NeXus: A Common Data Format for Neutron, X-ray, and Muon Science

Release 3.1

NeXus International Advisory Committee
nexus@nexusformat.org
http://nexusformat.org

2016-12-20 11:38:17 GMT+00:00

CONTENTS

http://www.nexusformat.org/

CHAPTER

ONE

NEXUS: USER MANUAL

1.1 NeXus Introduction

NeXus 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 two 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.
2. *processed data*: NeXus defines standards in the form of *application definitions* for the exchange of data between applications. NeXus provides structures for raw experimental data as well as for 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.

1.1.1 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.

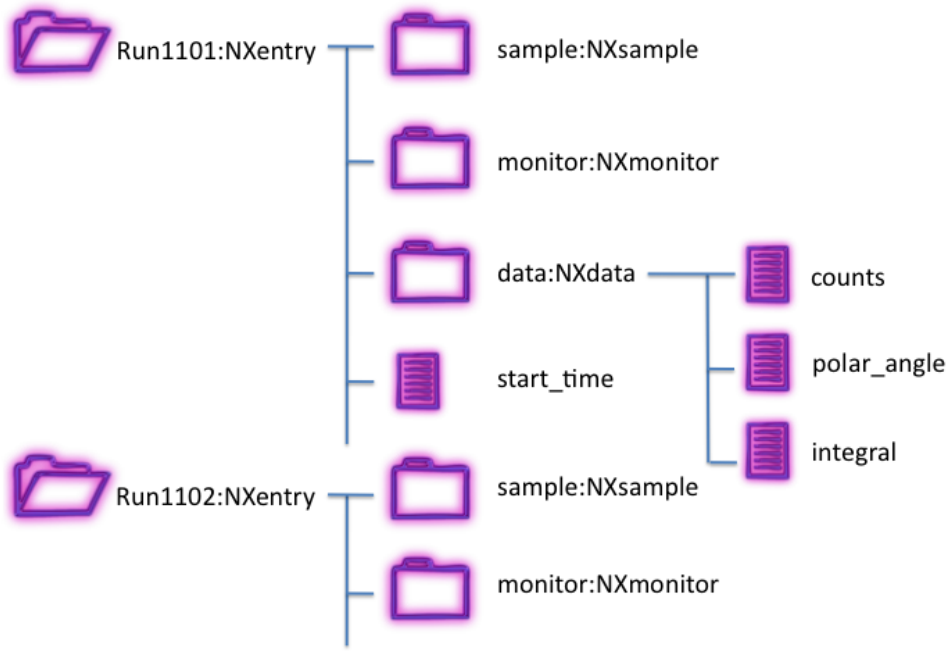
Links Links are used to reference the plottable data from `NXdata` when the data is provided in other groups such as `NXmonitor` or `NXdetector`.

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 Design* chapter.

Note: `NXentry` and `NXdata` are the only two classes necessary to store the minimum amount of information in a valid NeXus data file.

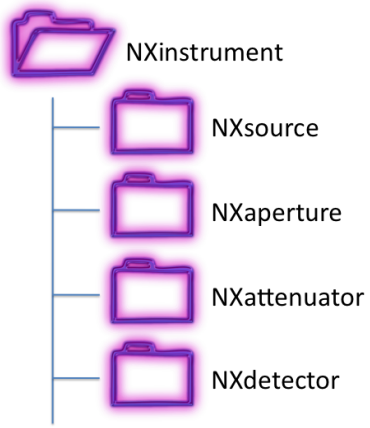
NXentry *Required:* The top level of any NeXus file contains one or more groups with the class `NXentry`. These contain all the data that is required to describe an experimental run or scan. Each `NXentry` typically contains a number of groups describing sample information (class `NXsample`), instrument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

NXdata *Required:* Each `NXentry` group contains one or more groups with class `NXdata`. 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

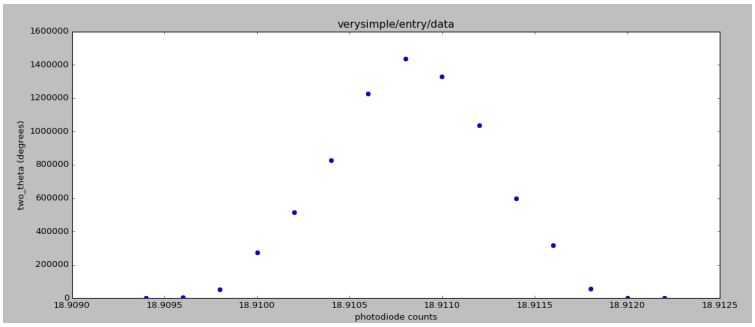
This illustrates the fact that the structure of NeXus files is extremely flexible. It can accommodate very complex instrumental information, if required, but it can also be used to store very simple data sets. Here is the structure of a very simple NeXus data file (`examples/verysimple.nx5`):

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 = two_theta (degrees)
      two_theta:float64[15] = [18.9094, 18.9096, '...', 18.9122]
        @units = degrees
        @long_name = photodiode counts
```

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

```
#!/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]

f = h5py.File('verysimple.nx5', 'w')
```

```
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'] = 'photodiode counts'

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

f.close()
```

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 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 through the NeXus wiki.

The *NeXus International Advisory Committee* (NIAC) supervises the development and maintenance of the NeXus common data format for neutron, X-ray, and muon science. The NIAC supervises a technical committee to oversee the *NAPI: NeXus Application Programmer Interface (frozen)* and the *Introduction to NeXus definitions*.

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

```
1  entry:NXentry
2    instrument:NXinstrument
3      detector:NXdetector
4        data:[]
5          @long_name = "strip detector 1-D array"
6        bins:[0, 1, 2, ... 1023]
7          @long_name = "bin index numbers"
8    sample:NXsample
9      name = "zeolite"
10   data:NXdata
11     @signal = "data"
12     @axes = ["bins", "bins"]
13     @bins_indices = [0, 1]
14     data --> /entry/instrument/detector/data
15     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 is required and 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* element 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*.

