentry.attrs["NX_class"] = "NXentry"
    nxentry.attrs["default"] = "data"

    nxinstrument = nxentry.create_group("instrument")
    nxinstrument.attrs["NX_class"] = "NXinstrument"

    nxdetector = nxinstrument.create_group("detector")
    nxdetector.attrs["NX_class"] = "NXdetector"

    # store the data in the NXdetector group
    ds_tth = nxdetector.create_dataset("two_theta", data=tthData)
    ds_tth.attrs["units"] = "degrees"
    ds_counts = nxdetector.create_dataset("counts", data=countsData)
    ds_counts.attrs["units"] = "counts"
```

(continues on next page)

(continued from previous page)

```

    # create the NXdata group to define the default plot
    nxdata = nxentry.create_group("data")
    nxdata.attrs["NX_class"] = "NXdata"
    nxdata.attrs["signal"] = "counts"
    nxdata.attrs["axes"] = "two_theta"
    nxdata.attrs["two_theta_indices"] = [
        0,
    ]

    source_addr = "/entry/instrument/detector/two_theta"  # existing data
    target_addr = "two_theta"  # new location
    ds_tth.attrs["target"] = source_addr  # a NeXus API convention for links
    nxdata[target_addr] = f[source_addr]  # hard link
    # nxdata._id.link(source_addr, target_addr, h5py.h5g.LINK_HARD)

    source_addr = "/entry/instrument/detector/counts"  # existing data
    target_addr = "counts"  # new location
    ds_counts.attrs["target"] = source_addr  # a NeXus API convention for links
    nxdata[target_addr] = f[source_addr]  # hard link
    # nxdata._id.link(source_addr, target_addr, h5py.h5g.LINK_HARD)
```

It is interesting to compare the output of the `h5dump` of the data file `simple_example_write2.hdf5` with our Python instructions. See the *downloads* section below.

Look carefully! It *appears* in the output of `h5dump` that the actual data for `two_theta` and `counts` has *moved* into the `NXdata` group at HDF5 path `/entry/data`! But we stored that data in the `NXdetector` group at `/entry/instrument/detector`. This is normal for `h5dump` output.

A bit of explanation is necessary at this point. The data is not stored in either HDF5 group directly. Instead, HDF5 creates a `DATA` storage element in the file and posts a reference to that `DATA` storage element as needed. An HDF5 *hard link* requests another reference to that same `DATA` storage element. The `h5dump` tool describes in full that `DATA` storage element the first time (alphabetically) it is called. In our case, that is within the `NXdata` group. The next time it is called, within the `NXdetector` group, `h5dump` reports that a hard link has been made and shows the HDF5 path to the description.

NeXus recognizes this behavior of the HDF5 library and adds an additional structure when building hard links, the `target` attribute, to preserve the original location of the data. Not that it actually matters. the *punx tree* tool knows about the additional NeXus `target` attribute and shows the data to appear in its original location, in the `NXdetector` group.

```
@default = "entry"
entry:NXentry
  @NX_class = "NXentry"
  @default = "data"
  data:NXdata
    @NX_class = "NXdata"
    @axes = "two_theta"
    @signal = "counts"
    @two_theta_indices = [0]
    counts --> /entry/instrument/detector/counts
    two_theta --> /entry/instrument/detector/two_theta
  instrument:NXinstrument
    @NX_class = "NXinstrument"
```

(continues on next page)

(continued from previous page)

```
    detector:NXdetector
      @NX_class = "NXdetector"
      counts:NX_INT32[31] = [1037, 1318, 1704, '...', 1321]
        @target = "/entry/instrument/detector/counts"
        @units = "counts"
      two_theta:NX_FLOAT64[31] = [17.92608, 17.92591, 17.92575, '...', 17.92108]
        @target = "/entry/instrument/detector/two_theta"
        @units = "degrees"
```

downloads

The Python code and files related to this section may be downloaded from the following table.

file	description
`../simple_example.dat`	2-column ASCII data used in this section
`simple_example_write2.py`	python code to write example *simple_example_write2*
`simple_example_write2.hdf5`	NeXus file written by this code
`simple_example_write2_h5dump.txt`	*h5dump* analysis of the NeXus file
`simple_example_write2_structure.txt`	*punx tree* analysis of the NeXus file

Write a NeXuS HDF5 File with links to external data

HDF5 files may contain links to data (or groups) in other files. This can be used to advantage to refer to data in existing HDF5 files and create NeXus-compliant data files. Here, we show such an example, using the same `counts` v. `two_theta` data from the examples above.

We use the *HDF5 external file* links with NeXus data files.

```
f[local_addr] = h5py.ExternalLink(external_file_name, external_addr)
```

where `f` is an open `h5py.File()` object in which we will create the new link, `local_addr` is an HDF5 path address, `external_file_name` is the name (relative or absolute) of an existing HDF5 file, and `external_addr` is the HDF5 path address of the existing data in the `external_file_name` to be linked.

file: external_angles.hdf5

Take for example, the structure of `external_angles.hdf5`, a simple HDF5 data file that contains just the `two_theta` angles in an HDF5 dataset at the root level of the file. Although this is a valid HDF5 data file, it is not a valid NeXus data file:

```
angles:float64[31] = [17.926079999999999, '...', 17.92108]
  @units = degrees
```

file: external_counts.hdf5

The data in the file external_angles.hdf5 might be referenced from another HDF5 file (such as external_counts.hdf5) by an HDF5 external link.[1] Here is an example of the structure:

```
entry:NXentry
  instrument:NXinstrument
  detector:NXdetector
    counts:NX_INT32[31] = [1037, '...', 1321]
      @units = counts
    two_theta --> file="external_angles.hdf5", path="/angles"
```

file: external_master.hdf5

A valid NeXus data file could be created that refers to the data in these files without making a copy of the data files themselves.

Note: It is necessary for all these files to be located together in the same directory for the HDF5 external file links to work properly.`

To be a valid NeXus file, it must contain a NXentry group. For the files above, it is simple to make a master file that links to the data we desire, from structure that we create. We then add the group attributes that describe the default plottable data:

```
data:NXdata
  @signal = counts
  @axes = "two_theta"
  @two_theta_indices = 0
```

Here is (the basic structure of) external_master.hdf5, an example:

```
entry:NXentry
@default = data
  instrument --> file="external_counts.hdf5", path="/entry/instrument"
  data:NXdata
    @signal = counts
    @axes = "two_theta"
     @two_theta = 0
    counts --> file="external_counts.hdf5", path="/entry/instrument/detector/counts"
    two_theta --> file="external_angles.hdf5", path="/angles"
```

[1] see these URLs for further guidance on HDF5 external links: https://portal.hdfgroup.org/display/HDF5/H5L_CREATE_EXTERNAL, http://docs.h5py.org/en/stable/high/group.html#external-links

source code: external_example_write.py

Here is the complete code of a Python program, using h5py to write a NeXus-compliant HDF5 file with links to data in other HDF5 files.

external_example_write.py: Write using HDF5 external links

```
#!/usr/bin/env python
"""
Writes a NeXus HDF5 file using h5py with links to data in other HDF5 files.

This example is based on ``writer_2_1``.
"""

from pathlib import Path
import h5py
import numpy

FILE_HDF5_MASTER = "external_master.hdf5"
FILE_HDF5_ANGLES = "external_angles.hdf5"
FILE_HDF5_COUNTS = "external_counts.hdf5"

# ---------------------------

# get some data
filename = str(Path(__file__).absolute().parent.parent / "simple_example.dat")
buffer = numpy.loadtxt(filename).T
tthData = buffer[0]  # float[]
countsData = numpy.asarray(buffer[1], "int32")  # int[]

# put the angle data in an external (non-NeXus) HDF5 data file
with h5py.File(FILE_HDF5_ANGLES, "w") as f:
    ds = f.create_dataset("angles", data=tthData)
    ds.attrs["units"] = "degrees"

# put the detector counts in an external HDF5 data file
# with *incomplete* NeXus structure (no NXdata group)
with h5py.File(FILE_HDF5_COUNTS, "w") as f:
    nxentry = f.create_group("entry")
    nxentry.attrs["NX_class"] = "NXentry"
    nxinstrument = nxentry.create_group("instrument")
    nxinstrument.attrs["NX_class"] = "NXinstrument"
    nxdetector = nxinstrument.create_group("detector")
    nxdetector.attrs["NX_class"] = "NXdetector"
    ds = nxdetector.create_dataset("counts", data=countsData)
    ds.attrs["units"] = "counts"
    # link the "two_theta" data stored in separate file
    local_addr = nxdetector.name + "/two_theta"
    f[local_addr] = h5py.ExternalLink(FILE_HDF5_ANGLES, "/angles")

# create a master NeXus HDF5 file
with h5py.File(FILE_HDF5_MASTER, "w") as f:
```

(continues on next page)

(continued from previous page)

```
    f.attrs["default"] = "entry"
    nxentry = f.create_group("entry")
    nxentry.attrs["NX_class"] = "NXentry"
    nxentry.attrs["default"] = "data"
    nxdata = nxentry.create_group("data")
    nxdata.attrs["NX_class"] = "NXdata"

    # link in the signal data
    local_addr = "/entry/data/counts"
    external_addr = "/entry/instrument/detector/counts"
    f[local_addr] = h5py.ExternalLink(FILE_HDF5_COUNTS, external_addr)
    nxdata.attrs["signal"] = "counts"

    # link in the axes data
    local_addr = "/entry/data/two_theta"
    f[local_addr] = h5py.ExternalLink(FILE_HDF5_ANGLES, "/angles")
    nxdata.attrs["axes"] = "two_theta"
    nxdata.attrs["two_theta_indices"] = [
        0,
    ]

    local_addr = "/entry/instrument"
    f[local_addr] = h5py.ExternalLink(FILE_HDF5_COUNTS, "/entry/instrument")
```

downloads

The Python code and files related to this section may be downloaded from the following table.

file	description
`external_angles_h5dump.txt`	*h5dump* analysis of *external_angles.hdf5*
`external_angles.hdf5`	HDF5 file written by *external_example_write*
`external_angles_structure.txt`	*punx tree* analysis of *external_angles.hdf5*
`external_counts_h5dump.txt`	*h5dump* analysis of *external_counts.hdf5*
`external_counts.hdf5`	HDF5 file written by *external_example_write*
`external_counts_structure.txt`	*punx tree* analysis of *external_counts.hdf5*
`external_example_write.py`	python code to write external linking examples
`external_master_h5dump.txt`	*h5dump* analysis of *external_master.hdf5*
`external_master.hdf5`	NeXus file written by *external_example_write*
`external_master_structure.txt`	*punx tree* analysis of *external_master.hdf5*

Find plottable data in a NeXus HDF5 file

Let's make a new reader that follows the chain of attributes (`@default`, `@signal`, and `@axes`) to find the default plottable data. We'll use the same data file as the previous example. Our demo here assumes one-dimensional data. (For higher dimensionality data, we'll need more complexity when handling the `@axes` attribute and we'll to check the field sizes. See section *Find the plottable data*, subsection *Version 3*, for the details.)

reader_attributes_trail.py: Read a NeXus HDF5 file using Python with h5py

```python
from pathlib import Path
import h5py

filename = str(
    Path(__file__).absolute().parent.parent
    / "simple_example_basic"
    / "simple_example_basic.nexus.hdf5"
)
with h5py.File(filename, "r") as nx:
    # find the default NXentry group
    nx_entry = nx[nx.attrs["default"]]
    # find the default NXdata group
    nx_data = nx_entry[nx_entry.attrs["default"]]
    # find the signal field
    signal = nx_data[nx_data.attrs["signal"]]
    # find the axes field(s)
    attr_axes = nx_data.attrs["axes"]
    if isinstance(attr_axes, (set, tuple, list)):
        #  but check that attr_axes only describes 1-D data
        if len(attr_axes) == 1:
            attr_axes = attr_axes[0]
        else:
            raise ValueError(f"expected 1-D data but @axes={attr_axes}")
    axes = nx_data[attr_axes]

    print(f"file: {nx.filename}")
    print(f"signal: {signal.name}")
    print(f"axes: {axes.name}")
    print(f"{axes.name} {signal.name}")
    for x, y in zip(axes, signal):
        print(x, y)
```

Output from `reader_attributes_trail.py` is shown next.

Output from `reader_attributes_trail.py`

```
file: simple_example_basic.nexus.hdf5
signal: /entry/mr_scan/I00
axes: /entry/mr_scan/mr
/entry/mr_scan/mr /entry/mr_scan/I00
17.92608 1037
17.92591 1318
17.92575 1704
17.92558 2857
17.92541 4516
17.92525 9998
17.92508 23819
17.92491 31662
17.92475 40458
17.92458 49087
17.92441 56514
17.92425 63499
17.92408 66802
17.92391 66863
17.92375 66599
17.92358 66206
17.92341 65747
17.92325 65250
17.92308 64129
17.92291 63044
17.92275 60796
17.92258 56795
17.92241 51550
17.92225 43710
17.92208 29315
17.92191 19782
17.92175 12992
17.92158 6622
17.92141 4198
17.92125 2248
17.92108 1321
```

downloads

The Python code and files related to this section may be downloaded from the following table.

file	description
`reader_attributes_trail.py`	Read NeXus HDF5 file and find plotaable data

- Write examples for different NeXus classes

Example data used

The data shown plotted in the next figure will be written to the NeXus HDF5 file using only two NeXus base classes, `NXentry` and `NXdata`, in the first example and then minor variations on this structure in the next two examples. The data model is identical to the one in the *Introduction* chapter except that the names will be different, as shown below:

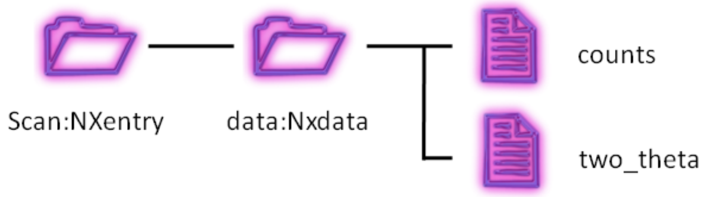


Fig. 3: data structure of the simple example

```
/entry:NXentry
    /mr_scan:NXdata
       /mr : float64[31]
       /I00 : int32[31]
```

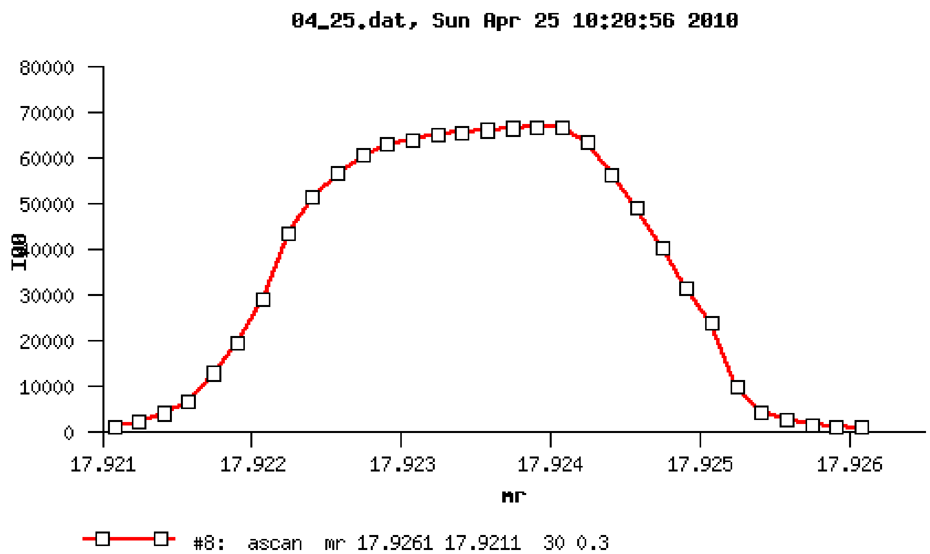


Fig. 4: plot of the simple example data

Simple example values

```
17.92608    1037
17.92591    1318
17.92575    1704
17.92558    2857
17.92541    4516
17.92525    9998
17.92508    23819
17.92491    31662
17.92475    40458
17.92458    49087
17.92441    56514
17.92425    63499
17.92408    66802
17.92391    66863
17.92375    66599
17.92358    66206
17.92341    65747
17.92325    65250
17.92308    64129
17.92291    63044
17.92275    60796
17.92258    56795
17.92241    51550
17.92225    43710
17.92208    29315
17.92191    19782
17.92175    12992
17.92158    6622
17.92141    4198
17.92125    2248
17.92108    1321
```

2.1.3 HDF5 in Python with `nexusformat`

`nexusformat` provides a higher level API on top of h5py (see chapter *HDF5 in Python with h5py*). While h5py provides a basic API to read and write HDF5 files, `nexusformat` enriches this API with NeXus specific utilities.

Please refer to the `NeXpy` documentation: https://nexpy.github.io/nexpy/.

Code examples

- Write examples for different NeXus classes

2.1.4 HDF5 in MATLAB

author
Paul Kienzle, NIST

Note: Editor's Note: These files were copied directly from an older version of the NeXus documentation (DocBook) and have not been checked that they will run under current Matlab versions.

input.dat

This is the same data used with *HDF5 in Python with h5py*.

```
17.92608    1037
17.92591    1318
17.92575    1704
17.92558    2857
17.92541    4516
17.92525    9998
17.92508    23819
17.92491    31662
17.92475    40458
17.92458    49087
17.92441    56514
17.92425    63499
17.92408    66802
17.92391    66863
17.92375    66599
17.92358    66206
17.92341    65747
17.92325    65250
17.92308    64129
17.92291    63044
17.92275    60796
17.92258    56795
17.92241    51550
17.92225    43710
17.92208    29315
17.92191    19782
17.92175    12992
17.92158    6622
17.92141    4198
17.92125    2248
17.92108    1321
```

writing data

basic_writer.m: Write a NeXus HDF5 file using Matlab

```
% Writes a NeXus HDF5 file using matlab

disp 'Write a NeXus HDF5 file'
filename = 'prj_test.nexus.hdf5';
timestamp = '2010-10-18T17:17:04-0500';

% read input data
A = load('input.dat');
mr = A(:,1);
I00 = int32(A(:,2));

% clear out old file, if it exists

delete(filename);

% using the simple h5 interface, there is no way to create a group without
% first creating a dataset; creating the dataset creates all intervening
% groups.

% store x
h5create(filename,'/entry/mr_scan/mr',[length(mr)]);
h5write(filename,'/entry/mr_scan/mr',mr);
h5writeatt(filename,'/entry/mr_scan/mr','units','degrees');
h5writeatt(filename,'/entry/mr_scan/mr','long_name','USAXS mr (degrees)');

% store y
h5create(filename,'/entry/mr_scan/I00',[length(I00)],'DataType','int32');
h5write(filename,'/entry/mr_scan/I00',I00);
h5writeatt(filename,'/entry/mr_scan/I00','units','counts');
h5writeatt(filename,'/entry/mr_scan/I00','long_name','USAXS I00 (counts)');

% indicate that we are plotting y vs. x
h5writeatt(filename,'/','default','entry');
h5writeatt(filename,'/entry','default','mr_scan');
h5writeatt(filename,'/entry/mr_scan','signal','I00');
h5writeatt(filename,'/entry/mr_scan','axes','mr_scan');
h5writeatt(filename,'/entry/mr_scan','mr_scan_indices', int32(0));

% add NeXus metadata
h5writeatt(filename,'/','file_name',filename);
h5writeatt(filename,'/','file_time',timestamp);
h5writeatt(filename,'/','instrument','APS USAXS at 32ID-B');
h5writeatt(filename,'/','creator','basic_writer.m');
h5writeatt(filename,'/','NeXus_version','4.3.0');
h5writeatt(filename,'/','HDF5_Version','1.6'); % no 1.8 features used in this example
h5writeatt(filename,'/entry','NX_class','NXentry');
h5writeatt(filename,'/entry/mr_scan','NX_class','NXdata');


```

(continues on next page)

(continued from previous page)

```
h5disp(filename);
```

reading data

basic_reader.m: Read a NeXus HDF5 file using Matlab

```
% Reads NeXus HDF5 file and print the contents

filename = 'prj_test.nexus.hdf5';
root = h5info(filename,'/');
attrs = root.Attributes;
for i = 1:length(attrs)
    fprintf('%s: %s\n', attrs(i).Name, attrs(i).Value);
end
mr = h5read(filename,'/entry/mr_scan/mr');
i00 = h5read(filename, '/entry/mr_scan/I00');
fprintf('#\t%s\t%s\n','mr','I00');
for i = 1:length(mr)
    fprintf('%d\t%g\t%d\n', i, mr(i), i00(i));
end
```

writing data file with links

writer_2_1.m: Write a NeXus HDF5 file with links

```
% Writes a simple NeXus HDF5 file with links
% according to the example from Figure 2.1 in the Design chapter

filename = 'writer_2_1.hdf5';

% read input data
A = load('input.dat');
two_theta = A(:,1);
counts = int32(A(:,2));

% clear out old file, if it exists
delete(filename);

% store x
h5create(filename,'/entry/instrument/detector/two_theta',[length(two_theta)]);
h5write(filename,'/entry/instrument/detector/two_theta',two_theta);
h5writeatt(filename,'/entry/instrument/detector/two_theta','units','degrees');

% store y
h5create(filename,'/entry/instrument/detector/counts',[length(counts)],'DataType','int32
↪');
h5write(filename,'/entry/instrument/detector/counts',counts);
h5writeatt(filename,'/entry/instrument/detector/counts','units','counts');

```

(continues on next page)

(continued from previous page)

```
% create group NXdata with links to detector
% note: requires the additional file h5link.m
h5link(filename,'/entry/instrument/detector/two_theta','/entry/data/two_theta');
h5link(filename,'/entry/instrument/detector/counts','/entry/data/counts');

% indicate that we are plotting y vs. x
h5writeatt(filename,'/','default','entry');
h5writeatt(filename,'/entry','default','data');
h5writeatt(filename,'/entry/data','signal','counts');
h5writeatt(filename,'/entry/data','axes','two_theta');
h5writeatt(filename,'/entry/data','two_theta_indices',int32(0));

% add NeXus metadata
h5writeatt(filename,'/','file_name',filename);
h5writeatt(filename,'/','file_time',timestamp);
h5writeatt(filename,'/','instrument','APS USAXS at 32ID-B');
h5writeatt(filename,'/','creator','writer_2_1.m');
h5writeatt(filename,'/','NeXus_version','4.3.0');
h5writeatt(filename,'/','HDF5_Version','1.6'); % no 1.8 features used in this example
h5writeatt(filename,'/entry','NX_class','NXentry');
h5writeatt(filename,'/entry/instrument','NX_class','NXinstrument');
h5writeatt(filename,'/entry/instrument/detector','NX_class','NXdetector');
h5writeatt(filename,'/entry/data','NX_class','NXdata');

% show structure of the file that was created
h5disp(filename);
```

h5link.m: support module for creating NeXus-style HDF5 hard links

```
function h5link(filename, from, to)
%H5LINK Create link to an HDF5 dataset.
%   H5LINK(FILENAME,SOURCE,TARGET) creates an HDF5 link from the
%   dataset at location SOURCE to a dataset at location TARGET.  All
%   intermediate groups in the path to target are created.
%
%   Example:  create a link from /hello/world to /goodbye/world
%      h5create('myfile.h5','/hello/world',[100 200]);
%      h5link('myfile.h5','/hello/world','/goodbye/world');
%      hgdisp('myfile.h5');
%
%   See also: h5create, h5read, h5write, h5info, h5disp

% split from and to into group/dataset
idx = strfind(from,'/');
from_path = from(1:idx(end)-1);
from_data = from(idx(end)+1:end);
idx = strfind(to,'/');
to_path = to(1:idx(end)-1);
to_data = to(idx(end)+1:end);

```

(continues on next page)

(continued from previous page)

```
% open the HDF file
fid = H5F.open(filename,'H5F_ACC_RDWR','H5P_DEFAULT');

% create target group if it doesn't already exist
create_intermediate = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(create_intermediate, 1);
try
    H5G.create(fid,to_path,create_intermediate,'H5P_DEFAULT','H5P_DEFAULT');
catch
end
H5P.close(create_intermediate);

% open groups and create link
from_id = H5G.open(fid, from_path);
to_id = H5G.open(fid, to_path);
H5L.create_hard(from_id, from_data, to_id, to_data, 'H5P_DEFAULT','H5P_DEFAULT');

% close all
H5G.close(from_id);
H5G.close(to_id);
H5F.close(fid);
end
```

Downloads

file	description
`input.dat`	two-column text data file, also used in other examples
`basic_writer.m`	writes a NeXus HDF5 file using `input.dat`
`basic_reader.m`	reads the NeXus HDF5 file written by `basic_writer.m`
`h5link.m`	support module for creating NeXus-style HDF5 hard links
`writer_2_1.m`	like `basic_writer.m` but stores data in `/entry/instrument/detector` and then links to **NXdata** group

2.1.5 HDF5 in C with NAPI

Code examples are provided in this section that write 2-D data to a NeXus HDF5 file in the C language using the *NAPI: NeXus Application Programmer Interface (frozen)*.

The following code reads a two-dimensional set `counts` with dimension scales of `t` and `phi` using local routines, and then writes a NeXus file containing a single `NXentry` group and a single `NXdata` group. This is the simplest data file that conforms to the NeXus standard.

NAPI C Example: write simple NeXus file

Note: This example uses the signal/axes attributes applied to the data field, as described in *Associating plottable data by name using the axes attribute*. New code should use the method described in *Associating plottable data using attributes applied to the NXdata group*.

```c
#include "napi.h"

int main()
{
    int counts[50][1000], n_t=1000, n_p=50, dims[2], i;
    float t[1000], phi[50];
    NXhandle file_id;
/*
 * Read in data using local routines to populate phi and counts
 *
 * for example you may create a getdata() function and call
 *
 *      getdata (n_t, t, n_p, phi, counts);
 */
/* Open output file and output global attributes */
    NXopen ("NXfile.nxs", NXACC_CREATE5, &file_id);
      NXputattr (file_id, "user_name", "Joe Bloggs", 10, NX_CHAR);
/* Open top-level NXentry group */
      NXmakegroup (file_id, "Entry1", "NXentry");
      NXopengroup (file_id, "Entry1", "NXentry");
/* Open NXdata group within NXentry group */
        NXmakegroup (file_id, "Data1", "NXdata");
        NXopengroup (file_id, "Data1", "NXdata");
/* Output time channels */
          NXmakedata (file_id, "time_of_flight", NX_FLOAT32, 1, &n_t);
          NXopendata (file_id, "time_of_flight");
            NXputdata (file_id, t);
            NXputattr (file_id, "units", "microseconds", 12, NX_CHAR);
          NXclosedata (file_id);
/* Output detector angles */
          NXmakedata (file_id, "polar_angle", NX_FLOAT32, 1, &n_p);
          NXopendata (file_id, "polar_angle");
            NXputdata (file_id, phi);
            NXputattr (file_id, "units", "degrees", 7, NX_CHAR);
          NXclosedata (file_id);
/* Output data */
          dims[0] = n_t;
          dims[1] = n_p;
          NXmakedata (file_id, "counts", NX_INT32, 2, dims);
          NXopendata (file_id, "counts");
            NXputdata (file_id, counts);
            i = 1;
            NXputattr (file_id, "signal", &i, 1, NX_INT32);
            NXputattr (file_id, "axes",  "polar_angle:time_of_flight", 26, NX_CHAR);
          NXclosedata (file_id);
```

(continues on next page)

(continued from previous page)

```
/* Close NXentry and NXdata groups and close file */
        NXclosegroup (file_id);
      NXclosegroup (file_id);
    NXclose (&file_id);
    return;
}
```

2.1.6 HDF5 in Fortran with NAPI

Code examples are provided in this section that write 2-D data to a NeXus HDF5 file in F77, and F90 languages using the *NAPI: NeXus Application Programmer Interface (frozen)*.

The following code reads a two-dimensional set `counts` with dimension scales of `t` and `phi` using local routines, and then writes a NeXus file containing a single `NXentry` group and a single `NXdata` group. This is the simplest data file that conforms to the NeXus standard.

NAPI F77 Example: write simple NeXus file

Note: The F77 interface is no longer being developed.

```
      program WRITEDATA

      include 'NAPIF.INC'
      integer*4 status, file_id(NXHANDLESIZE), counts(1000,50), n_p, n_t, dims(2)
      real*4 t(1000), phi(50)

!Read in data using local routines
      call getdata (n_t, t, n_p, phi, counts)
!Open output file
      status = NXopen ('NXFILE.NXS', NXACC_CREATE, file_id)
        status = NXputcharattr
     +          (file_id, 'user', 'Joe Bloggs', 10, NX_CHAR)
!Open top-level NXentry group
        status = NXmakegroup (file_id, 'Entry1', 'NXentry')
        status = NXopengroup (file_id, 'Entry1', 'NXentry')
!Open NXdata group within NXentry group
          status = NXmakegroup (file_id, 'Data1', 'NXdata')
          status = NXopengroup (file_id, 'Data1', 'NXdata')
!Output time channels
            status = NXmakedata
     +          (file_id, 'time_of_flight', NX_FLOAT32, 1, n_t)
            status = NXopendata (file_id, 'time_of_flight')
              status = NXputdata (file_id, t)
              status = NXputcharattr
     +          (file_id, 'units', 'microseconds', 12, NX_CHAR)
            status = NXclosedata (file_id)
!Output detector angles
            status = NXmakedata (file_id, 'polar_angle', NX_FLOAT32, 1, n_p)
            status = NXopendata (file_id, 'polar_angle')
```

(continues on next page)

(continued from previous page)

```
              status = NXputdata (file_id, phi)
              status = NXputcharattr (file_id, 'units', 'degrees', 7, NX_CHAR)
            status = NXclosedata (file_id)
!Output data
            dims(1) = n_t
            dims(2) = n_p
            status = NXmakedata (file_id, 'counts', NX_INT32, 2, dims)
            status = NXopendata (file_id, 'counts')
              status = NXputdata (file_id, counts)
              status = NXputattr (file_id, 'signal', 1, 1, NX_INT32)
              status = NXputattr
     +          (file_id, 'axes', 'polar_angle:time_of_flight', 26, NX_CHAR)
            status = NXclosedata (file_id)
!Close NXdata and NXentry groups and close file
          status = NXclosegroup (file_id)
        status = NXclosegroup (file_id)
      status = NXclose (file_id)

      stop
      end
```

NAPI F90 Example: write simple NeXus file

Note: This example uses the signal/axes attributes applied to the data field, as described in *Associating plottable data by name using the axes attribute*. New code should use the method described in *Associating plottable data using attributes applied to the NXdata group*.

```
program WRITEDATA

   use NXUmodule

   type(NXhandle) :: file_id
   integer, pointer :: counts(:,:)
   real, pointer :: t(:), phi(:)

!Use local routines to allocate pointers and fill in data
   call getlocaldata (t, phi, counts)
!Open output file
   if (NXopen ("NXfile.nxs", NXACC_CREATE, file_id) /= NX_OK) stop
   if (NXUwriteglobals (file_id, user="Joe Bloggs") /= NX_OK) stop
!Set compression parameters
   if (NXUsetcompress (file_id, NX_COMP_LZW, 1000) /= NX_OK) stop
!Open top-level NXentry group
   if (NXUwritegroup (file_id, "Entry1", "NXentry") /= NX_OK) stop
   !Open NXdata group within NXentry group
      if (NXUwritegroup (file_id, "Data1", "NXdata") /= NX_OK) stop
   !Output time channels
         if (NXUwritedata (file_id, "time_of_flight", t, "microseconds") /= NX_OK) stop
   !Output detector angles
```

(continues on next page)

(continued from previous page)

```
        if (NXUwritedata (file_id, "polar_angle", phi, "degrees") /= NX_OK) stop
   !Output data
        if (NXUwritedata (file_id, "counts", counts, "counts") /= NX_OK) stop
           if (NXputattr (file_id, "signal", 1) /= NX_OK) stop
           if (NXputattr (file_id, "axes", "polar_angle:time_of_flight") /= NX_OK) stop
   !Close NXdata group
      if (NXclosegroup (file_id) /= NX_OK) stop
!Close NXentry group
   if (NXclosegroup (file_id) /= NX_OK) stop
!Close NeXus file
   if (NXclose (file_id) /= NX_OK) stop

end program WRITEDATA
```

2.1.7 HDF5 in Python with NAPI

A single code example is provided in this section that writes 3-D data to a NeXus HDF5 file in the Python language using the *NAPI: NeXus Application Programmer Interface (frozen)*.

The data to be written to the file is a simple three-dimensional array (2 x 3 x 4) of integers. The single dataset is intended to demonstrate the order in which each value of the array is stored in a NeXus HDF5 data file.

NAPI Python Example: write simple NeXus file

```
#!/usr/bin/python

import sys
import nxs
import numpy

a = numpy.zeros((2,3,4),dtype=numpy.int)
val = 0
for i in range(2):
    for j in range(3):
        for k in range(4):
            a[i,j,k] = val
            val = val + 1

nf = nxs.open("simple3D.h5", "w5")

nf.makegroup("entry","NXentry")
nf.opengroup("entry","NXentry")

nf.makegroup("data","NXdata")
nf.opengroup("data","NXdata")
nf.putattr("signal","test")

nf.makedata("test",'int32',[2,3,4])
nf.opendata("test")
nf.putdata(a)
```

(continues on next page)

(continued from previous page)

```
nf.closedata()

nf.closegroup() # NXdata
nf.closegroup() # NXentry

nf.close()

exit
```

2.2 Visualization tools

Tools to visualize NeXus HDF5 files graphically or in text form.

2.2.1 View a NeXus HDF5 file with *h5dump*

The `h5dump` tool[1] provided as part of the HDF5 tool kit[2] can be used to print the content of an HDF5 file. As an example we show the result of the command `h5dump simple3D.h5` on the result of *HDF5 in Python with NAPI*

```
HDF5 "simple3D.h5" {
GROUP "/" {
   ATTRIBUTE "NeXus_version" {
      DATATYPE  H5T_STRING {
            STRSIZE 5;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SCALAR
      DATA {
      (0): "4.1.0"
      }
   }
   ATTRIBUTE "file_name" {
      DATATYPE  H5T_STRING {
            STRSIZE 11;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SCALAR
      DATA {
      (0): "simple3D.h5"
      }
   }
   ATTRIBUTE "HDF5_Version" {
      DATATYPE  H5T_STRING {
            STRSIZE 5;
```

(continues on next page)

[1] **h5dump** : https://support.hdfgroup.org/HDF5/doc/RM/Tools.html#Tools-Dump

[2] **HDF5 tools** : https://support.hdfgroup.org/products/hdf5_tools/

(continued from previous page)

```
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SCALAR
      DATA {
      (0): "1.6.6"
      }
   }
   ATTRIBUTE "file_time" {
      DATATYPE  H5T_STRING {
            STRSIZE 24;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SCALAR
      DATA {
      (0): "2011-11-18 17:26:27+0100"
      }
   }
   GROUP "entry" {
      ATTRIBUTE "NX_class" {
         DATATYPE  H5T_STRING {
               STRSIZE 7;
               STRPAD H5T_STR_NULLTERM;
               CSET H5T_CSET_ASCII;
               CTYPE H5T_C_S1;
            }
         DATASPACE  SCALAR
         DATA {
         (0): "NXentry"
         }
      }
      GROUP "data" {
         ATTRIBUTE "NX_class" {
            DATATYPE  H5T_STRING {
                  STRSIZE 6;
                  STRPAD H5T_STR_NULLTERM;
                  CSET H5T_CSET_ASCII;
                  CTYPE H5T_C_S1;
               }
            DATASPACE  SCALAR
            DATA {
            (0): "NXdata"
            }
         }
         DATASET "test" {
            DATATYPE  H5T_STD_I32LE
            DATASPACE  SIMPLE { ( 2, 3, 4 ) / ( 2, 3, 4 ) }
            DATA {
            (0,0,0): 0, 1, 2, 3,
```

(continues on next page)

(continued from previous page)

```
            (0,1,0): 4, 5, 6, 7,
            (0,2,0): 8, 9, 10, 11,
            (1,0,0): 12, 13, 14, 15,
            (1,1,0): 16, 17, 18, 19,
            (1,2,0): 20, 21, 22, 23
            }
            ATTRIBUTE "signal" {
               DATATYPE  H5T_STD_I32LE
               DATASPACE  SCALAR
               DATA {
               (0): 1
               }
            }
         }
      }
   }
}
}
```

2.2.2 View a NeXus HDF5 file with *punx tree*

The `punx tree` tool[1] provided as part of `punx`[2] can be used to print the content of an HDF5 file. As an example we show the result of the command `punx tree simple3D.h5` on the result of *HDF5 in Python with NAPI*