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 `NeXusIntern.pdf`, 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.

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. (Compare this example with *NAPI Simple 2-D Write Example (C, F77, F90)*, in the *NAPI* chapter, using the native HDF5 commands.) 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.

---

[1] http://download.nexusformat.org

```
#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.

[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*

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*.)

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)
  NX Data  : time_of_flight[751] (NX_FLOAT32)
  NX Data  : polar_angle[148] (NX_FLOAT32)
NX/Histogram1/data> read time_of_flight
  time_of_flight[751] (NX_FLOAT32) = [ 1900.000000 1902.000000 1904.000000 ...]
    units = microseconds
    long_name = Time-of-Flight [microseconds]
NX/Histogram1/data> read data
  data[148,750] (NX_INT32) = [ 1 1 0 ...]
    units = counts
    signal = 1
    long_name = Neutron Counts
    axes = polar_angle:time_of_flight
NX/Histogram1/data> close
NX/Histogram1> close
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.

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 in `NXdetector/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"`

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. 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 (4 x n x m) 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 (3 x n x m). Allowed values include:

- `scaler` (0-D data)
- `spectrum` (1-D data)
- `image` (2-D data)
- `rgba-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 *Python Examples using 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.

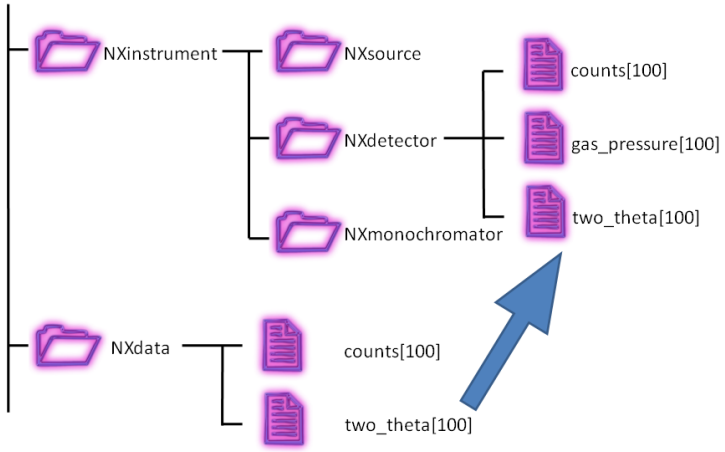


Fig. 1.1: Linking in a NeXus file

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

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

[1] 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?**

```
        ...
        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"
```

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"
}
```

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 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`, `NXparameter`). 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*, [2])**

`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*.

[2] 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 Coordinate Systems

The NeXus coordinate system is shown *below*. Note that it is the same as that used by *McStas* (http://mcstas.org).

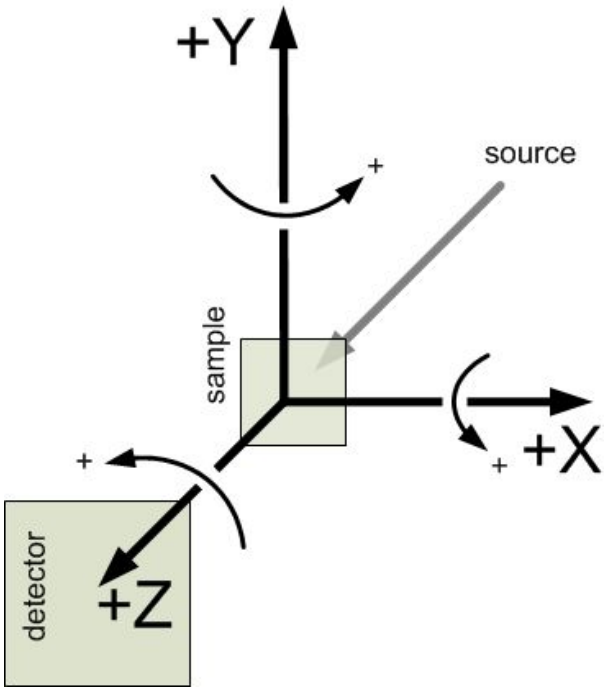


Fig. 1.2: NeXus coordinate system, as viewed from detector

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

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. These operations can be expressed as transformation matrices and their combination as matrix multiplication. A very important aspect is that the order of application of the individual operations *does* matter. Another important aspect is that any operation transforms the whole coordinate system and gives rise to a new local coordinate system. The mathematics behind this is well known and used in such applications such as industrial robot control, space flight 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.

NeXus chooses to encode this information in the following way:

Type Through a field attribute **transformation_type**. This can take the value of either *translation* or *rotation*.

Direction Through a field attribute **vector**. This is a set of three values describing either the components of the rotation axis or the direction along which the translation happens.