```
simple3D.h5:NeXus data file
  @NeXus_version = 4.1.0
  @file_name = simple3D.h5
  @HDF5_Version = 1.6.6
  @file_time = 2011-11-18 17:26:27+0100
  entry:NXentry
    @NX_class = NXentry
    data:NXdata
      @NX_class = NXdata
      test:NX_INT32[2,3,4] = __array
        @signal = 1
        __array = [
            [
              [0, 1, 2, 3]
              [4, 5, 6, 7]
              [8, 9, 10, 11]
            ]
            [
              [12, 13, 14, 15]
              [16, 17, 18, 19]
              [20, 21, 22, 23]
            ]
          ]
```

[1] **punx tree** : https://punx.readthedocs.io/en/latest/source_code/h5tree.html#how-to-use-h5tree
[2] **punx** : https://punx.readthedocs.io/

2.2.3 Plot a NeXus HDF5 file with *NeXpy*

A NeXus HDF5 file with plottable data (see *Find plottable data in a NeXus HDF5 file*) can be plotted by `NeXpy`[1].

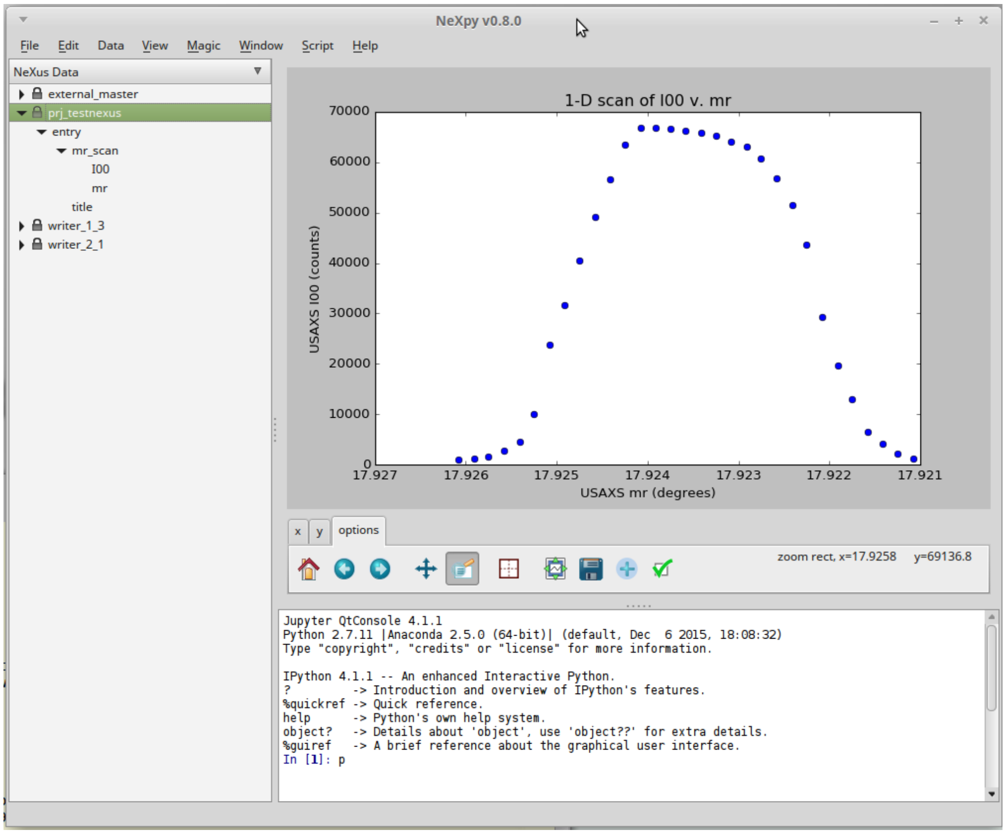


Fig. 5: plot the simple example using NeXpy

Compare this with *plot of the simple example data* and note that the horizontal axis of this plot is mirrored from that above. This is because the data is stored in the file in descending `mr` order and `NeXpy` has plotted it that way (in order of appearance) by default.

2.2.4 Plot a NeXus HDF5 file with *silx view*

A NeXus HDF5 file with plottable data (see *Find plottable data in a NeXus HDF5 file*) can be plotted by the `silx view`[1] tool provided as part of `silx`[2].

[1] *NeXpy*: http://nexpy.github.io/nexpy/

[1] **silx view** : http://www.silx.org/doc/silx/latest/applications/view.html

[2] **silx** : http://www.silx.org/doc/silx/latest/

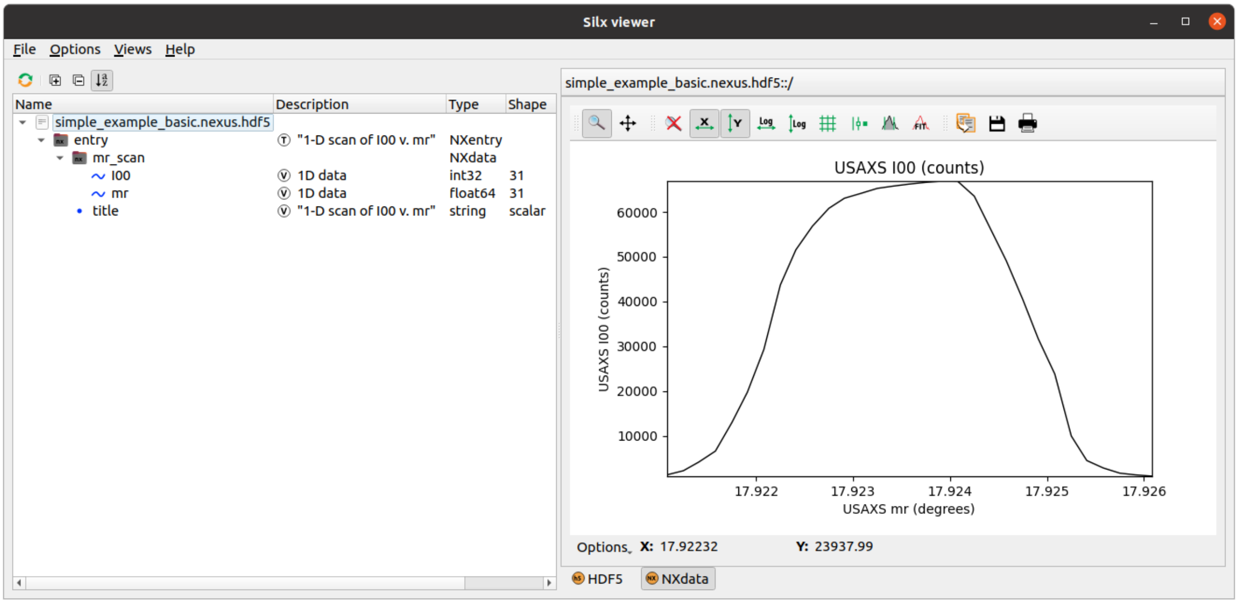


Fig. 6: plot the simple example using silx

2.3 Examples for Specific Instruments

Examples of working with data from specific instruments.

2.3.1 Viewing 2-D Data from LRMECS

The IPNS LRMECS instrument stored data in NeXus HDF4 data files. One such example is available from the repository of NeXus data file examples.[1] For this example, we will start with a conversion of that original data file into *HDF5* format.

format	file name
HDF4	lrcs3701.nxs
HDF5	lrcs3701.nx5

This dataset contains two histograms with 2-D images (148x750 and 148x32) of 32-bit integers. First, we use the `h5dump` tool to investigate the header content of the file (not showing any of the data).

Visualize Using `h5dump`

Here, the output of the command:

```
h5dump -H lrcs3701.nx5
```

has been edited to only show the first *NXdata* group (`/Histogram1/data`):

[1] LRMECS example data: https://github.com/nexusformat/exampledata/tree/master/IPNS/LRMECS

LRMECS `lrcs3701` data: `h5dump` output

```
HDF5 "C:\Users\Pete\Documents\eclipse\NeXus\definitions\exampledata\IPNS\LRMECS\lrcs3701.
↪nx5" {
GROUP "/Histogram1/data" {
   DATASET "data" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 148, 750 ) / ( 148, 750 ) }
   }
   DATASET "polar_angle" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 148 ) / ( 148 ) }
   }
   DATASET "time_of_flight" {
      DATATYPE  H5T_IEEE_F32LE
      DATASPACE  SIMPLE { ( 751 ) / ( 751 ) }
   }
   DATASET "title" {
      DATATYPE  H5T_STRING {
            STRSIZE 44;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         }
      DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   }
}
}
```

Visualize Using *HDFview*

For many, the simplest way to view the data content of an HDF5 file is to use the *HDFview* program (https://portal.hdfgroup.org/display/HDFVIEW/HDFView) from The HDF Group. After starting *HDFview*, the data file may be loaded by dragging it into the main HDF window. On opening up to the first `NXdata` group */Histogram1/data* (as above), and then double-clicking the dataset called: *data*, we get our first view of the data.

The data may be represented as an image by accessing the *Open As* menu from HDFview (on Windows, right click the dataset called *data* and select the *Open As* item, consult the HDFview documentation for different platform instructions). Be sure to select the *Image* radio button, and then (accepting everything else as a default) press the *Ok* button.

Note: In this image, dark represents low intensity while white represents high intensity.

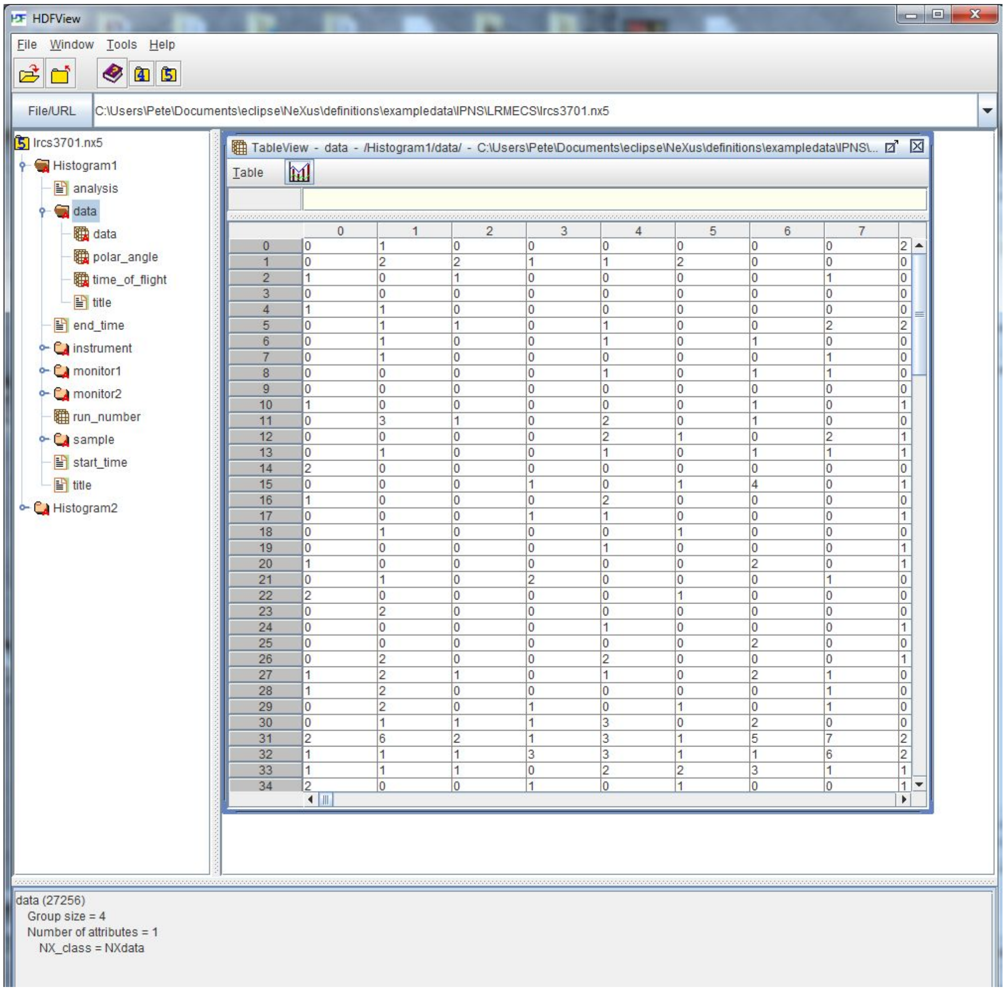


Fig. 7: LRMECS `lrcs3701` data: *HDFview*

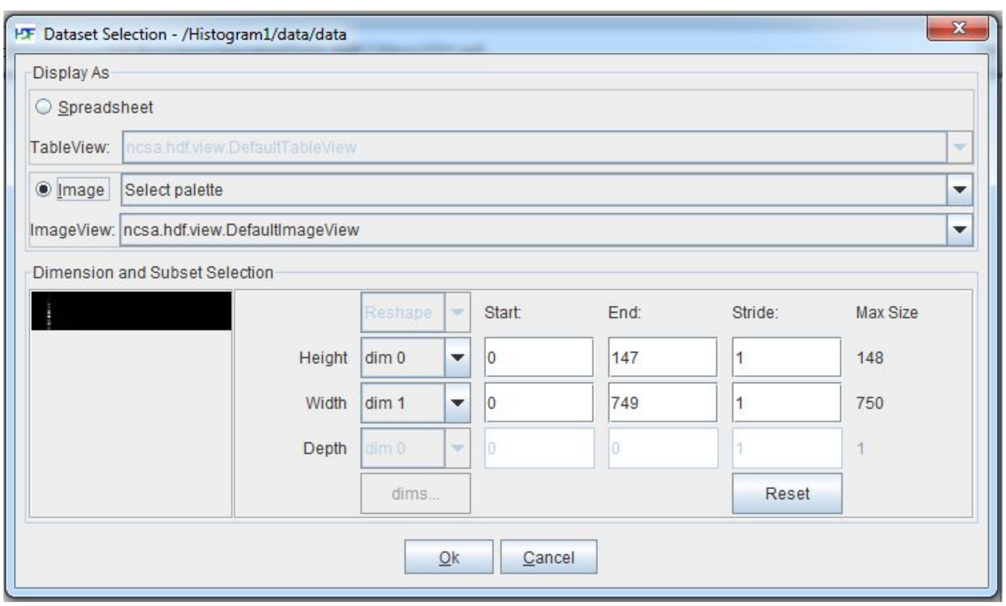


Fig. 8: LRMECS `lrcs3701` data: *HDFview Open As* dialog

LRMECS `lrcs3701` data: image

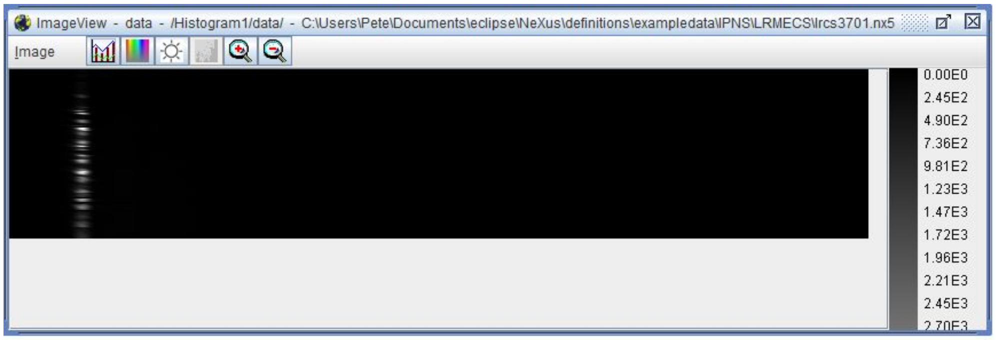


Fig. 9: LRMECS `lrcs3701` data: *HDFview* Image

Visualize Using *IgorPro*

Another way to visualize this data is to use a commercial package for scientific data visualization and analysis. One such package is *IgorPro* from http://www.wavemetrics.com

IgorPro provides a browser for HDF5 files that can open our NeXus HDF5 and display the image. Follow the instructions from WaveMetrics to install the *HDF5 Browser* package: http://www.wavemetrics.com/products/igorpro/dataaccess/hdf5.htm

You may not have to do this step if you have already installed the *HDF5 Browser*. IgorPro will tell you if it is not installed properly. To install the *HDF5 Browser*, first start *IgorPro*. Next, select from the menus and submenus: `Data; Load Waves; Packages; Install HDF5 Package` as shown in the next figure. IgorPro may direct you to perform more activities before you progress from this step.

Next, open the *HDF5 Browser* by selecting from the menus and submenus: `Data; Load Waves; New HDF5 Browser` as shown in the next figure.

Next, click the *Open HDF5 File* button and open the NeXus HDF5 file `lrcs3701.nxs`. In the lower left *Groups* panel, click the *data* dataset. Also, under the panel on the right called *Load Dataset Options*, choose `No Table` as shown. Finally, click the *Load Dataset* button (in the *Datasets* group) to display the image.

Note: In this image, dark represents low intensity while white represents high intensity. The image has been rotated for easier representation in this manual.

LRMECS `lrcs3701` data: image

..Example-EPICS:

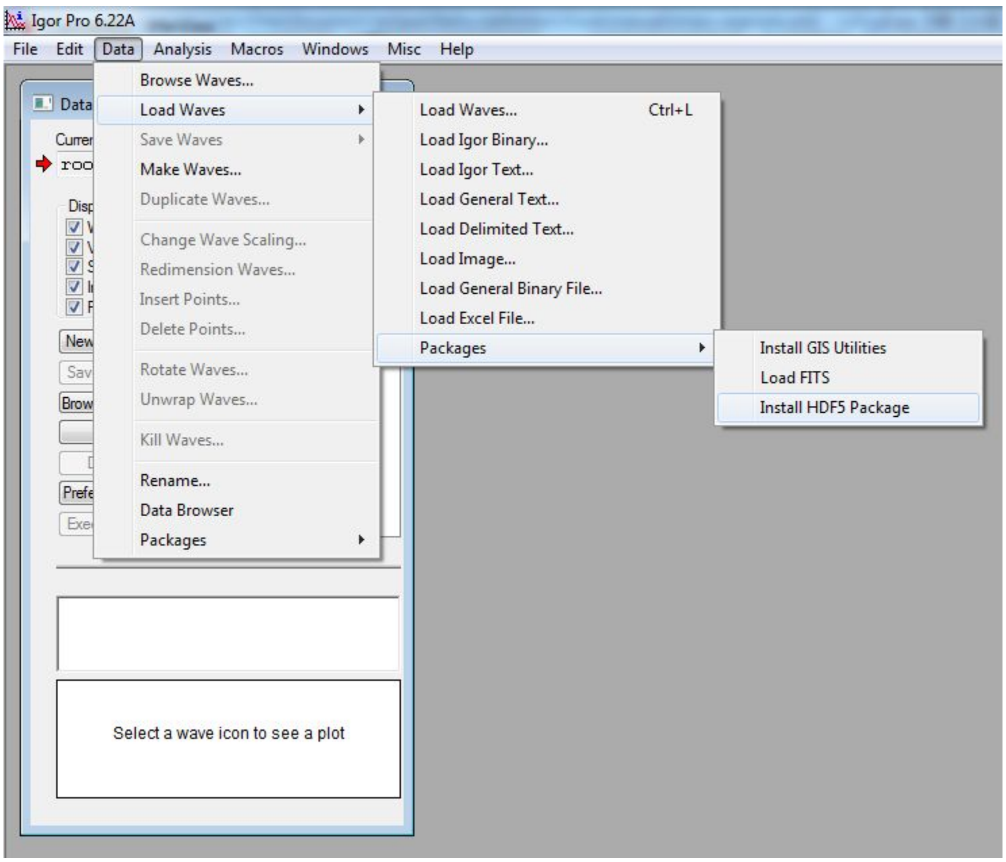


Fig. 10: LRMECS `lrcs3701` data: *IgorPro* install HDF5 Browser

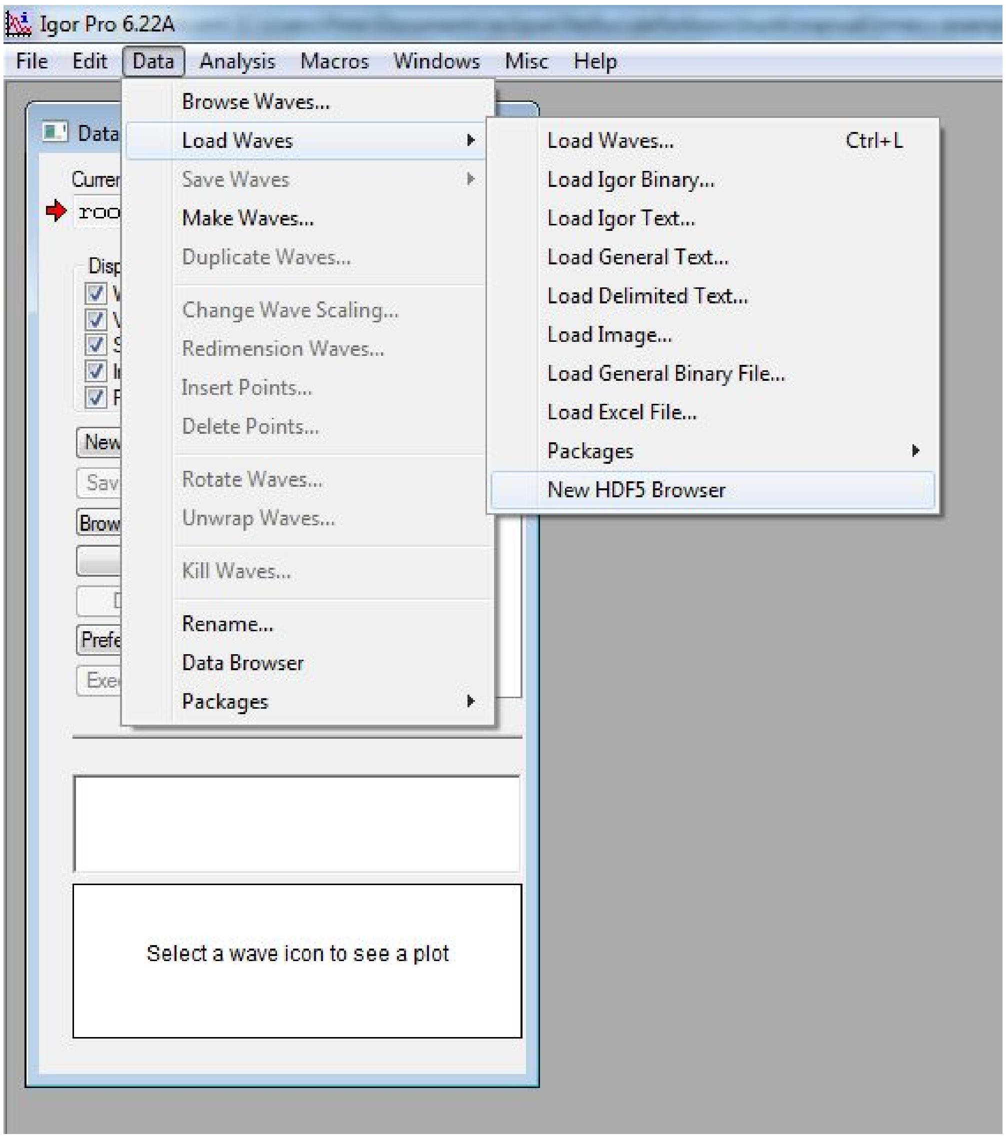


Fig. 11: LRMECS `lrcs3701` data: *IgorPro HDFBrowser* dialog

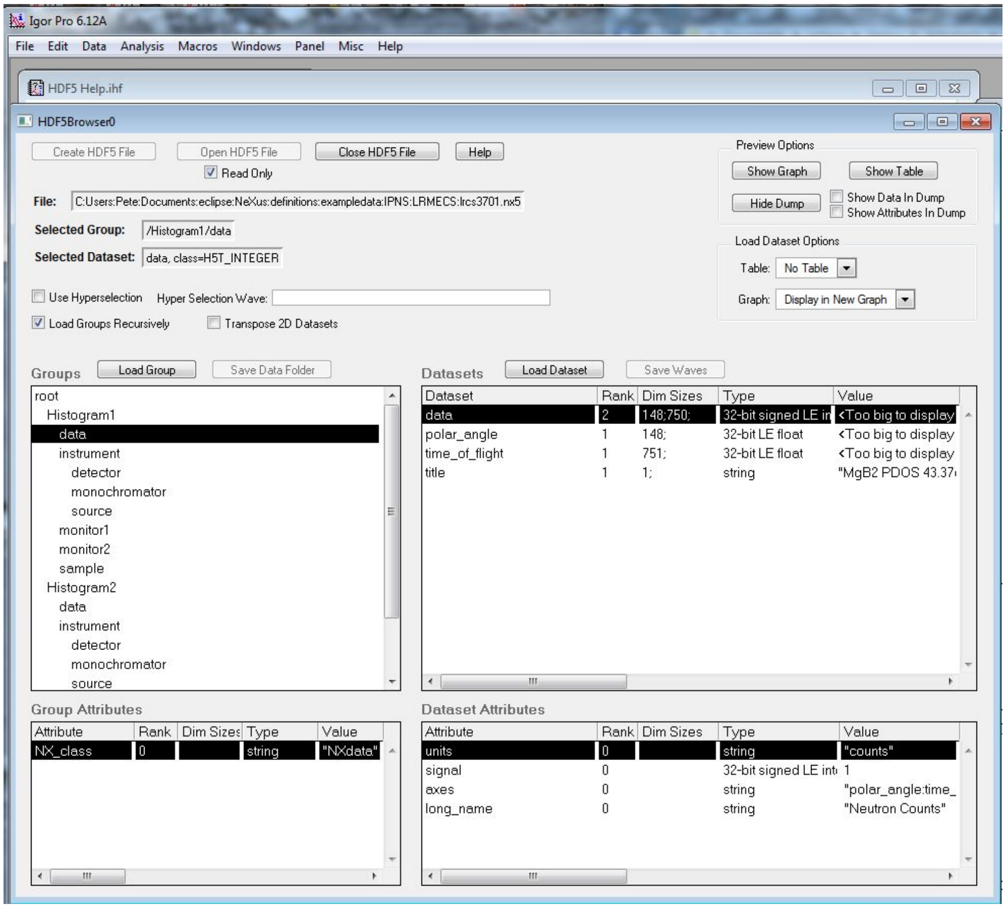


Fig. 12: LRMECS `lrcs3701` data: *IgorPro HDFBrowser* dialog

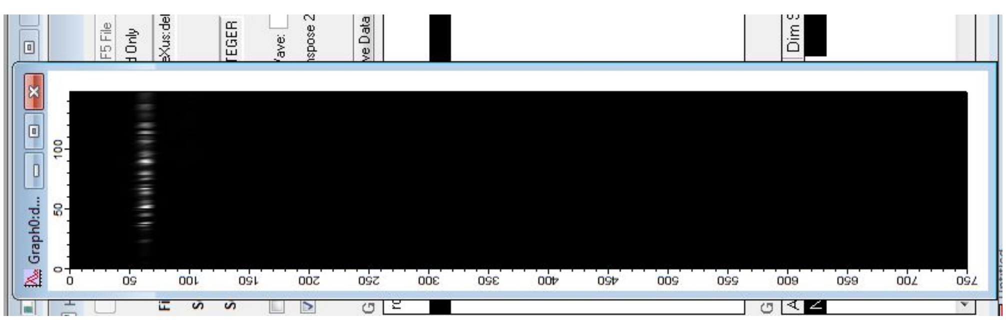


Fig. 13: LRMECS `lrcs3701` data: *IgorPro* Image

2.3.2 EPICS Area Detector Examples

Two examples in this section show how to write NeXus HDF5 data files with EPICS Area Detector images. The first shows how to configure the HDF5 File Writing Plugin of the EPICS Area Detector software. The second example shows how to write an EPICS Area Detector image using Python.

HDF5 File Writing Plugin

This example describes how to write a NeXus HDF5 data file using the EPICS[1] Area Detector[2] HDF5 file writing plugin[3]. We will use the EPICS SimDetector[4] as an example. (PV prefix: `13SIM1:`) Remember to replace that with the prefix for your detector's IOC.

One data file will be produced for each image generated by EPICS.

You'll need AreaDetector version 2.5 or higher to use this as the procedures for using the HDF5 file writing plugin changed with this release.

configuration files

There are two configuration files we must edit to configure an EPICS AreaDetector to write NeXus files using the HDF5 File Writer plugin:

file	description
`attributes.xml`	what information to know about from EPICS and other sources
`layout.xml`	where to write that information in the HDF5 file

Put these files into a known directory where your EPICS IOC can find them.

`attributes.xml`

The attributes file is easy to edit. Any text editor will do. A wide screen will be helpful.

Each `<Attribute />` element declares a single **ndattribute** which is associated with an area detector image. These **ndattribute** items can be written to specific locations in the HDF5 file or placed by default in a *default location*.

Note: The attributes file shown here has been reformatted for display in this manual. The *downloads* section below provides an attributes file with the same content using its wide formatting (one complete Attribute per line). Either version of this file is acceptable.

```xml
<?xml version="1.0" standalone="no" ?>
<!-- Attributes -->
<Attributes
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
    "https://github.com/areaDetector/ADCore/blob/master/iocBoot/NDAttributes.xsd"
    >
```

(continues on next page)

[1] EPICS: https://epics-controls.org/
[2] EPICS Area Detector: https://areadetector.github.io/master/index.html
[3] HDF5 File Writer: https://areadetector.github.io/master/ADCore/NDFileHDF5.html
[4] EPICS SimDetector: https://github.com/areaDetector/ADSimDetector

(continued from previous page)

```
    <Attribute name="AcquireTime"
               type="EPICS_PV"
               source="13SIM1:cam1:AcquireTime"
               dbrtype="DBR_NATIVE"
               description="Camera acquire time"/>
    <Attribute name="ImageCounter"
               type="PARAM"
               source="ARRAY_COUNTER"
               datatype="INT"
               description="Image counter"/>
    <Attribute name="calc1_val"
               type="EPICS_PV"
               source="prj:userCalc1.VAL"
               datatype="DBR_NATIVE"
               description="some calculation result"/>
    <Attribute name="calc2_val"
               type="EPICS_PV"
               source="prj:userCalc2.VAL"
               datatype="DBR_NATIVE"
               description="another calculation result"/>
    <Attribute name="MaxSizeX"
               type="PARAM"
               source="MAX_SIZE_X"
               datatype="INT"
               description="Detector X size"/>
    <Attribute name="MaxSizeY"
               type="PARAM"
               source="MAX_SIZE_Y"
               datatype="INT"
               description="Detector Y size"/>
    <Attribute name="CameraModel"
               type="PARAM"
               source="MODEL"
               datatype="STRING"
               description="Camera model"/>
    <Attribute name="CameraManufacturer"
               type="PARAM"
               source="MANUFACTURER"
               datatype="STRING"
               description="Camera manufacturer"/>
</Attributes>
```

If you want to add additional EPICS process variables (PVs) to be written in the HDF5 file, create additional `<Attribute />` elements (such as the `calc1_val`) and modify the `name`, `source`, and `description` values. Be sure to use a unique **name** for each **ndattribute** in the attributes file.

Note: ndattribute : item specified by an `<Attribute />` element in the attributes file.

layout.xml

You might not need to edit the layout file. It will be fine (at least a good starting point) as it is, even if you add PVs (a.k.a. *ndattribute*) to the attributes.xml file.

```
<?xml version="1.0" standalone="no" ?>
<hdf5_layout>
  <group name="entry">
    <attribute name="NX_class" source="constant" value="NXentry" type="string"/>
    <group name="instrument">
      <attribute name="NX_class" source="constant" value="NXinstrument" type="string"/>
      <group name="detector">
        <attribute name="NX_class" source="constant" value="NXdetector" type="string"/>
        <dataset name="data" source="detector" det_default="true">
          <attribute name="NX_class" source="constant" value="SDS" type="string"/>
          <attribute name="signal" source="constant" value="1" type="int"/>
          <attribute name="target" source="constant" value="/entry/instrument/detector/data" type="string"/>
        </dataset>
        <group name="NDAttributes">
          <attribute name="NX_class" source="constant" value="NXcollection" type="string"/>
          <dataset name="ColorMode" source="ndattribute" ndattribute="ColorMode"/>
        </group>          <!-- end group NDAttribute -->
      </group>            <!-- end group detector -->
      <group name="NDAttributes" ndattr_default="true">
        <attribute name="NX_class" source="constant" value="NXcollection" type="string"/>
      </group>            <!-- end group NDAttribute (default) -->
      <group name="performance">
        <dataset name="timestamp" source="ndattribute"/>
      </group>            <!-- end group performance -->
    </group>              <!-- end group instrument -->
    <group name="data">
      <attribute name="NX_class" source="constant" value="NXdata" type="string"/>
      <hardlink name="data" target="/entry/instrument/detector/data"/>
      <!-- The "target" attribute in /entry/instrument/detector/data is used to
           tell Nexus utilities that this is a hardlink -->
    </group>              <!-- end group data -->
  </group>                <!-- end group entry -->
</hdf5_layout>
```

If you do not specify where in the file to write an *ndattribute* from the attributes file, it will be written within the group that has `ndattr_default="true"`. This identifies the group to the HDF5 file writing plugin as the *default location* to store content from the attributes file. In the example layout file, that *default location* is the `/entry/instrument/NDAttributes` group:

```
<group
      name="NDAttributes"
      ndattr_default="true">
  <attribute
      name="NX_class"
      source="constant"
      value="NXcollection"
      type="string"/>
```

(continues on next page)

(continued from previous page)

```
</group>
```

To specify where PVs are written in the HDF5 file, you must create `<dataset />` (or `<attribute />`) elements at the appropriate place in the NeXus HDF5 file layout. See the NeXus manual[5] for placement advice if you are unsure.

You reference each *ndattribute* by its `name` value from the attributes file and use it as the value of the `ndattribute` in the layout file. In this example, `ndattribute="calc1_val"` in the layout file references `name="calc1_val"` in the attributes file and will be identified in the HDF5 file by the name `userCalc1`:

```
<dataset
   name="userCalc1"
   source="ndattribute"
   ndattribute="calc1_val"/>
```

Note: A value from the attributes file is only written either in the *default location* or in the location named by a `<dataset/>` or `<attribute/>` entry in the layout file. Expect problems if you define the same *ndattribute* in more than one place in the layout file.

You can control when a value is written to the file, using `when=""` in the layout file. This can be set to one of these values: `OnFileOpen`, `OnFileClose`

Such as:

```
<dataset
   name="userCalc1"
   source="ndattribute"
   ndattribute="calc1_val"
   when="OnFileOpen"/>
```

or:

```
<attribute
   name="exposure_s"
   source="ndattribute"
   ndattribute="AcquireTime"
   when="OnFileClose"/>
```

additional configuration

Additional configurations of the EPICS Area Detector and the HDF5 File Plugin are done using the EPICS screens (shown here using caQtDM[6]):

Additional configuration on the **ADBase** screen:

- Set *Image mode* to "Single"
- Set *Exposure time* as you wish
- Set # *Images* to 1
- for testing, it is common to bin the data to reduce the image size

[5] NeXus manual: https://manual.nexusformat.org/
[6] caQtDM: http://epics.web.psi.ch/software/caqtdm/

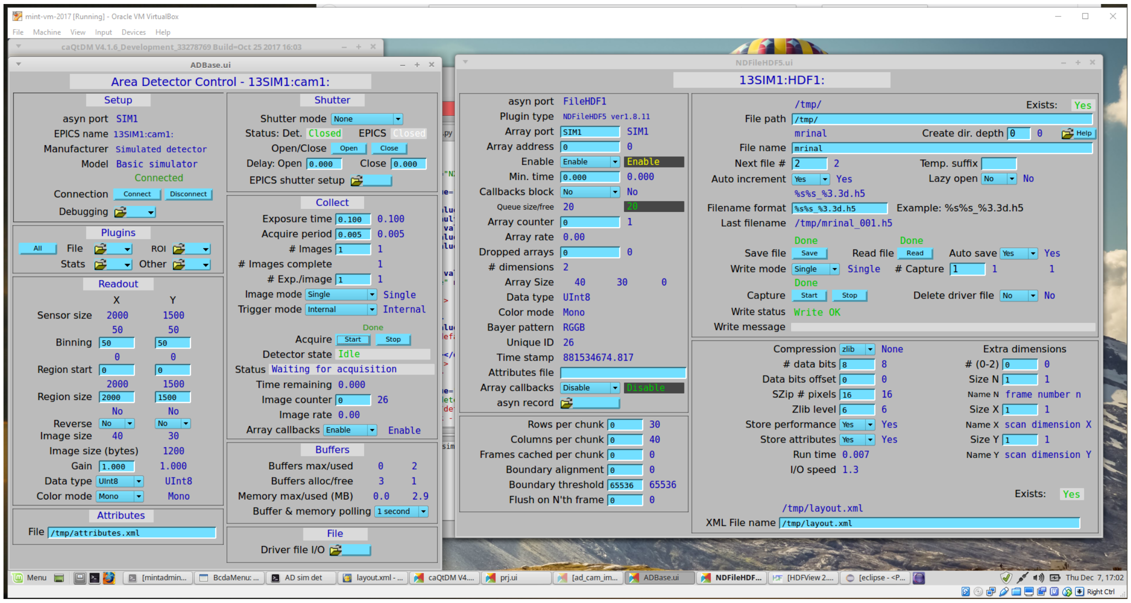


Fig. 14: **ADBase** and **NDFileHDF5** configuration screens

- The full path to the `attributes.xml` file goes in the bottom/left **File** box

Additional configuration on the **NDFileHDF5** screen:

- Set the **File path** and "File name" to your choice.
- Set **Auto save** to "Yes".
- Set **Compression** to "zlib" if you wish (optional)
- Set **Enable** to "Enable" or the HDF5 plugin won't get images to write!
- Set **Callbacks block** to "Yes" if you want to wait for HDF5 files to finish writing before collecting the next image
- The full path to the `layout.xml` file goes into the bottom/right **XML File name** box
- Leave the **Attributes file** box empty in this screen.

When you enter the names of these files in the configuration screen boxes, AreaDetector will check the files for errors and let you know.

Example view

We collected data for one image, `/tmp/mrinal_001.h5`, in the HDF5 file provided in the **downloads** section. You may notice that the values for `calc1_val` and `calc2_val` were arrays rather than single values. That was due to an error in the original `attributes.xml` file, which had `type="PARAM"` instead of `type="EPICS_PV"`. This has been fixed in the `attributes.xml` file presented here.

Python code to store an image in a NeXus file

Suppose you want to write area detector images into NeXus HDF5 files python code. Let's assume you have the image already in memory in a numpy array, perhaps from reading a TIFF file or from an EPICS PV using PyEpics. The file `write_nexus_file.py` (provided below) reads an image from the sim detector and writes it to a NeXus HDF5 data file, along with some additional metadata.

using the *h5py* package

This example uses the *h5py*[7] package to write the HDF5 file.