Value This is represented in the actual data of the data set. In addition, there is the **offset** attribute which has three components describing a translation to apply before applying the operation of the real axis. Without the offset attribute additional virtual translations would need to be introduced in order to encode mechanical offsets in the axis.

Order The order is encoded through the **depends_on** attribute on a data set. The value of the **depends_on** attribute is the axis upon which the current axis sits. If the axis sits in the same group it is just a name, if it is in another group it is a path to the dependent axis. In addition, for each beamline component, there is a **depends_on** field which points to the data set at the head of the axis dependency chain. Take as an example 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
  rotation_angle
    @transformation_type=rotation
    @vector=0,1,0
    @offset=0,0,0,
  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
    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.

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.

Coordinate Transformation Attributes

The coordinate transformation attributes are:

vector (*NX_FLOAT*) 3 values describing the axis of rotation or the direction of translation

offset (*NX_FLOAT*) 3 values describing a translation of the axis before applying the actual operation.

transformation_type Is either rotation or translation and describes the kind of operation performed

depends_on States the dataset which is next in the dependency chain. Allowed values for depends_on are:

. A dot ends the depends_on chain

name The name of a dataset within the enclosing group

dir/name The name of a dataset further along the path

/dir/dir/name An absolute path to a dataset in another group

Legacy Geometry Decriptions

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:

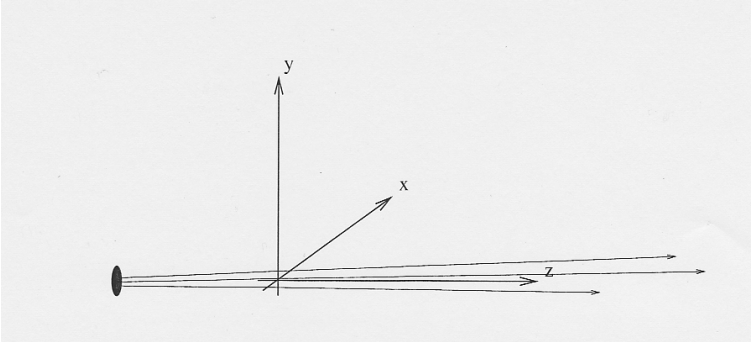


Fig. 1.3: The McStas Coordinate System

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*.

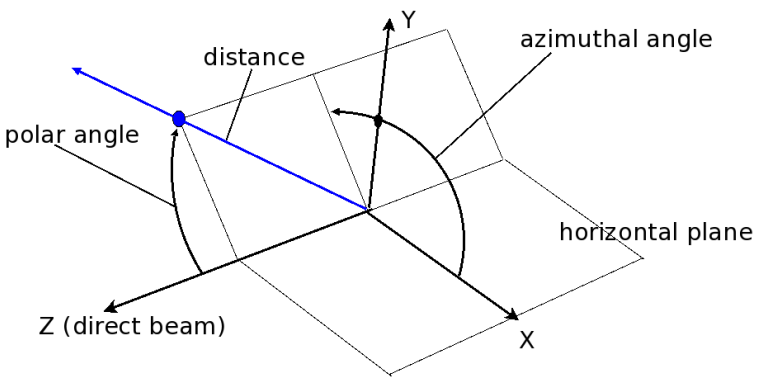


Fig. 1.4: NeXus Simple (Spherical Polar) Coordinate System

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:NXparameter
            filename = "sn2013287.nxs"
    sample:NXsample
    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 NXparameter 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 easily resolved. 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.
- 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
        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.
- Datasets in an NXdata group must contain the proper 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
        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
            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.

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 may be 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. 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*.

Regular expression pattern for NXDL group and field names

It is recommended that all group and field names contain only these characters:

- lower case letters
- digits
- "_" (underscore character)

and that they begin with a lower case letter. This is the regular expression used to check this recommendation.

[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.

```
[a-z_][a-z\d_]*
```

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 group and field names with upper case letters. *NeXus data files with upper case characters in the group or field names might not be accepted by all software that reads NeXus data files.* Hence, group and field names that do not pass the regular expression above but pass this expression (named *validItemName* in the XML Schema file: *nxdl.xsd*):

```
[A-Za-z_][\w_]*
```

will be flagged as a warning during data file validation.

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*.

NeXus Array Storage Order

NeXus stores multi-dimensional arrays of physical values in C language storage order, where the last dimension is the fastest varying. This is the rule. *Good reasons are required to deviate from this rule.*

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:
              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-z_][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 matches 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 All strings are to be encoded in UTF-8. Since most strings in a NeXus file are restricted to a small set of characters and the first 128 characters are standard across encodings, the encoding of most of the strings in a NeXus file will be a moot point. Where encoding in UTF-8 will be important is when recording people's names in `NXuser` and text notes in `NXnotes`.

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 [2] format, e.g. `1996-07-31T21:15:22+0600`. 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.

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`".

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

[2] ISO 8601: http://www.w3.org/TR/NOTE-datetime

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 [3] 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*

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.

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

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`, then its value is the name of the `NXentry` group to be used. Otherwise, pick any `NXentry` group. This is trivial if there is only one `NXentry` group.