```python
import numpy as np
import h5py
import datetime

def write_nexus_file(fname, image, md={}):
    """
    write the image to a NeXus HDF5 data file

    Parameters
    ----------
    fname : str
        name of the file (relative or absolute) to be written
    image : numpy array
        the image data
    md : dictionary
        key: value where value is something that can be written by h5py
             (such as str, int, float, numpy array, ...)
    """
    nexus = h5py.File(fname, "w")
    nexus.attrs["filename"] = fname
    nexus.attrs["file_time"] = datetime.datetime.now().astimezone().isoformat()
    nexus.attrs["creator"] = "write_nexus_file()"
    nexus.attrs["H5PY_VERSION"] = h5py.__version__

    # /entry
    nxentry = nexus.create_group("entry")
    nxentry.attrs["NX_class"] = "NXentry"
    nexus.attrs["default"] = nxentry.name

    # /entry/instrument
    nxinstrument = nxentry.create_group("instrument")
    nxinstrument.attrs["NX_class"] = "NXinstrument"

    # /entry/instrument/detector
    nxdetector = nxinstrument.create_group("detector")
    nxdetector.attrs["NX_class"] = "NXdetector"

    # /entry/instrument/detector/image
    ds = nxdetector.create_dataset("image", data=image, compression="gzip")
    ds.attrs["units"] = "counts"
```

(continues on next page)

[7] h5py: http://docs.h5py.org

(continued from previous page)

```
    ds.attrs["target"] = "/entry/instrument/detector/image"

    # /entry/data
    nxdata = nxentry.create_group("data")
    nxdata.attrs["NX_class"] = "NXdata"
    nxentry.attrs["default"] = nxdata.name

    # /entry/data/data --> /entry/instrument/detector/image
    nxdata["data"] = nexus["/entry/instrument/detector/image"]
    nxdata.attrs["signal"] = "data"

    if len(md) > 0:
        # /entry/instrument/metadata (optional, for metadata)
        metadata = nxinstrument.create_group("metadata")
        metadata.attrs["NX_class"] = "NXcollection"
        for k, v in md.items():
            try:
                metadata.create_dataset(k, data=v)
            except Exception:
                metadata.create_dataset(k, data=str(v))

    nexus.close()


if __name__ == "__main__":
    """demonstrate how to use this code"""
    import epics
    prefix = "13SIM1:"
    img = epics.caget(prefix+"image1:ArrayData")
    size_x = epics.caget(prefix+"cam1:ArraySizeX_RBV")
    size_y = epics.caget(prefix+"cam1:ArraySizeY_RBV")
    # edit the full image for just the binned data
    img = img[:size_x*size_y].reshape((size_x, size_y))

    extra_information = dict(
        unique_id = epics.caget(prefix+"image1:UniqueId_RBV"),
        size_x = size_x,
        size_y = size_y,
        detector_state = epics.caget(prefix+"cam1:DetectorState_RBV"),
        bitcoin_value="15000",
    )
    write_nexus_file("example.h5", img, md=extra_information)
```

The output from that code is given in the example.h5 file. It has this tree structure:

```
example.h5 : NeXus data file
  @H5PY_VERSION = "3.6.0"
  @creator = "write_nexus_file()"
  @default = "entry"
  @file_time = "2022-03-07 14:34:04.418733"
  @filename = "example.h5"
  entry:NXentry
```

(continues on next page)

(continued from previous page)

```
    @NX_class = "NXentry"
    @default = "data"
    data:NXdata
      @NX_class = "NXdata"
      @signal = "data"
      data --> /entry/instrument/detector/image
    instrument:NXinstrument
      @NX_class = "NXinstrument"
      detector:NXdetector
        @NX_class = "NXdetector"
        image:NX_UINT8[1024,1024] = __array
          __array = [
              [76, 77, 78, '...', 75]
              [77, 78, 79, '...', 76]
              [78, 79, 80, '...', 77]
              ...
              [75, 76, 77, '...', 74]
            ]
          @target = "/entry/instrument/detector/image"
          @units = "counts"
      metadata:NXcollection
        @NX_class = "NXcollection"
        bitcoin_value:NX_CHAR = b'15000'
        detector_state:NX_INT64[] =
        size_x:NX_INT64[] =
        size_y:NX_INT64[] =
        unique_id:NX_INT64[] =
```

Note: Alternatively, the metadata shown in this example might be placed in the `/entry/instrument/detector` (*NXdetector*) group along with the image `data` since it provides image-related information such as size.

In the interest of keeping this example simpler and similar to the one above using the HDF5 File Writing Plugin, the metadata has been written into a *NXcollection* group at `/entry/instrument/metadata` location. (Compare with the *NXcollection* group `/entry/instrument/NDAttributes` above.)

using the *nexusformat* package

The *nexusformat*[8] package for python simplifies the work to create a NeXus file. Rewriting the above code using *nexusformat*:

```python
import numpy as np
from nexusformat.nexus import *


def write_nexus_file(fname, image, md={}):
    """
    write the image to a NeXus HDF5 data file

```

(continues on next page)

[8] nexusformat: This Python package is described on the NeXPy web site

(continued from previous page)

```
    Parameters
    ----------
    fname : str
        name of the file (relative or absolute) to be written
    image : numpy array
        the image data
    md : dictionary
        key: value where value is something that can be written by h5py
             (such as str, int, float, numpy array, ...)
    """
    nx = NXroot()
    nx['/entry'] = NXentry(NXinstrument(NXdetector()))
    nx['entry/instrument/detector/image'] = NXfield(image, units='counts',
                                                    compression='gzip')
    nx['entry/data'] = NXdata()
    nx['entry/data'].makelink(nx['entry/instrument/detector/image'])
    nx['entry/data'].nxsignal = nx['entry/data/image']

    if len(md) > 0:
        # /entry/instrument/metadata (optional, for metadata)
        metadata = nx['/entry/instrument/metadata'] = NXcollection()
        for k, v in md.items():
            metadata[k] = v

    nx.save(fname, 'w')


if __name__ == "__main__":
    """demonstrate how to use this code"""
    import epics
    prefix = "13SIM1:"
    img = epics.caget(prefix+"image1:ArrayData")
    size_x = epics.caget(prefix+"cam1:ArraySizeX_RBV")
    size_y = epics.caget(prefix+"cam1:ArraySizeY_RBV")
    # edit the full image for just the binned data
    img = img[:size_x*size_y].reshape((size_x, size_y))

    extra_information = dict(
        unique_id = epics.caget(prefix+"image1:UniqueId_RBV"),
        size_x = size_x,
        size_y = size_y,
        detector_state = epics.caget(prefix+"cam1:DetectorState_RBV"),
        bitcoin_value="15000",
    )
    write_nexus_file("example.h5", img, md=extra_information)
```

Visualization

You can visualize the HDF5 files with several programs, such as: hdfview[9], nexpy[10], or pymca[11]. Views of the test image shown using **NeXPy** (from the HDF5 file) and **caQtDM** (the image from EPICS) are shown.

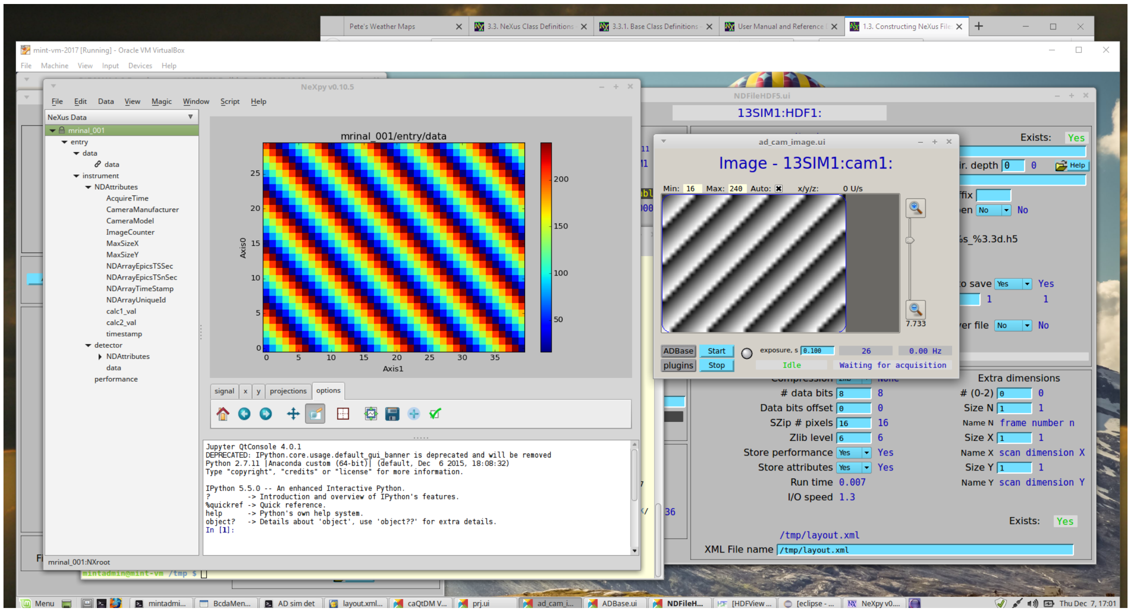


Fig. 15: Views of the image in **NeXPy** (left) and in **caQtDM** (right)

Get the installation instructions for any of these programs from a web search. Other data analysis programs such as MatLab, IgorPro, and IDL can also read HDF5 files but you might have to work a bit more to get the data to a plot.

Downloads

file	description
`attributes.xml`	The attributes file
`layout.xml`	The layout file
`mrinal_001.h5`	example NeXus HDF5 file written from EPICS
`write_nexus_file.py`	Python code to get images from EPICS and write a NeXus file
`write_nexus_file2.py`	*write_nexus_file.py* rewritten with *nexusformat* package
`example.h5`	example NeXus HDF5 file written from Python

[9] hdfview: https://support.hdfgroup.org/products/java/hdfview/
[10] nexpy: https://nexpy.github.io/nexpy/
[11] pymca: http://pymca.sourceforge.net/

Footnotes

2.4 Other tools to handle NeXus data files

The number of tools that read NeXus data files, either for general use or to read a specific application definition, is growing. Many of these are open source and so also serve as code examples. In the section *NeXus Utilities*, we describe many applications and software packages that can read, write, browse, and use NeXus data files. Examples of code (mostly from the NeXus community) that read NeXus data are listed in section *Language APIs for NeXus and HDF5*.

The NIAC welcomes your continued contributions to this documentation.

CHAPTER

THREE

NEXUS: REFERENCE DOCUMENTATION

3.1 Introduction to NeXus definitions

While the design principles of NeXus are explained in the *NeXus: User Manual*, this Reference Documentation specifies all allowed base classes and all standardized application definitions. Furthermore, it also contains contributed definitions of new bases classes or application definitions that are currently under review.

Base class definitions and application definitions have basically the same structure, but different semantics:

- Base class definitions define the *complete* set of terms that *might* be used in an instance of that class.
- Application definitions define the *minimum* set of terms that *must* be used in an instance of that class.

Base classes and application definitions are specified using a domain-specific XML scheme, the *NXDL: The NeXus Definition Language*.

3.1.1 Overview of NeXus definitions

For each class definition, the documentation is derived from content provided in the NXDL specification.

The documentation for each class consists of sections describing the *Status*, *Description*, table of *Symbols* (if defined), other NeXus base class *Groups cited*, an annotated *Structure*, and a link to the *NXDL Source* (XML) file.

Each of the NXDL files has its own tag in the version repository. Such as *NXcrystal-1.0* is tagged in GiHub and accessible via URL: https://github.com/nexusformat/definitions/releases/tag/NXcrystal-1.0

Description

General documentation if this NXDL file.

Symbols table

The `Symbols` table describes keywords used in this NXDL file to designate array dimensions. For reasons of avoiding naming collisions and to facilitate readbility and comprehension for those whom are new to an NXDL file, the following guidelines are strongly encouraged:

- All symbols used in the application definition are defined in a single `Symbols` table.
- The *name* of a symbol uses camel case without any white space or underscores.

  examples:

  nP: Total number of scan points

  nE: Number of photon energies scanned

  nFrames: Number of frames

  detectorRank: Rank of data array provided by the detector for a single measurement

- the `Symbols` table appears early in the .nxdl file above the `NXentry` group

  example from NXtomo.nxdl.xml

```
<definition name="NXtomo" extends="NXobject" type="group"
  category="application"
  xmlns="http://definition.nexusformat.org/nxdl/3.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://definition.nexusformat.org/nxdl/3.1 ../nxdl.xsd"
>
	<symbols>
		<doc>
			These symbols will be used below to coordinate datasets with the␣
↪same shape.
		</doc>
		<symbol name="nFrames">
			<doc>Number of frames</doc>
		</symbol>
		<symbol name="xSize">
			<doc>Number of pixels in X direction</doc>
		</symbol>
		<symbol name="ySize">
			<doc>Number of pixels in Y direction</doc>
		</symbol>
	</symbols>
	<doc>
		This is the application definition for x-ray or neutron tomography raw␣
↪data.

		In tomography
		a number of dark field images are measured, some bright field images and,
↪ of course the sample.
		In order to distinguish between them images carry a image_key.
```

(continues on next page)

(continued from previous page)

```
        </doc>
        <group type="NXentry" name="entry">
        <field name="title"  minOccurs="0" maxOccurs="1"/>
...
```

Annotated Structure

A representation of the basic structure (groups, fields, dimensions, attributes, and links) is prepared for each NXDL specification. Indentation shows nested structure. Attributes are prepended with the `@` symbol. Links use the characters `->` to represent the path to the intended source of the information.

Indentation is used to indicate nesting of subgroups (a feature common to application definitions). Within each indentation level, NeXus *fields* are listed first in the order presented in the NXDL file, then *groups*. *Attributes* are listed after the documentation of each item and are prefixed with the letter `@` (do not use the `@` symbol in the actual attribute name). The name of each item is in **bold**, followed by either *optional* or *required* and then the NXDL base class name (for groups) or the NeXus data type (for fields). If units are to be provided with the *field*, the type of the units is described, such as `NX_DATE_TIME`.

NeXus Links (these specifications are typically present only in application definitions) are described by a local name, the text `->`, then a suggested path to the source item to be linked to the local name.

Names (groups, fields, links, and attributes)

Name of the item. Since `name` needs to be restricted to valid program variable names, no "`-`" characters can be allowed. Name must satisfy both HDF and XML naming.

```
    NameStartChar ::=  _ | a..z | A..Z
    NameChar      ::=  NameStartChar | 0..9
    Name          ::=  NameStartChar (NameChar)*

    Or, as a regular expression:    [_a-zA-Z][_a-zA-Z0-9]*
    equivalent regular expression:  [_a-zA-Z][\w_]*
```

Attributes, identified with a leading "at" symbol (`@`) and belong with the preceding field or group, are additional metadata used to define this field or group. In the example above, the `program_name` element has the `configuration` (optional) attribute while the `thumbnail` element has the `mime_type` (optional) attribute.

For groups, the name may not be declared in the NXDL specification. In such instances, the *value shown in parentheses* in the *Name and Attributes* column is a suggestion, obtained from the group by removing the "NX" prefix. See NXentry for examples.

When the name is allowed to be *flexible* (the exact name given by this NXDL specification is not required but is set at the time the HDF file is written), the flexible part of the name will be written in all capital letters. For example, in the NXdata group, the `DATA`, `VARIABLE`, and `VARIABLE_errors` fields are *flexible*.

NeXus data type

Type of data to be represented by this variable. The type is one of those specified in *NXDL: The NeXus Definition Language*. In the case where the variable can take only one value from a known list, the list of known values is presented, such as in the `target_material` field above: `Ta | W | depleted_U | enriched_U | Hg | Pb | C`. Selections with included whitespace are surrounded by quotes. See the example above for usage.

For fields, the data type may not be specified in the NXDL file. The *default data type* is `NX_CHAR`. See NXdata for examples.

Units

Data units, are given as character strings, must conform to the NeXus *units standard*. See the *NeXus units* section for details.

Description

A simple text description of the field. No markup or formatting is allowed.

NXDL element type	minOccurs	maxOccurs
group	[1]	unbounded
field	[1]	unbounded
attribute	[1]	1

Choice

The `choice` element allows one to create a group with a defined name that is one specific NXDL base class from a defined list of possibilities

In some cases when creating an application definition, more than one choice of base class might be used to define a particular subgroup. For this particular situation, the `choice` was added to the NeXus NXDL Schema.

In this example fragment of an NXDL application definition, the `pixel_shape` could be represented by *either* `NXoff_geometry` or `NXcylindrical_geometry`.

```xml
<choice name="pixel_shape">
  <group type="NXoff_geometry">
    <doc>
      Shape description of each pixel. Use only if all pixels in the detector
      are of uniform shape.
    </doc>
  </group>
  <group type="NXcylindrical_geometry">
    <doc>
      Shape description of each pixel. Use only if all pixels in the detector
      are of uniform shape and require being described by cylinders.
    </doc>
  </group>
</choice>
```

[1] For NXDL *base classes*, `minOccurs=0` is the default, for NXDL *application definitions* and *contributed definitions*, `minOccurs=1` is the default. In all cases, the `minOccurs` attribute in the NXDL file will override the default for that element (group, field, attribute, or link).

The `@name` attribute of the `choice` element specifies the name that will appear in the HDF5 data file using one of the groups listed within the choice. Thus, it is not necessary to specify the name in each group. (At some point, the NXDL Schema may be modified to enforce this rule.)

A `choice` element may be used wherever a `group` element is used. It **must** have at least two groups listed (otherwise, it would not be useful).

3.2 NXDL: The NeXus Definition Language

Information in NeXus data files is arranged by a set of rules. These rules facilitate the exchange of data between scientists and software by standardizing common terms such as the way engineering units are described and the names for common things and the way that arrays are described and stored.

The set of rules for storing information in NeXus data files is declared using the NeXus Definition Language. NXDL itself is governed by a set of rules (a *schema*) that should simplify learning the few terms in NXDL. In fact, the NXDL rules, written as an XML Schema, are machine-readable using industry-standard and widely-available software tools for XML files such as `xsltproc` and `xmllint`. This chapter describes the rules and terms from which NXDL files are constructed.

3.2.1 Introduction

NeXus Definition Language (NXDL) files allow scientists to define the nomenclature and arrangement of information in NeXus data files. These NXDL files can be specific to a scientific discipline such as tomography or small-angle scattering, specific analysis or data reduction software, or even to define another component (base class) used to design and build NeXus data files.

In addition to this chapter and the *Tutorial* chapter, look at the set of NeXus NXDL files to learn how to read and write NXDL files. These files are available from the NeXus *definitions* repository and are most easily viewed on GitHub: https://github.com/nexusformat/definitions in the `base_classes`, `applications`, and `contributed` directories. The rules (expressed as XML Schema) for NXDL files may also be viewed from this URL. See the files `nxdl.xsd` for the main XML Schema and `nxdlTypes.xsd` for the listings of allowed data types and categories of units allowed in NXDL files.

NXDL files can be checked (validated) for syntax and content. With validation, scientists can be certain their definitions will be free of syntax errors. Since NXDL is based on the XML standard, there are many editing programs[1] available to ensure that the files are *well-formed*.[2] There are many standard tools such as `xmllint` and `xsltproc` that can process XML files. Further, NXDL files are backed by a set of rules (an *XML Schema*) that define the language and can be used to check that an NXDL file is both correct by syntax and valid by the NeXus rules.

NXDL files are machine-readable. This enables their automated conversion into schema files that can be used, in combination with other NXDL files, to validate NeXus data files. In fact, all of the tables in the *Class Definitions* Chapter have been generated directly from the NXDL files.

Writing references and anchors in the documentation.

Tip: Use the reST anchors when writing documentation in NXDL source files. Since the anchors have no title or caption associated, you will need to supply text with the reference, such as:

```
:ref:`this text will appear <anchor>`
```

[1] For example *XML Copy Editor* (http://xml-copy-editor.sourceforge.net/)
[2] http://en.wikipedia.org/wiki/XML#Well-formedness_and_error-handling

Since these anchors are absolute references, they may be used anywhere in the documentation source (that is, within XML <doc> structures in *.nxdl.xml* files or in `.rst` files).

The language of NXDL files is intentionally quite small, to provide only that which is necessary to describe scientific data structures (or to establish the necessary XML structures). Rather than have scientists prepare XML Schema files directly, NXDL was designed to reduce the jargon necessary to define the structure of data files. The two principle objects in NXDL files are: `group` and `field`. Documentation (`doc`) is optional for any NXDL component. Either of these objects may have additional `attributes` that contribute simple metadata.

The *Class Definitions* Chapter lists the various classes from which a NeXus file is constructed. These classes provide the glossary of items that could, in principle, be stored in a standard-conforming NeXus file (other items may be inserted into the file if the author wishes, but they won't be part of the standard). If you are going to include a particular piece of metadata, refer to the class definitions for the standard nomenclature. However, to assist those writing data analysis software, it is useful to provide more than a glossary; it is important to define the required contents of NeXus files that contain data from particular classes of neutron, X-ray, or muon instrument.

NXDL Data Types and Units

Data Types allowed in NXDL specifications

Data types for use in NXDL describe the expected type of data for a NeXus field or attribute. These terms are very broad. More specific terms are used in actual NeXus data files that describe size and array dimensions. In addition to the types in the following table, the `NAPI` type is defined when one wishes to permit a field with any of these data types. The default type `NX_CHAR` is applied in cases where a field or attribute is defined in an NXDL specification without explicit assignment of a `type`.

Unit Categories allowed in NXDL specifications

Unit categories in NXDL specifications describe the expected type of units for a NeXus field. They should describe valid units consistent with the *NeXus units* section. The values for unit categories are restricted (by an enumeration) to the following table.

NXDL File Organisation

NXDL File Name

In order for the XML machinery to find and link the code in the various files, the name of the file must be composed of the definition name (matching both the spelling and the case) and a ".nxdl.xml" extension. For example, the base class `NXarbitrary_example` should be defined by NXDL code within the `NXarbitrary_example.nxdl.xml` file. Note also that the definition name is stated twice in application definitions, once in the `definition` tag, and again as the value of an `item` contained within the `field` tag that is named "definition".

Listing 1: NXarbitrary_example.nxdl.xml

```
<definition name="NXarbitrary_example">

<!-- later -->

   <field name="definition">
```

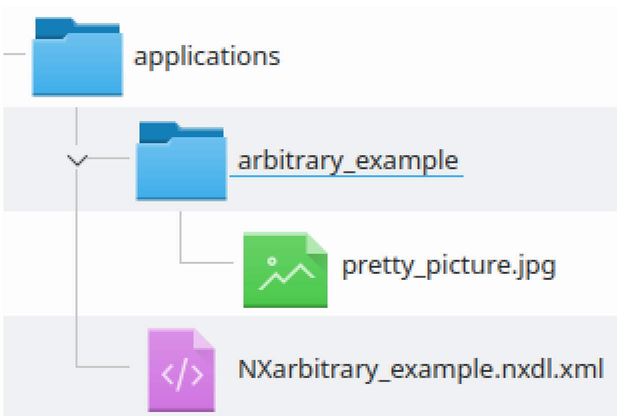
(continues on next page)

(continued from previous page)

```
      <doc>Official NeXus NXDL schema to which this file conforms.</doc>
      <enumeration>
         <item value="NXarbitrary_example"/>
      </enumeration>
   </field>
</definition>
```

Documentation Images

Including images (or other related content) in the documentation of NXDL definitions can be very effective for communicating how different parts of the definition interact. To be properly included in the compilation of the NeXus documentation, the extra files must go into a directory having the same name as the definition without the NX prefix. For example, if the `NXarbitrary_example` base class has a `pretty_picture.jpg` image included in its documentation, then the image file should be located by the path (relative to `NXarbitrary_example.xml`) `arbitrary_example/pretty_picture.jpg`.



3.3 NeXus Class Definitions

Definitions of NeXus classes. These are split into base_classes (low level objects), application definitions (groupings of objects for a particular technique) and contributed_definitions (proposed definitions from the community)

The complete vocabulary of terms used in NeXus NXDL files (names of groups, fields, attributes, and links) is available for *download*.

3.3.1 Base classes

NeXus base class definitions define the set of terms that *might* be used in an instance of that class. Consider the base classes as a set of *components* that are used to construct a data file.

Base class definitions are permissive rather than restrictive. While the terms defined aim to cover most possible use cases, and to codify the spelling and meaning of such terms, the class specifications cannot list all acceptable groups and fields. To be able to progress the NeXus standard, additional data (groups, fields, attributes) are acceptable in NeXus HDF5 data files.

Users are encouraged to find the best *defined* location in which to place their information. It is understood there is not a predefined place for all possible data.

Validation procedures should treat such additional items (not covered by a base class specification) as notes or warnings rather than errors.

3.3.2 Application Definitions

NeXus application definitions define the *minimum* set of terms that *must* be used in an instance of that class. Application definitions also may define terms that are optional in the NeXus data file.

As in base classes (see above), additional terms that are not described by the application definition may be added to data files that incorporate or adhere to application definitions.

Use NeXus links liberally in data files to reduce duplication of data. In application definitions involving raw data, write the raw data in the NXinstrument tree and then link to it from the location(s) defined in the relevant application definition. See figure *NeXus Multi Method Hierarchy* for an example.

To write a data file with an application definition, start with either a NXentry (or NXsubentry) group[1] and write the name of the application definition in the `definition` field. Then write data into this group according to the specifications of the application definition.

3.3.3 Contributed Definitions

NXDL files in the NeXus contributed definitions include propositions from the community for NeXus base classes or application definitions, as well as other NXDL files for long-term archival by NeXus. Consider the contributed definitions as either in *incubation* or a special case not for general use.

3.3.4 Downloads

See this table for the different formats available:

download file	description
`/_static/nxdl_vocabulary.html`	Human-readable HTML list of anchors, by vocabulary term, with links to the manual.
`/_static/nxdl_vocabulary.json`	vocabulary list by key in JSON format[2]
`/_static/nxdl_vocabulary.txt`	list of all anchors, sorted alphabetically
`/_static/nxdl_vocabulary.yml`	vocabulary list by key in YAML format[3]

[1] For data files involving just an application definition, use the NXentry group. Such as this structure:

```
entry:NXentry
   definition="NXsas"
```

For files that describe multi-modal data and require use of two or more application definitions (such as NXsas *and* NXcanSAS), you must place each application definition in a NXsubentry of the NXentry group. Such as this structure:

```
entry:NXentry
   raw:NXsubentry
      definition="NXsas"
   reduced:NXsubentry
      definition="NXcanSAS"
   fluo:NXsubentry
      definition="NXfluo"
```

If you anticipate your data file will eventually require an additional application definition, you should start with each application definition in a NXsubentry group.

[2] JSON: https://www.w3schools.com/whatis/whatis_json.asp

[3] YAML https://yaml.org

CHAPTER

FOUR

NAPI: NEXUS APPLICATION PROGRAMMER INTERFACE (FROZEN)

4.1 Status

This application program interface (API) was developed to support the reading and writing of NeXus files through unified function calls, regardless of the physical data format (XML, HDF4, HDF5).

In the meantime it has been decided that active development of NeXus definitions and tools will concentrate on HDF5 as the only supported physical data format. It is expected that most application developers will use standard HDF5 tools to read and write NeXus. Two examples are provided in *HDF5 in C with libhdf5*.

Therefore, the decision has been taken to freeze the NAPI. Maintenance is reduced to bug fixes.

4.2 Overview

The core routines have been written in C but wrappers are available for a number of other languages including C++, Fortran 77, Fortran 90, Java, Python and IDL. The API makes the reading and writing of NeXus files transparent; the user doesn't even need to know the underlying format when reading a file since the API calls are the same.

The NeXus Application Programming Interface for the various language backends is available on-line from https://github.com/nexusformat/code/

The NeXusIntern.pdf document (https://github.com/nexusformat/code/blob/master/doc/api/NeXusIntern.pdf) describes the internal workings of the NeXus-API. You are very welcome to read it, but it will not be of much use if all you want is to read and write files using the NAPI.

The NeXus Application Program Interface call routines in the appropriate backend (HDF4, HDF5 or XML) to read and write files with the correct structure. The API serves a number of purposes:

1. It simplifies the reading and writing of NeXus files.
2. It ensures a certain degree of compliance with the NeXus standard.
3. It hides the implementation details of the format. In particular, the API can read and write HDF4, HDF5, and XML files using the same routines.

4.3 Core API

The core API provides the basic routines for reading, writing and navigating NeXus files. Operations are performed using a handle that keeps a record of its current position in the file hierarchy. All are read or write requests are then implicitly performed on the currently *open* entity. This limits number of parameters that need to be passed to API calls, at the cost of forcing a certain mode of operation. It is very similar to navigating a directory hierarchy; NeXus groups are the directories, which can contain data sets and/or other directories.

The core API comprises the following functional groups:

- General initialization and shutdown: opening and closing the file, creating or opening an existing group or dataset, and closing them.
- Reading and writing data and attributes to previously opened datasets.
- Routines to obtain meta-data and to iterate over component datasets and attributes.
- Handling of linking and group hierarchy.
- Routines to handle memory allocation. (Not required in all language bindings.)

4.3.1 NAPI C and C++ Interface

Documentation is provided online:

C
: https://manual.nexusformat.org/doxygen/html-c/

C++
: https://manual.nexusformat.org/doxygen/html-cpp/ https://github.com/nexusformat/code/tree/master/bindings/cpp

4.3.2 NAPI Fortran 77 Interface

The bindings are listed at https://github.com/nexusformat/code/tree/master/bindings/f77 and can be built as part of the API distribution https://github.com/nexusformat/code/releases

4.3.3 NAPI Fortran 90 Interface

The Fortran 90 interface is a wrapper to the C interface with nearly identical routine definitions. As with the Fortran 77 interface, it is necessary to reverse the order of indices in multidimensional arrays, compared to an equivalent C program, so that data are stored in the same order in the NeXus file.

Any program using the F90 API needs to put the following line at the top (after the `PROGRAM` statement):

```
use NXmodule
```

Use the following table to convert from the C data types listed with each routine to the Fortran 90 data types.

C data type	F90 data type
`int, int`	`integer`
`char*`	`character(len=*)`
`NXhandle, NXhandle*`	`type(NXhandle)`
`NXstatus`	`integer`
`int[]`	`integer(:)`
`void*`	`real(:)` or `integer(:)` or `character(len=*)`
`NXlink a`, `NXlink* a`	`type(NXlink)`

The parameters in the next table, defined in `NXmodule`, may be used in defining variables.

Name	Description	Value
`NX_MAXRANK`	Maximum number of dimensions	`32`
`NX_MAXNAMELEN`	Maximum length of NeXus name	`64`
`NXi1`	Kind parameter for a 1-byte integer	`selected_int_kind(2)`
`NXi2`	Kind parameter for a 2-byte integer	`selected_int_kind(4)`
`NXi4`	Kind parameter for a 4-byte integer	`selected_int_kind(8)`
`NXr4`	Kind parameter for a 4-byte real	`kind(1.0)`
`NXr8`	Kind parameter for an 8-byte real	`kind(1.0D0)`

The bindings are listed at https://github.com/nexusformat/code/tree/master/bindings/f90 and can be built as part of the API distribution https://github.com/nexusformat/code/releases

4.3.4 NAPI Java Interface

This section includes installation notes, instructions for running NeXus for Java programs and a brief introduction to the API.

The Java API for NeXus (`jnexus`) was implemented through the Java Native Interface (JNI) to call on to the native C library. This has a number of disadvantages over using pure Java, however the most popular file backend HDF5 is only available using a JNI wrapper anyway.

Acknowledgement

This implementation uses classes and native methods from NCSA's Java HDF Interface project. Basically all conversions from native types to Java types is done through code from the NCSA HDF group. Without this code the implementation of this API would have taken much longer. See NCSA's copyright for more information.

Installation

Requirements

Caution: Documentation is old and may need revision.

For running an application with `jnexus` an recent Java runtime environment (JRE) will do.

In order to compile the Java API for NeXus a Java Development Kit is required on top of the build requirements for the C API.

Installation under Windows

1. Copy the HDF DLL's and the file `jnexus.dll` to a directory in your path. For instance `C:\\Windows\\system32`.
2. Copy the `jnexus.jar` to the place where you usually keep library jar files.

Note that the location or the naming of these files in the binary Nexus distributions have changed over the years. In the Nexus 4.3.0 Windows 64-bit distribution (see Assets in https://github.com/nexusformat/code/releases/tag/4.3.0), By default, the DLL is at: `C:\Program Files\NeXus Data Format\bin\libjnexus-0.dll`. Please rename this file to `jnexus.dll` before making it available in your path. This is important, otherwise, JVM runtime will not be able to locate this file.

For the same distribution, the location of `jnexus.jar` is at: `C:\Program Files\NeXus Data Format\share\java`.

Installation under Unix

The `jnexus.so` shared library as well as all required file backend `.so` libraries are required as well as the `jnexus.jar` file holding the required Java classes. Copy them wherever you like and see below for instructions how to run programs using `jnexus`.

Running Programs with the NeXus API for Java

In order to successfully run a program with `jnexus`, the Java runtime systems needs to locate two items:

1. The shared library implementing the native methods.
2. The `nexus.jar` file in order to find the Java classes.

Locating the shared libraries

The methods for locating a shared library differ between systems. Under Windows32 systems the best method is to copy the `jnexus.dll` and the HDF4, HDF5 and/or XML-library DLL files into a directory in your path.

On a UNIX system, the problem can be solved in three different ways:

1. Make your system administrator copy the `jnexus.so` file into the systems default shared library directory (usually `/usr/lib` or `/usr/local/lib`).
2. Put the `jnexus.so` file wherever you see fit and set the `LD_LIBRARY_PATH` environment variable to point to the directory of your choice.
3. Specify the full pathname of the jnexus shared library on the java command line with the `-Dorg.nexusformat.JNEXUSLIB=full-path-2-shared-library` option.

Locating `jnexus.jar`

This is easier, just add the the full pathname to `jnexus.jar` to the classpath when starting java. Here are examples for a UNIX shell and the Windows shell.

UNIX example shell script to start `jnexus.jar`

```
#!/sbin/sh
java -classpath /usr/lib/classes.zip:../jnexus.jar:. \
    -Dorg.nexusformat.JNEXUSLIB=../libjnexus.so TestJapi
```

Windows 32 example batch file to start `jnexus.jar`

```
set JL=-Dorg.nexusformat.JNEXUSLIB=..\jnexus\bin\win32\jnexus.dll
java -classpath C:\jdk1.5\lib\classes.zip;..\jnexus.jar;. %JL% TestJapi
```

Programming with the NeXus API for Java

The NeXus C-API is good enough but for Java a few adaptions of the API have been made in order to match the API better to the idioms used by Java programmers. In order to understand the Java-API, it is useful to study the NeXus C-API because many methods work in the same way as their C equivalents. A full API documentation is available in Java documentation format. For full reference look especially at:

- The interface `NeXusFileInterface` first. It gives an uncluttered view of the API.
- The implementation `NexusFile` which gives more details about constructors and constants. However this documentation is interspersed with information about native methods which should not be called by an application programmer as they are not part of the standard and might change in future.

See the following code example for opening a file, opening a vGroup and closing the file again in order to get a feeling for the API:

fragment for opening and closing

```
    try{
        NexusFile nf = new NexusFile(filename, NexusFile.NXACC_READ);
        nf.opengroup("entry1","NXentry");
        nf.finalize();
    }catch(NexusException ne) {
        // Something was wrong!
    }
```

Some notes on this little example:

- Each NeXus file is represented by a `NexusFile` object which is created through the constructor.
- The `NexusFile` object takes care of all file handles for you. So there is no need to pass in a handle anymore to each method as in the C language API.
- All error handling is done through the Java exception handling mechanism. This saves all the code checking return values in the C language API. Most API functions return void.

- Closing files is tricky. The Java garbage collector is supposed to call the finalize method for each object it decides to delete. In order to enable this mechanism, the `NXclose()` function was replaced by the `finalize()` method. In practice it seems not to be guaranteed that the garbage collector calls the `finalize()` method. It is safer to call `finalize()` yourself in order to properly close a file. Multiple calls to the `finalize()` method for the same object are safe and do no harm.

Data Writing and Reading

Again a code sample which shows how this looks like:

fragment for writing and reading

```
int idata[][] = new idata[10][20];
int iDim[] = new int[2];

// put some data into idata.......