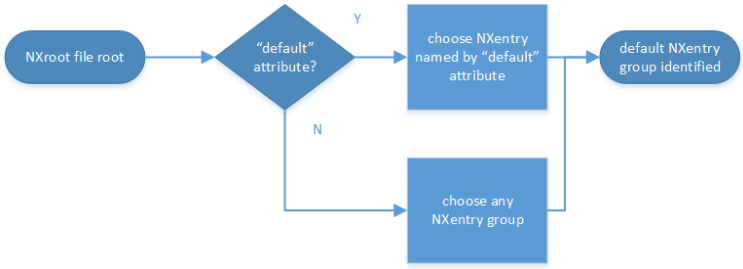


Fig. 1.5: Find plottable data: select the `NXentry` group

3. Pick the default *NXdata* group.

   Open the `NXentry` group selected above. If it has an attribute `default`, then its value is the name of the `NXdata` group to be used. Otherwise, pick any `NXdata` group. This is trivial if there is only one `NXdata` group.

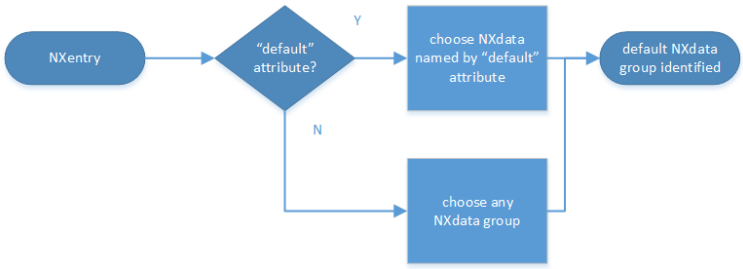


Fig. 1.6: Find plottable data: select the `NXdata` group

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

   Open the `NXdata` group selected above. If it has an attribute `signal`, then its value is the name of the field (dataset) to be plotted. If no `signal` attribute is not present on the `NXdata` group, then proceed to try an *older NeXus method* to find the default plottable data.

   (a) 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 *abcissae* 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.

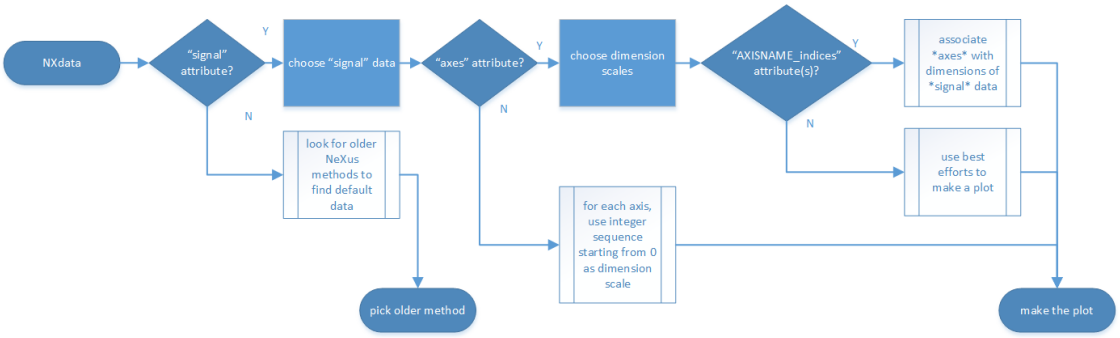


Fig. 1.7: Find plottable data: select the *signal* data

(b) 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.

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 abcissae 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*.

Version 2

Tip: Try this method for older NeXus data files.

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.

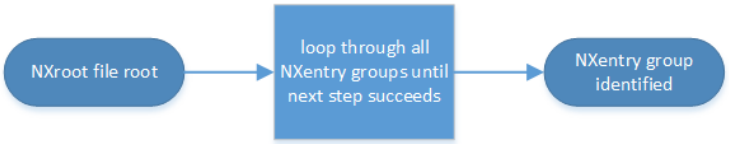


Fig. 1.8: Find plottable data: pick a `NXentry` group

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

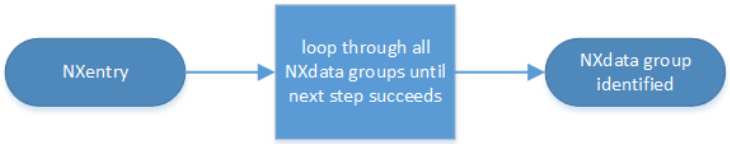


Fig. 1.9: Find plottable data: pick a `NXdata` group

(a) If this field has an attribute `axes`:

i. 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"`

ii. Parse `axes` and open the datasets to describe your dimension scales

(b) If this field has no attribute `axes`:

i. Search for datasets with attributes `axis=1`, `axis=2`, etc.

ii. 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.