// write idata
iDim[0] = 10;
iDim[1] = 20;
nf.makedata("idata",NexusFile.NX_INT32,2,iDim);
nf.opendata("idata");
nf.putdata(idata);

// read idata
nf.getdata(idata);
```

The dataset is created as usual with `makedata()` and opened with `putdata()`. The trick is in `putdata()`. Java is meant to be type safe. One would think then that a `putdata()` method would be required for each Java data type. In order to avoid this, the data to `write()` is passed into `putdata()` as type `Object`. Then the API proceeds to analyze this object through the Java introspection API and convert the data to a byte stream for writing through the native method call. This is an elegant solution with one drawback: An array is needed at all times. Even if only a single data value is written (or read) an array of length one and an appropriate type is the required argument.

Another issue are strings. Strings are first class objects in Java. HDF (and NeXus) sees them as dumb arrays of bytes. Thus strings have to be converted to and from bytes when reading string data. See a writing example:

String writing

```
String ame = "Alle meine Entchen";
nf.makedata("string_data",NexusFile.NX_CHAR,
        1,ame.length()+2);
nf.opendata("string_data");
nf.putdata(ame.getBytes());
```

And reading:

String reading

```
byte bData[] = new byte[132];
nf.opendata("string_data");
nf.getdata(bData);
String string_data = new String(bData);
```

The aforementioned holds for all strings written as SDS content or as an attribute. SDS or vGroup names do not need this treatment.

Inquiry Routines

Let us compare the C-API and Java-API signatures of the `getinfo()` routine (C) or method (Java):

C API signature of `getinfo()`

```
/* C -API */
NXstatus NXgetinfo(NXhandle handle, int *rank, int iDim[],
                   int *datatype);
```

Java API signature of `getinfo()`

```
// Java
void getinfo(int iDim[], int args[]);
```

The problem is that Java passes arguments only by value, which means they cannot be modified by the method. Only array arguments can be modified. Thus `args` in the `getinfo()` method holds the rank and datatype information passed in separate items in the C-API version. For resolving which one is which, consult a debugger or the API-reference.

The attribute and vGroup search routines have been simplified using Hashtables. The `Hashtable` returned by `groupdir()` holds the name of the item as a key and the classname or the string SDS as the stored object for the key. Thus the code for a vGroup search looks like this:

vGroup search

```
nf.opengroup(group,nxclass);
h = nf.groupdir();
e = h.keys();
System.out.println("Found in vGroup entry:");
while(e.hasMoreElements())
{
    vname = (String)e.nextElement();
    vclass = (String)h.get(vname);
    System.out.println("     Item: " + vname + " class: " + vclass);
}
```

For an attribute search both at global or SDS level the returned Hashtable will hold the name as the key and a little class holding the type and size information as value. Thus an attribute search looks like this in the Java-API:

attribute search

```
Hashtable h = nf.attrdir();
Enumeration e = h.keys();
while(e.hasMoreElements())
{
    attname = (String)e.nextElement();
    atten = (AttributeEntry)h.get(attname);
    System.out.println("Found global attribute: " + attname +
        " type: "+ atten.type + " ,length: " + atten.length);
}
```

For more information about the usage of the API routines see the reference or the NeXus C-API reference pages. Another good source of information is the source code of the test program which exercises each API routine.

Known Problems

These are a couple of known problems which you might run into:

Memory
: As the Java API for NeXus has to convert between native and Java number types a copy of the data must be made in the process. This means that if you want to read or write 200MB of data your memory requirement will be 400MB! This can be reduced by using multiple `getslab()/putslab()` to perform data transfers in smaller chunks.

`Java.lang.OutOfMemoryException`
: By default the Java runtime has a low default value for the maximum amount of memory it will use. This ceiling can be increased through the `-mxXXm` option to the Java runtime. An example: `java -mx512m ...` starts the Java runtime with a memory ceiling of 512MB.

Maximum 8192 files open
: The NeXus API for Java has a fixed buffer for file handles which allows only 8192 NeXus files to be open at the same time. If you ever hit this limit, increase the `MAXHANDLE` define in `native/handle.h` and recompile everything.

On-line Documentation

The following documentation is browsable online:

1. The API source code
2. A verbose tutorial for the NeXus for Java API.
3. The API Reference.
4. Finally, the source code for the test driver for the API which also serves as a documented usage example.

4.3.5 NAPI IDL Interface

IDL is an interactive data evaluation environment developed by Research Systems - it is an interpreted language for data manipulation and visualization. The NeXus IDL bindings allow access to the NeXus API from within IDL - they are installed when NeXus is compiled from source after being configured with the following options:

```
configure \
        --with-idlroot=/path/to/idl/installation \
        --with-idldlm=/path/to/install/dlm/files/to
```

For further details see the README (https://htmlpreview.github.com/?https://github.com/nexusformat/code/blob/master/bindings/idl/README.html) for the NeXus IDL binding. The source code is stored at https://github.com/nexusformat/code/tree/master/bindings/idl

4.4 Utility API

The NeXus F90 Utility API provides a number of routines that combine the operations of various core API routines in order to simplify the reading and writing of NeXus files. At present, they are only available as a Fortran 90 module but a C version is in preparation.

The utility API comprises the following functional groups:

- Routines to read or write data.
- Routines to find whether or not groups, data, or attributes exist, and to find data with specific signal or axis attributes, i.e. to identify valid data or axes.
- Routines to open other groups to which `NXdata` items are linked, and to return again.

line required for use with F90 API

Any program using the F90 Utility API needs to put the following line near the top of the program:

```
use NXUmodule
```

Note: Do not put `USE` statements for both `NXmodule` and `NXUmodule`. The former is included in the latter

4.4.1 List of F90 Utility Routines

name	description
Reading and Writing	
`NXUwriteglobals`	Writes all the valid global attributes of a file.
`NXUwritegroup`	Opens a group (creating it if necessary).
`NXUwritedata`	Opens a data item (creating it if necessary) and writes data and its units.
`NXUreaddata`	Opens and reads a data item and its units.
`NXUwritehistogram`	Opens one dimensional data item (creating it if necessary) and writes histogram centers and their units.
`NXUreadhistogram`	Opens and reads a one dimensional data item and converts it to histogram bin boundaries.
`NXUsetcompress`	Defines the compression algorithm and minimum dataset size for subsequent write operations.
Finding Groups, Data, and Attributes	
`NXUfindclass`	Returns the name of a group of the specified class if it is contained within the currently open group.
`NXUfinddata`	Checks whether a data item of the specified name is contained within the currently open group.
`NXUfindattr`	Checks whether the currently open data item has the specified attribute.
`NXUfindsignal`	Searches the currently open group for a data item with the specified `SIGNAL` attribute.
`NXUfindaxis`	Searches the currently open group for a data item with the specified `AXIS` attribute.
Finding Linked Groups	
`NXUfindlink`	Finds another link to the specified NeXus data item and opens the group it is in.
`NXUresumelink`	Reopens the original group from which `NXUfindlink` was used.

Currently, the F90 utility API will only write character strings, 4-byte integers and reals, and 8-byte reals. It can read other integer sizes into four-byte integers, but does not differentiate between signed and unsigned integers.

4.5 Building Programs

The install kit provides a utility call `nxbuild` that can be used to build simple programs:

```
nxbuild -o test test.c
```

This script links in the various libraries for you and reading its contents would provide the necessary information for creating a separate Makefile. You can also use `nxbuild` with the example files in the NeXus distribution kit which are installed into `/usr/local/nexus/examples`

Note that the executable name is important in this case as the test program uses it internally to determine the `NXACC_CREATE*` argument to pass to `NXopen`.

building and running a simple NeXus program

```
#  builds HDF5 specific test
nxbuild -o napi_test-hdf5 napi_test.c

# runs the test
./napi_test-hdf5
```

NeXus is also set up for pkg-config so the build can be done as:

```
gcc `pkg-config --cflags` `pkg-config --libs` -o test test.c
```

4.6 Reporting Bugs in the NeXus API

If you encounter any bugs in the installation or running of the NeXus API, please report them online using our Issue Reporting system. (https://www.nexusformat.org/IssueReporting.html)

CHAPTER

FIVE

NEXUS COMMUNITY

NeXus began as a group of scientists with the goal of defining a common data storage format to exchange experimental results and to exchange ideas about how to analyze them.

The NeXus Scientific Community provides the scientific data, advice, and continued involvement with the NeXus standard. NeXus provides a forum for the scientific community to exchange ideas in data storage.

The NeXus International Advisory Committee (NIAC) supervises the development and maintenance of the NeXus common data format for neutron, X-ray, and muon science through the NeXus class definitions and oversees the maintenance of the NeXus Application Programmer Interface (NAPI) as well as the technical infrastructure.

There are several mechanisms in place in order to coordinate the development of NeXus with the larger community.

5.1 NeXus Webpage

First of all, there is the NeXus webpage, https://www.nexusformat.org/, which provides all kinds of information, including membership, minutes, and discussions from the meetings of the NIAC, Code Camps, and Tele Conferences, as well as some proposed designs for consideration by NeXus.

The webpage is kept with a number of other repositories in the nexusformat.org Github organisation https://github.com/nexusformat/. As for all of these repositories, pull requests to correct or improve the content or code are always welcome!

5.2 Contributed Definitions

The community is encouraged to provide new definitions (base.class.definitions or application.definitions) for consideration in the NeXus standard. These community contributions will be entered in the contributed.definitions and will be curated according to procedures set forth by the *NIAC: The NeXus International Advisory Committee*.

5.3 Other Ways NeXus Coordinates with the Scientific Community

5.3.1 NIAC: The NeXus International Advisory Committee

The purpose of the NeXus International Advisory Committee (NIAC)[1] is to supervise the development and maintenance of the NeXus common data format for neutron, X-ray, and muon science. This purpose includes, but is not limited to, the following activities.

[1] For more details about the NIAC constitution, procedures, and meetings, refer to the NIAC web page: https://www.nexusformat.org/NIAC.html The members of the NIAC may be reached by email: nexus-committee@nexusformat.org

1. To establish policies concerning the definition, use, and promotion of the NeXus format.
2. To ensure that the specification of the NeXus format is sufficiently complete and clear for its use in the exchange and archival of neutron, X-ray, and muon data.
3. To receive and examine all proposed amendments and extensions to the NeXus format. In particular, to ratify proposed instrument and group class definitions, to ensure that the data structures conform to the basic NeXus specification, and to ensure that the definitions of data items (fields) are clear and unambiguous and conform to accepted scientific usage.
4. To ensure that documentation of the NeXus format is sufficient, current, and available to potential users both on the internet and in other forms.
5. To coordinate the maintenance of the NeXus Application Programming Interface and to promote other software development that will benefit users of the NeXus format.
6. To coordinate with other organizations that maintain and develop related data formats to reach compatibility.

The committee will meet at least once every other calendar year according to the following plan:

- In years coinciding with the NOBUGS series of conferences (once every two years), members of the entire NIAC will meet as a satellite meeting to NOBUGS, along with interested members of the community.
- In intervening years, the executive officers of the NIAC will attend, along with interested members of the NIAC. This is intended to be a working meeting with a small group.

5.3.2 NeXus Mailing List

We invite anyone who is associated with neutron and/or X-ray synchrotron science and who wishes to be involved in the development and testing of the NeXus format to subscribe to this list. It is a public list for the free discussion of all aspects of the design and operation of the NeXus format.

- List Address: nexus@nexusformat.org
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus
- Archive: http://lists.nexusformat.org/pipermail/nexus

Note: Subscription to nexus@nexusformat.org does not subscribe you automatically to any other NeXus mailing list.

5.3.3 NeXus International Advisory Committee (NIAC) Mailing List

This list contains discussions of the *NIAC: The NeXus International Advisory Committee*, which oversees the development of the NeXus data format. Its members represent many of the major neutron and synchrotron scattering sources in the world. Membership and posting to this list are limited to the committee members, but the archives are public. The NIAC mailing list is for communications specific to NIAC and not for public contribution. General discussions should be held in the public mailing list.

- List Address: nexus-committee@nexusformat.org
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-committee
- Archive: http://lists.nexusformat.org/pipermail/nexus-committee

Note: Subscription to nexus-committee@nexusformat.org does not subscribe you automatically to any other NeXus mailing list.

5.3.4 NeXus Video Conference Announcements

There are video conferences on NeXus roughly twice a month. Agenda and joining details are posted on the webpage: https://www.nexusformat.org/Teleconferences.html In addition calendar invites are sent to this list. NeXus-Tech used to be used for discussions in the past. Now the list is moderated to only allow communication related to holding meetings. All other traffic should go to the main list nexus@nexusformat.org

- List Address: nexus-tech@nexusformat.org
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-tech

5.3.5 NeXus Developers Mailing List (retired)

This mailing list was for discussions concerning the technical development of NeXus (the Definitions, NXDL, and the NeXus Application Program Interface). There was, however, much overlap with the general NeXus mailing list and so this separate list was closed in October 2012, but the archive of previous posting is still available.

- Archive: http://lists.nexusformat.org/pipermail/nexus-developers

5.3.6 NeXus Repositories

NeXus NXDL class definitions (both base classes and application definitions) and the NeXus code library source are held in a pair of git repositories on GitHub.

The repositories are world readable. You can browse them directly:

NeXus code library and applications
: https://github.com/nexusformat/code

NeXus NXDL class definitions
: https://github.com/nexusformat/definitions

NeXus GitHub *organization*
: https://github.com/nexusformat

If you would like to contribute (thank you!), the normal GitHub procedure of forking the repository and generating pull requests should be used.

Please report any problems via the *Issue Reporting* system.

5.3.7 NeXus Issue Reporting

NeXus is using GitHUb (https://github.com) as source code repository and for problem reporting. The issue reports (see *View current issues* below) are used to guide the NeXus developers in resolving problems as well as implementing new features.

NeXus Code (NAPI, Library, and Applications)

Report a new issue
https://github.com/nexusformat/code/issues/new

View current issues
https://github.com/nexusformat/code/issues

Timeline (recent ticket and code changes)
https://github.com/nexusformat/code/pulse

NeXus Definitions (NXDL base classes and application definitions)

Report a new issue
https://github.com/nexusformat/definitions/issues/new

View current issues
https://github.com/nexusformat/definitions/issues

Timeline (recent ticket and definition changes)
https://github.com/nexusformat/definitions/pulse

CHAPTER

SIX

INSTALLATION

This section describes how to install the NeXus API and details the requirements. The NeXus API is distributed under the terms of the GNU Lesser General Public License version 3.

The source distribution of NAPI can be downloaded from the release page of the associated GitHub project. Instructions how to build the code can be found in the *INSTALL.rst* file shipped with the source distribution. In case you need help, feel free to contact the NeXus mailing list: http://lists.nexusformat.org/mailman/listinfo/nexus

6.1 Precompiled Binary Installation

6.1.1 Linux RPM Distribution Kits

An installation kit (source or binary) can be downloaded from: https://github.com/nexusformat/code/releases/tag/4.3.0

A NeXus binary RPM (nexus-*.i386.rpm) contains ready compiled NeXus libraries whereas a source RPM (nexus-*.src.rpm) needs to be compiled into a binary RPM before it can be installed. In general, a binary RPM is installed using the command

```
rpm -Uvh file.i386.rpm
```

or, to change installation location from the default (e.g. /usr/local) area, using

```
rpm -Uvh --prefix /alternative/directory file.i386.rpm
```

If the binary RPMS are not the correct architecture for you (e.g. you need x86_64 rather than i386) or the binary RPM requires libraries (e.g. HDF4) that you do not have, you can instead rebuild a source RPM (.src.rpm) to generate the correct binary RPM for you machine. Download the source RPM file and then run

```
rpmbuild --rebuild file.src.rpm
```

This should generate a binary RPM file which you can install as above. Be careful if you think about specifying an alternative buildroot for rpmbuild by using `--buildroot` option as the "buildroot" directory tree will get remove (so `--buildroot /` is a really bad idea). Only change buildroot it if the default area turns out not to be big enough to compile the package.

If you are using Fedora, then you can install all the dependencies by typing

```
yum install hdf hdf-devel hdf5 hdf5-devel mxml mxml-devel
```

6.1.2 Microsoft Windows Installation Kit

A Windows MSI based installation kit is available and can be downloaded from: https://github.com/nexusformat/code/releases/tag/4.3.0

6.1.3 Mac OS X Installation Kit

An installation disk image (.dmg) can be downloaded from: https://github.com/nexusformat/code/releases/tag/4.3.0

6.2 Source Installation

6.2.1 NeXus Source Code Distribution

The source code distribution can be obtained from GitHub. One can either checkout the git repositories to get access to the most recent development code. To clone the definitions repository use

```
$ git clone https://github.com/nexusformat/definitions.git definitions
```

or for the NAPI

```
$ git clone https://github.com/nexusformat/code.git code
```

For release tarballs go to the release page for the NAPI or the definitions. For the definitions it is currently recommended to work directly with the Git repository as the actual release is rather outdated.

Instructions how to build the NAPI code can be found either on the GitHub project website or in the *README.rst* file shipped with the source distribution.

6.3 Releases

The NeXus definitions are expected to evolve. The evolution is marked as a series of *releases* which are snapshots of the repository (and current state of the NeXus standard). Each new *release* of the definitions will be posted to the definitions GitHub repository and announced to the community via the NeXus mailing list: *nexus@nexusformat.org*

6.3.1 NeXus definitions

Releases of the NeXus definitions are listed on the GitHub web site: https://github.com/nexusformat/definitions/releases

Release Notes

Detailed notes about each release (start with v3.3) are posted to the definitions GitHub wiki: https://github.com/nexusformat/definitions/wiki/Release-Notes

Release Process

The process to make a new release of the NeXus definitions repository is documented in the repository's GitHub wiki: https://github.com/nexusformat/definitions/wiki/Release-Procedure.

The release process starts by creating a GitHub [Milestone](https://help.github.com/articles/tracking-the-progress-of-your-work-with-milestones/) for the new release. Milestones for the NeXus definitions repository are available on the GitHub site: https://github.com/nexusformat/definitions/milestones

Versioning (Tags)

Versioning of each of the NXDL files, as well as the complete set of NXDL files is now described in the wiki[1] of the NeXus definitions repository[2]. Please see that wiki for complete information.

In case you need help, feel free to contact the *NeXus Mailing List*:

Archives
    http://lists.nexusformat.org/mailman/listinfo/nexus

email
    nexus@nexusformat.org

[1] Release Procedure: https://github.com/nexusformat/definitions/wiki/Release-Procedure
[2] Definitions repository: https://github.com/nexusformat/definitions

CHAPTER

SEVEN

NEXUS UTILITIES

There are many utilities available to read, browse, write, and use NeXus data files. Some are provided by the NeXus technical group while others are provided by the community. Still, other tools listed here can read or write one of the low-level file formats used by NeXus (HDF5, HDF4, or XML).

Furthermore, there are specific examples of code that can read, write, (or both) NeXus data files, given in the section *Language APIs for NeXus and HDF5*.

The NIAC welcomes your continued contributions to this documentation.

Please note that NeXus maintains a repository of example data files[1] which you may browse and download. There is a cursory analysis[2] of every file in this repository as to whether it can be read as HDF5 or NeXus HDF5. The analysis code[3], which serves as yet another example reader, is made using python and h5py.

7.1 Utilities supplied with NeXus

Most of these utility programs are run from the command line. It will be noted if a program provides a graphical user interface (GUI). Short descriptions are provided here with links to further information, as available.

nxbrowse
NeXus Browser

nxconvert
Utility to convert a NeXus file into HDF4/HDF5/XML/…

nxdir
`nxdir` is a utility for querying a NeXus file about its contents. Full documentation can be found by running this command:

```
nxdir -h
```

nxingest
`nxingest` extracts the metadata from a NeXus file to create an XML file according to a mapping file. The mapping file defines the structure (names and hierarchy) and content (from either the NeXus file, the mapping file or the current time) of the output file. See the man page for a description of the mapping file. This tool uses the NAPI. Thus, any of the supported formats (HDF4, HDF5 and XML) can be read.

nxsummary
Use `nxsummary` to generate summary of a NeXus file. This program relies heavily on a configuration file. Each `item` tag in the file describes a node to print from the NeXus file. The `path` attribute describes where in the NeXus file to get information from. The `label` attribute will be printed when showing the value of the specified

[1] https://github.com/nexusformat/exampledata
[2] https://github.com/nexusformat/exampledata/blob/master/critique.md
[3] https://github.com/nexusformat/exampledata/blob/master/critique.py

field. The optional `operation` attribute provides for certain operations to be performed on the data before printing out the result. See the source code documentation for more details.

nxtranslate

`nxtranslate` is an anything to NeXus converter. This is accomplished by using translation files and a plugin style of architecture where `nxtranslate` can read from new formats as plugins become available. The documentation for `nxtranslate` describes its usage by three types of individuals:

- the person using existing translation files to create NeXus files
- the person creating translation files
- the person writing new *retrievers*

All of these concepts are discussed in detail in the documentation provided with the source code.

NXplot

An extendable utility for plotting any NeXus file. `NXplot` is an Eclipse-based GUI project in Java to plot data in NeXus files. (The project was started at the first NeXus Code Camp in 2009.)

7.2 Validation

The list of applications below are for *validating* NeXus files. The list is not intended to be a complete list of all available packages.

cnxvalidate

NeXus validation tool written in C (not via NAPI).

Its dependencies are libxml2 and the HDF5 libraries, version 1.8.9 or better. Its purpose is to validate HDF5 files against NeXus application definitions.

See the program documentation for more details: https://github.com/nexusformat/cnxvalidate.git

punx

Python Utilities for NeXus HDF5 files

punx can validate both NXDL files and NeXus HDF5 data files, as well as print the structure of any HDF5 file, even non-NeXus files.

NOTE: project is under initial construction, not yet released for public use, but is useful in its present form (version 0.2.5).

punx can show the tree structure of any HDF5 file. The output is more concise than that from *h5dump*.

See the program documentation for more details: https://punx.readthedocs.io

7.3 Other Utilities

NeXus Constructor (https://github.com/ess-dmsc/nexus-constructor)

The NeXus Constructor facilitates constructing NeXus files in which to record data from experiments at neutron science facilities. This includes all supporting metadata typically required to perform analysis of such experiments, including instrument geometry information.

nxdl_to_hdf5.py (https://github.com/nexusformat/exampledata/tree/master/nxdl)

`nxdl_to_hdf5.py` is a Python script that reads the NeXus definition files (files ending with `.nxdl.xml`) and creates example Python scripts as well as HDF5 files for each definition. There are generated example scripts of each application definition for both *h5py* and *nexusformat*. Currently, only application definitions and some contributed_definitions are supported as the code depends on the existence of an NXentry in the definition.

7.4 Data Analysis

The list of applications below are some of the utilities that have been developed (or modified) to read/write NeXus files as a data format. It is not intended to be a complete list of all available packages.

DAVE (http://www.ncnr.nist.gov/dave/)
: DAVE is an integrated environment for the reduction, visualization and analysis of inelastic neutron scattering data. It is built using IDL (Interactive Data Language) from ITT Visual Information Solutions.

DAWN (http://www.dawnsci.org)
: The Data Analysis WorkbeNch (DAWN) project is an eclipse based workbench for doing scientific data anaylsis. It offers generic visualisation, and domain specific processing.

GDA (http://www.opengda.org)
: The GDA project is an open-source framework for creating customised data acquisition software for science facilities such as neutron and X-ray sources. It has elements of the DAWN analysis workbench built in.

Gumtree (https://archive.ansto.gov.au/ResearchHub/OurInfrastructure/ACNS/Facilities/Computing/GumTree/index.htm)
: Gumtree is an open source project, providing a graphical user interface for instrument status and control, data acquisition and data reduction.

IDL (https://www.harrisgeospatial.com/docs/using_idl_home.html)
: IDL is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.

IgorPro (http://www.wavemetrics.com/)
: IGOR Pro is an extraordinarily powerful and extensible scientific graphing, data analysis, image processing and programming software tool for scientists and engineers.

ISAW (ftp://ftp.sns.gov/ISAW/)
: The Integrated Spectral Analysis Workbench software project (ISAW) is a Platform-Independent system Data Reduction/Visualization. ISAW can be used to read, manipulate, view, and save neutron scattering data. It reads data from IPNS run files or NeXus files and can merge and sort data from separate measurements.

LAMP (http://www.ill.eu/data_treat/lamp/>)
: LAMP (Large Array Manipulation Program) is designed for the treatment of data obtained from neutron scattering experiments at the Institut Laue-Langevin. However, LAMP is now a more general purpose application which can be seen as a GUI-laboratory for data analysis based on the IDL language.

Mantid (http://www.mantidproject.org/)
: The Mantid project provides a platform that supports high-performance computing on neutron and muon data. It is being developed as a collaboration between Rutherford Appleton Laboratory and Oak Ridge National Laboratory.

MATLAB (http://www.mathworks.com/)
: MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.

NeXpy (http://nexpy.github.io/nexpy/)
: The goal of NeXpy is to provide a simple graphical environment, coupled with Python scripting capabilities, for the analysis of X-Ray and neutron scattering data. (It was decided at the NIAC 2010 meeting that a large portion of this code would be adopted in the future by NeXus and be part of the distribution)

silx (http://www.silx.org/doc/silx/latest/)
: The silx project aims to provide a collection of Python packages to support the development of data assessment, reduction and analysis at synchrotron radiation facilities. In particular it provides tools to read, write and visualize NeXus HDF5 files.

OpenGENIE (http://www.opengenie.org/)
A general purpose data analysis and visualisation package primarily developed at the ISIS Facility, Rutherford Appleton Laboratory.

PyMCA (http://pymca.sourceforge.net/)
PyMca is a ready-to-use, and in many aspects state-of-the-art, set of applications implementing most of the needs of X-ray fluorescence data analysis. It also provides a Python toolkit for visualization and analysis of energy-dispersive X-ray fluorescence data. Reads, browses, and plots data from NeXus HDF5 files.

spec2nexus (https://spec2nexus.readthedocs.io)
(Python code) Converts SPEC data files and scans into NeXus HDF5 files. (Note the *h5toText* tool mentioned here previously is no longer available from the *spec2nexus* project. The code has been moved into the *punx* project: https://punx.readthedocs.io/.)

spec2nexus provides libraries:

- *spec2nexus.spec*: python binding to read SPEC[4] data files
- *spec2nexus.eznx*: (Easy NeXus) supports writing NeXus HDF5 files using h5py

7.5 HDF Tools

Here are some of the generic tools that are available to work with HDF files. In addition to the software listed here there are also APIs for many programming languages that will allow low level programmatic access to the data structures.

HDF Group command line tools (http://www.hdfgroup.org/products/hdf5_tools/#h5dist/)
There are various command line tools that are available from the HDF Group, these are usually shipped with the HDF5 kits but are also available for download separately.

HDFexplorer (http://www.space-research.org/)
A data visualization program that reads Hierarchical Data Format files (HDF, HDF-EOS and HDF5) and also netCDF data files.

HDFview (http://www.hdfgroup.org)
A Java based GUI for browsing (and some basic plotting) of HDF files.

7.6 Language APIs for NeXus and HDF5

Collected here are some of the tools identified[5] as a result of a simple question asked at the 2018 Nobugs conference: *Are there examples of code that reads NeXus data?* Some of these are very specific to an instrument or application definition while others are more generic. The lists below are organized by programming language, yet some collections span several languages so they are listed in the section *Language API: mixed*.

Note these example listed in addition to the many examples described here in the manual, in section :*Examples*.

[4] SPEC: http://www.certif.com
[5] https://github.com/nexusformat/definitions/issues/630

7.6.1 Language API: *F77*

- **POLDI**: `poldi.zip`[6] contains: - A F77 reading routine using NAPI for POLDI at SINQ PSI - an example of a file which it reads

7.6.2 Language API: *IDL*

- **aXis2000**[7], with the NeXus-specific IDL code in the `read_nexus.pro`[8], reads NXstxm

7.6.3 Language API: *IgorPro*

- **HDF5gateway**[9] makes it easy to read a HDF5 file (including NeXus) into an IgorPro[10] folder, including group and dataset attributes, such as a NeXus data file, modify it, and then write the folder structure back out.

7.6.4 Language API: *Java*

- **Dawn**[11] has java code to read[12] and write[13] HDF5 NeXus files (generic NeXus, not tied to specific application definitions).
- `NXreader.zip`[14] is java code which reads NeXus files into **ImageJ.** It uses the Java-hdf interface to HDF5. It tries to do a good job locating the image dataset by NeXus conventions. But it uses the old style conventions.

7.6.5 Language API: *Python*

- **Dials**[15] has python (and some C++) code for reading NXmx[16]
  - *cctbx.xfel* code for writing[17] NXmx master files for JF16M at SwissFEL
- **h5py**[Page 167, 18]
  HDF5 for Python (h5py) is a general-purpose Python interface to HDF5.
- **Mantis**[19], with NeXus-specific python code[20], reads NXstxm
- **nexusformat**[21] **NeXus package for Python**
  Provides an API to open, create, plot, and manipulate NeXus data.

[6] https://github.com/nexusformat/definitions/files/4107360/poldi.zip
[7] http://unicorn.chemistry.mcmaster.ca/aXis2000.html
[8] `read_nexus.pro`: http://unicorn.chemistry.mcmaster.ca/axis/aXis2000.zip
[9] https://github.com/prjemian/hdf5gateway
[10] IgorPro: https://wavemetrics.com
[11] https://dawnsci.org/
[12] read: https://github.com/DawnScience/scisoft-core/blob/master/uk.ac.diamond.scisoft.analysis/src/uk/ac/diamond/scisoft/analysis/io/NexusHDF5Loader.java
[13] write: https://github.com/DawnScience/dawnsci/blob/master/org.eclipse.dawnsci.hdf5/src/org/eclipse/dawnsci/hdf5/nexus/NexusFileHDF5.java
[14] https://github.com/nexusformat/definitions/files/4107439/NXreader.zip
[15] https://dials.github.io/
[16] read: https://github.com/cctbx/dxtbx/blob/master/format/nexus.py
[17] write: https://github.com/cctbx/cctbx_project/blob/master/xfel/swissfel/jf16m_cxigeom2nexus.py
[18] http://docs.h5py.org
[19] Mantis: http://spectromicroscopy.com/
[20] python code: https://bitbucket.org/mlerotic/spectromicroscopy/src/default/
[21] https://github.com/nexpy/nexusformat

- **SasView**[22] has python code to read[23] and write[24] NXcanSAS

7.6.6 Language API: *mixed*

- **FOCUS**: `focus.zip`[25] contains:
  - An example FOCUS file
  - focusreport: A h5py program which skips through a list of files and prints statistics
  - focusreport.tcl, same as above but in Tcl using the Swig generated binding to NAPI
  - i80.f contains a F77 routine for reading FOCUS files into Ida. The routine is RRT_in_Foc.
- **ZEBRA**: `zebra.zip`[26] contains:
  - an example file
  - zebra-to-ascii, a h5py script which dumps a zebra file to ASCII
  - `TRICSReader.*` for reading ZEBRA files in C++ using C-NAPI calls

[22] https://www.sasview.org/
[23] read: https://github.com/SasView/sasview/blob/master/src/sas/sascalc/dataloader/readers/cansas_reader_HDF5.py
[24] write: https://github.com/SasView/sasview/blob/master/src/sas/sascalc/file_converter/nxcansas_writer.py
[25] https://github.com/nexusformat/definitions/files/4107386/focus.zip
[26] https://github.com/nexusformat/definitions/files/4107416/zebra.zip

CHAPTER

EIGHT

BRIEF HISTORY OF NEXUS

Two things to note about the development and history of NeXus:

- All efforts on NeXus have been voluntary except for one year when we had one full-time worker.
- The NIAC has already discussed many matters related to the format.

2018-05:

- *release v2018.5 <https://github.com/nexusformat/definitions/wiki/releasenotes__v2018.5>* of NeXus Definitions
- **#597**
  changed versioning scheme and procedures

2017-07:
release 3.3 <https://github.com/nexusformat/definitions/wiki/releasenotes__v3.3> of NeXus Definitions

2016-10:
release 3.2 <https://github.com/nexusformat/definitions/releases/tag/v3.2> of NeXus Definitions

2014-12:
The NIAC approves a new method to identify the default data to be plotted, applying attributes at the group level to the root of the HDF5 tree, and the NXentry and NXdata groups. See the description in *Associating plottable data using attributes applied to the NXdata group* and the proposal: https://www.nexusformat.org/2014_How_to_find_default_data.html

2012-05:
first release (3.1.0) of NXDL (NeXus Definition Language)

2010-01:
NXDL presented to ESRF HDF5 workshop on hyperspectral data

2009-09:
NXDL and draft NXsas (base class) presented to canSAS at SAS2009 conference

2009-04:
NeXus API version 4.2.0 is released with additional C++, IDL, and python/numpy interfaces.

2008-10:
NXDL: The NeXus Definition Language is defined. Until now, NeXus used another XML format, meta-DTD, for defining base classes and application definitions. There were several problems with meta-DTD, the biggest one being that it was not easy to validate against it. NXDL was designed to circumvent these problems. All current base classes and application definitions were ported into the NXDL.

2007-10:
NeXus API version 4.1.0 is released with many bug-fixes.

2007-05:
NeXus API version 4.0.0 is released with broader support for scripting languages and the feature to link with external files.

2005-07:
The community asked the NeXus team to provide an ASCII based physical file format which allows them to edit their scientific results in emacs. This lead to the development of a XML NeXus physical format. This was released with NeXus API version 3.0.0.

2003-10:
In 2003, NeXus had arrived at a stage where informal gatherings of a group of people were no longer good enough to oversee the development of NeXus. This lead to the formation of the NeXus International Advisory Committee (NIAC) which strives to include representatives of all major stake holders in NeXus. A first meeting was held at CalTech. Since 2003, the NIAC meets every year to discuss all matters NeXus.

2003-06:
Przemek Klosowski, Ray Osborn, and Richard Riedel received the only known grant explicitly for working on NeXus from the Systems Integration for Manufacturing Applications (SIMA) program of the National Institute of Standards and Technology (NIST). The grant funded a person for one year to work on community wide infrastructure in NeXus.

2002-09:
NeXus API version 2.0.0 is released. This version brought support for the new version of HDF, HDF5, released by the HDF group. HDF4 imposed limits on file sizes and the number of objects in a file. These issues were resolved with HDF5. The NeXus API abstracted the difference between the two physical file formats away form the user.

2001-summer:
MLNSC at LANL started writing NeXus files to store raw data

1997-07:
SINQ at PSI started writing NeXus files to store raw data.

1996-10:
At *SoftNeSS 1996* (at ANL), after reviewing the different scientific data formats discussed, it was decided to use HDF4 as it provided the best grouping support. The basic structure of a NeXus file was agreed upon. the various data format proposals were combined into a single document by Przemek Klosowski (NIST), Mark Könnecke (then ISIS), Jonathan Tischler (ORNL and APS/ANL), and Ray Osborn (IPNS/ANL) coauthored the first proposal for the NeXus scientific data standard.[1]

1996-08:
The HDF-4 API is quite complex. Thus a NeXus Abstract Programmer Interface NAPI was released which simplified reading and writing NeXus files.

1995-09:
At *SoftNeSS 1995* (at NIST), three individual data format proposals by Przemek Klosowski (NIST), Mark Könnecke (then ISIS), and Jonathan Tischler (ORNL and APS/ANL) were joined to form the basis of the current NeXus format. At this workshop, the name *NeXus* was chosen.

1994-10:
Ray Osborn convened a series of three workshops called *SoftNeSS*. In the first meeting, Mark Könnecke and Jon Tischler were invited to meet with representatives from all the major U.S. neutron scattering laboratories at Argonne National Laboratory to discuss future software development for the analysis and visualization of neutron data. One of the main recommendations of *SoftNeSS'94* was that a common data format should be developed.

1994-08:
Jonathan Tischler (ORNL) proposed an HDF-based format[2] as a standard for data storage at APS

[1] https://www.nexusformat.org/pdfs/NeXus_Proposal.pdf
[2] https://www.nexusformat.org/pdfs/Proposed_Data_Standard_for_the_APS.pdf

1994-06:

Mark Könnecke (then ISIS, now PSI) made a proposal using netCDF[3] for the European neutron scattering community while working at ISIS

[3] https://www.nexusformat.org/pdfs/European-Formats.pdf

CHAPTER

NINE

ABOUT THESE DOCS

9.1 Authors

Pete R. Jemian, Documentation Editor:
`<jemian@anl.gov>`, Advanced Photon Source, Argonne National Laboratory, Argonne, IL, USA,

Frederick Akeroyd:
`<freddie.akeroyd@stfc.ac.uk>`, Rutherford Appleton Laboratory, Didcot, UK,

Stuart I. Campbell:
`<campbellsi@ornl.gov>`, Oak Ridge National Laboratory, Oak Ridge, TN, USA,

Przemek Klosowski:
`<przemek.klosowski@nist.gov>`, U. of Maryland and NIST, Gaithersburg, MD, USA,

Mark Könnecke:
`<Mark.Koennecke@psi.ch>`, Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland,

Ray Osborn:
`<rosborn@anl.gov>`, Argonne National Laboratory, Argonne, IL, USA,

Peter F. Peterson:
`<petersonpf@ornl.gov>`, Spallation Neutron Source, Oak Ridge, TN, USA,

Tobias Richter:
`<Tobias.Richter@esss.se>`, European Spallation Source, Lund, Sweden,

Joachim Wuttke:
`<j.wuttke@fz-juelich.de>`, Forschungszentrum Jülich, Jülich Centre for Neutron Science at Heinz Maier-Leibnitz Zentrum Garching, Germany.

9.2 Colophon

These docs (manual and reference) were produced using Sphinx (http://sphinx-doc.org). The source for the manual shows many examples of the structures used to create the manual. If you have any questions about how to contribute to this manual, please contact the NeXus Documentation Editor (Pete Jemian <jemian@anl.gov>).

Note: The indentation is very important to the syntax of the restructured text manual source. Be careful not to mix tabs and spaces in the indentation or the manual may not build properly.

9.3 Revision History

Browse the most recent Issues on the GitHub repository: https://github.com/nexusformat/definitions/pulse/monthly

9.4 Copyright and Licenses

Publishing Information

This manual built Jun 24, 2022.

See also:

This document is available in these formats online:

HTML
: https://manual.nexusformat.org/

PDF
: https://manual.nexusformat.org/_static/NeXusManual.pdf

A very brief overview (title: *NeXus for the Impatient*) is also available (separate from the manual).

HTML
: https://manual.nexusformat.org/impatient/

PDF
: https://manual.nexusformat.org/_static/NXImpatient.pdf

INDEX