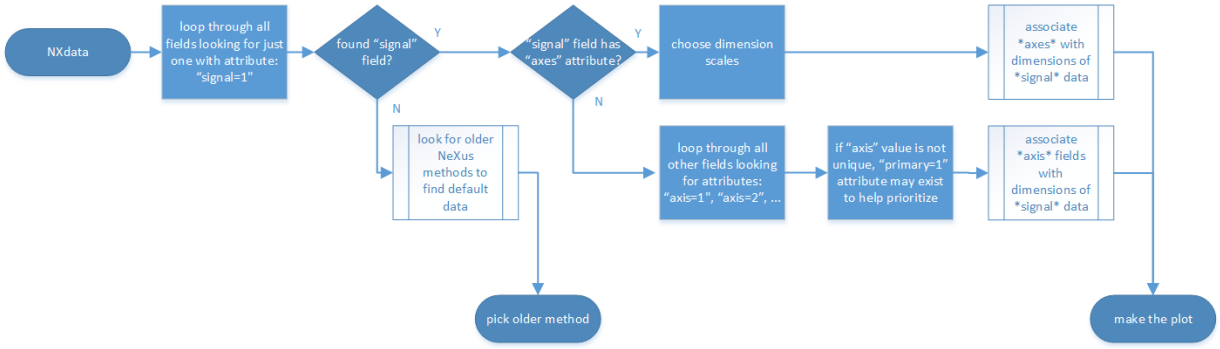


Fig. 1.10: Find plottable data: select the *signal* data

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

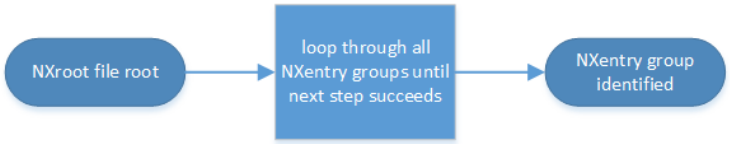


Fig. 1.11: Find plottable data: pick the first `NXentry` group

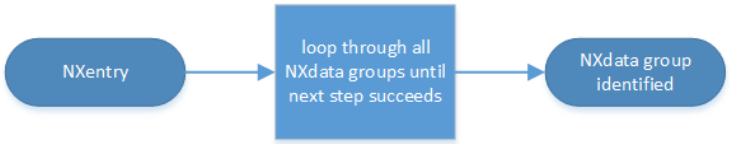


Fig. 1.12: Find plottable data: pick the first `NXdata` group

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).

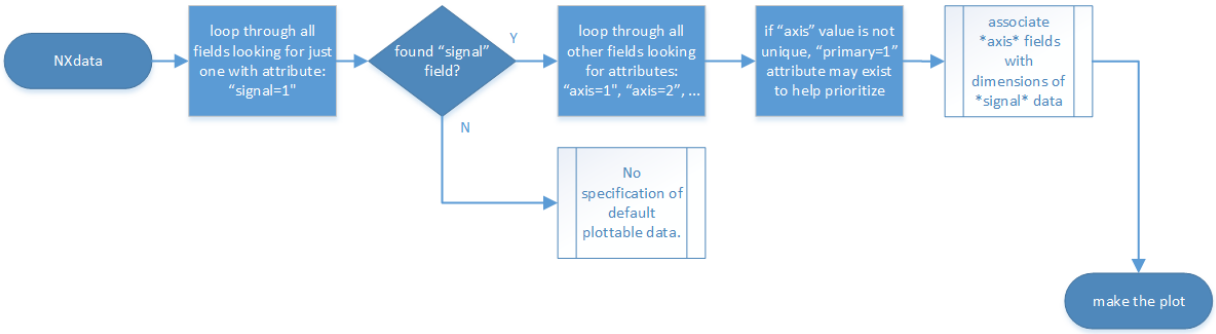


Fig. 1.13: Find plottable data: select the *signal* data

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.

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 dataset *in the NXdata group*. A field of this name *must* exist (either as dataset or link to dataset).

It is recommended to use this attribute rather than adding a signal attribute to the dataset. [4] The procedure 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 will 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 will have a `default` attribute that names the default *NXdata* group. This attribute may be omitted if there is only one NXdata group. 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 [5] 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 wiki ([6]).

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

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

[4] Summary of the discussion at NIAC2014 to revise how to find default data: http://wiki.nexusformat.org/2014_How_to_find_default_data

[5] 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.

[6] NIAC2014 proposition: http://wiki.nexusformat.org/2014_axes_and_uncertainties

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".

Examples

Several examples are provided to illustrate this method. More examples are available in the NeXus wiki ([6]).

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?*
- *NAPI Simple 2-D Write Example (C, F77, F90)*
- *Writing a simple NeXus file using native HDF5 commands in C*
- *Reading a simple NeXus file using native HDF5 commands in C*
- *Writing the HDF5 file using h5py*
- *Reading the HDF5 file using h5py*

Perhaps the easiest way to view the implementation of NeXus in HDF5 is to view how the data structures look. 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:

```
# --tba--
# TODO: write some text about HDF5 hard links
# until then, see the h5dump example below
```

See the sections *NAPI Simple 2-D Write Example (C, F77, F90)* and *NAPI Python Simple 3-D Write Example* in the *Examples of writing and reading NeXus data files* chapter for examples that use the native HDF5 calls to write NeXus data files.

`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

```
1
2  # NeXus links have two parts in HDF5 files.
3
4  # The dataset is created in some group.
5  # A "target" attribute is added to indicate the HDF5 path to this dataset.
6
7  ATTRIBUTE "target" {
8     DATATYPE  H5T_STRING {
9           STRSIZE 21;
10          STRPAD H5T_STR_NULLPAD;
11          CSET H5T_CSET_ASCII;
12          CTYPE H5T_C_S1;
13       }
14    DATASPACE  SCALAR
15    DATA {
16    (0): "/entry/data/two_theta"
17    }
18 }
19
20 # then, the hard link is created that refers to the original dataset
21 # (Since the name is "two_theta" in this example, it is understood that
22 # this link is created in a different HDF5 group than "/entry/data".)
23
24 DATASET "two_theta" {
25    HARDLINK "/entry/data/two_theta"
26 }
```

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, refer to *Code Examples that use the NeXus API (NAPI)* chapter. Alternatively, there are examples in the *Example NeXus C programs using native HDF5 commands* chapter of writing and reading NeXus data files using the native HDF5 interfaces in C. Further, there are also some Python examples using the `h5py` package in the *Python Examples using h5py* section.

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.

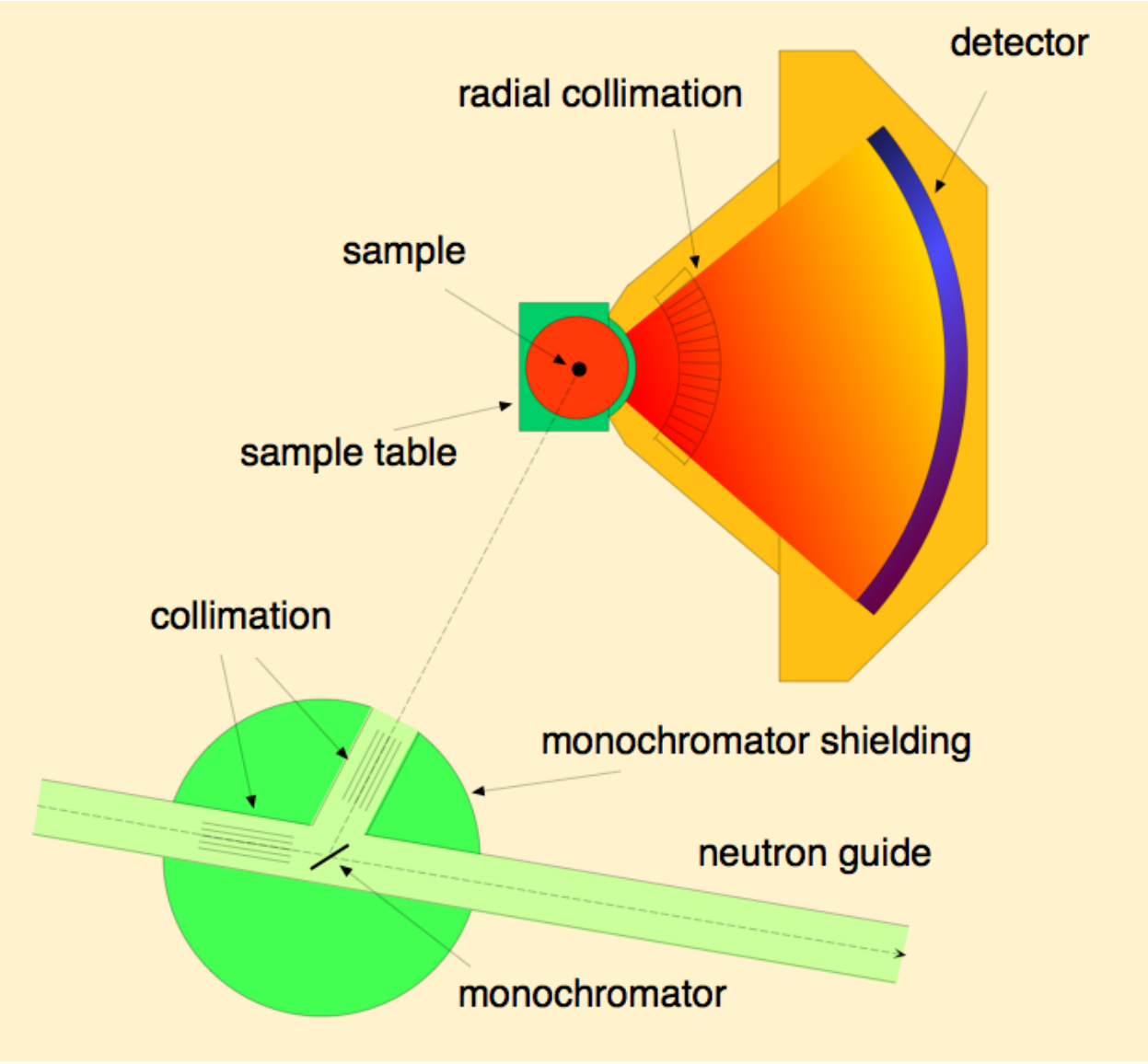


Fig. 1.14: The (fictional) WONI example powder diffractometer

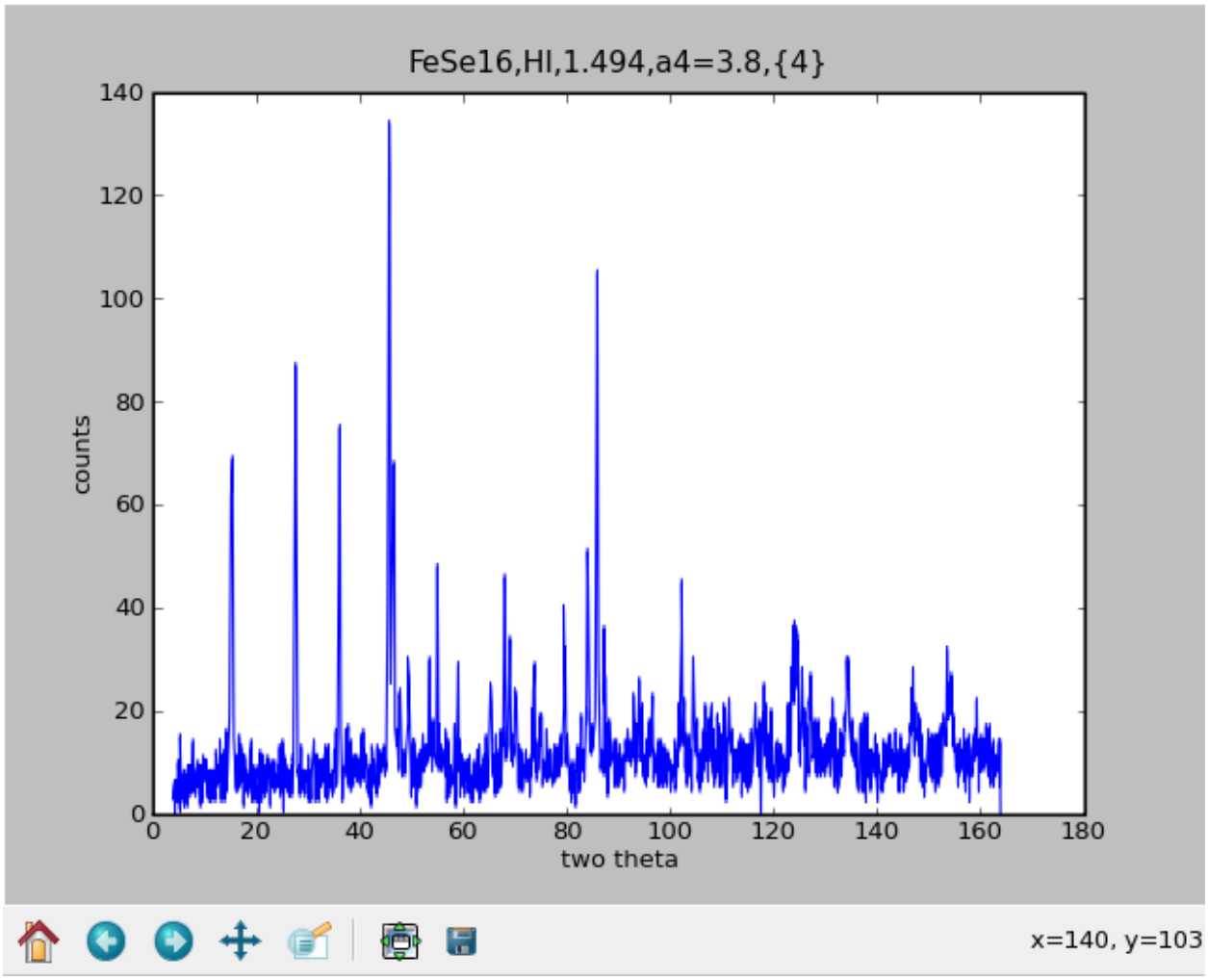


Fig. 1.15: Example Powder Diffraction Plot from (fictional) WONI at HYNES

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`
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
        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) 2008-2016 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
# 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 http://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. Also consider keeping track of /definition/@version as suits your development of this NXDL file.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="nxdlformat.xsl" ?>
<!--
# NeXus - Neutron and X-ray Common Data Format
#
```

```
# Copyright (C) 2008-2016 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
# 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 http://www.nexusformat.org
-->
<definition name="NXmonopd" 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>
        Monochromatic Neutron and X-Ray Powder diffractometer

        Instrument
        definition for a powder diffractometer at a monochromatic neutron
        or X-ray beam. This is both suited for a powder diffractometer
        with a single detector or a powder diffractometer with a position
        sensitive detector.
    </doc>
    <group type="NXentry" name="entry">
        <field name="title"/>
        <field name="start_time" type="NX_DATE_TIME"/>
        <field name="definition">
            <doc> Official NeXus NXDL schema to which this file conforms </doc>
            <enumeration>
                <item value="NXmonopd"/>
            </enumeration>
        </field>
        <group type="NXinstrument">
            <group type="NXsource">
                <field name="type"/>
                <field name="name"/>
                <field name="probe">
                    <enumeration>
                        <item value="neutron"/>
                        <item value="x-ray"/>
                        <item value="electron"/>
                    </enumeration>
                </field>
            </group>
            <group type="NXcrystal">
                <field name="wavelength" type="NX_FLOAT" units="NX_WAVELENGTH">
                    <doc>Optimum diffracted wavelength</doc>
```

```
                    <dimensions rank="1">
                        <dim index="1" value="i"/>
                    </dimensions>
                </field>
            </group>
            <group type="NXdetector">
                <field name="polar_angle" type="NX_FLOAT" axis="1">
                  <doc>where ndet = number of detectors</doc>
                  <dimensions rank="1">
                    <dim index="1" value="ndet" />
                  </dimensions>
                </field>
                <field name="data" type="NX_INT" signal="1">
                  <doc>
                    detector signal (usually counts) are already
                    corrected for detector efficiency
                  </doc>
                  <dimensions rank="1">
                    <dim index="1" value="ndet" />
                  </dimensions>
                </field>
            </group>
        </group>
        <group type="NXsample">
            <field name="name">
                <doc>Descriptive name of sample</doc>
            </field>
            <field name="rotation_angle" type="NX_FLOAT" units="NX_ANGLE">
                <doc>
                    Optional rotation angle for the case when the powder diagram
                    has been obtained through an omega-2theta scan like from a
                    traditional single detector powder diffractometer
                </doc>
            </field>
        </group>
        <group type="NXmonitor">
            <field name="mode">
                <doc>
                    Count to a preset value based on either clock time (timer)
                    or received monitor counts (monitor).
                </doc>
                <enumeration>
                    <item value="monitor"/>
                    <item value="timer"/>
                </enumeration>
            </field>
            <field name="preset" type="NX_FLOAT">
                <doc>preset value for time or monitor</doc>
            </field>
            <field name="integral" type="NX_FLOAT" units="NX_ANY">
                <doc>Total integral monitor counts</doc>
            </field>
        </group>
        <group type="NXdata">
            <link name="polar_angle" target="/NXentry/NXinstrument/NXdetector/polar_
↪angle">
                <doc>Link to polar angle in /NXentry/NXinstrument/NXdetector</doc>
            </link>
```

```
            <link name="data" target="/NXentry/NXinstrument/NXdetector/data">
                <doc>Link to data in /NXentry/NXinstrument/NXdetector</doc>
            </link>
        </group>
    </group>
</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 in.

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:

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

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
```

More recently, this location makes more sense to many:

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

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: 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

The *cnxvalidate* utility [1], new in 2016, is available for testing. For the moment, the most recent documentation is served from the GitHub web site.

This utility only works on HDF5 files and is aimed to be faster, simpler, more portable and robust than previous programmes for NeXus file validation.

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 Wiki page. Please have a look at this list. Keep in mind that it is not complete.

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 *h5toText* 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

[1] *cnxvalidate*: from https://github.com/nexusformat/cnxvalidate

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 configuration, please contact the NIAC Executive Secretary explaining the problem, and post a suggestion at the relevant class wiki page. Or raise the problem in one of the mailing lists. 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`?

   `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*.

1. 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.

2. 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.

3. 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.

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

[1] Refer to the most recent version of the NIAC constitution on the NIAC wiki: http://wiki.nexusformat.org/NIAC#Constitution

---

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.

5. **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.

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

   See the *Strategies: The wavelength* section.

7. Where do I store metadata about my experiment?

   See the *Where to Store Metadata* section.

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. Here, three examples are provided showing how to write a NeXus data file without using the NAPI.

2.1 Code Examples in Various Languages

Each example in this section demonstrates either reading NeXus files in one of the supported storage containers (HDF5 or one of the legacy container formats: HDF4 or XML) or writing compliant NeXus files in the HDF5 storage containers. Please be aware that not all exmples are up to date with the latest format recommendations.

2.1.1 Example NeXus C programs using native HDF5 commands

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. Compare these code examples with *Example NeXus programs using NAPI*.

Writing a simple NeXus file using native HDF5 commands in C

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*.

```
/**
 * 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
 *         counts[]
 *            @signal=1
 *         two_theta[]
 *            @units=degrees
 *
 *  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>

#define LENGTH 400
int main(int argc, char *argv[])
{
  float two_theta[LENGTH];
  int counts[LENGTH], i, rank, signal;

  /* HDF-5 handles */
  hid_t fid, fapl, gid, atts, atttype, attid;
  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, (const char *)"scan",0);
  /*
   * store the NX_class attribute. Notice that you
   * have to take care to close those hids after use
   */
  atts = H5Screate(H5S_SCALAR);
  atttype = H5Tcopy(H5T_C_S1);
  H5Tset_size(atttype, strlen("NXentry"));
```

```
  attid = H5Acreate(gid,"NX_class", atttype, atts, H5P_DEFAULT);
  H5Awrite(attid, atttype, (char *)"NXentry");
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);

  /*
   * same thing for data:Nxdata in scan:NXentry.
   * A subroutine would be nice to have here.......
   */
  gid = H5Gcreate(fid, (const char *)"/scan/data",0);
  atts = H5Screate(H5S_SCALAR);
  atttype = H5Tcopy(H5T_C_S1);
  H5Tset_size(atttype, strlen("NXdata"));
  attid = H5Acreate(gid,"NX_class", atttype, atts, H5P_DEFAULT);
  H5Awrite(attid, atttype, (char *)"NXdata");
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);

  /*
   * store the counts dataset
   */
  dataspace = H5Screate_simple(rank,dim,maxdim);
  datatype = H5Tcopy(H5T_NATIVE_INT);
  dataprop = H5Pcreate(H5P_DATASET_CREATE);
  dataid = H5Dcreate(gid,(char *)"counts",datatype,dataspace,dataprop);
  H5Dwrite(dataid, datatype, H5S_ALL, H5S_ALL, H5P_DEFAULT, counts);
  H5Sclose(dataspace);
  H5Tclose(datatype);
  H5Pclose(dataprop);
  /*
   * set the signal=1 attribute
   */
  atts = H5Screate(H5S_SCALAR);
  atttype = H5Tcopy(H5T_NATIVE_INT);
  H5Tset_size(atttype,1);
  attid = H5Acreate(dataid,"signal", atttype, atts, H5P_DEFAULT);
  signal = 1;
  H5Awrite(attid, atttype, &signal);
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);

  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,(char *)"two_theta",datatype,dataspace,dataprop);
  H5Dwrite(dataid, datatype, H5S_ALL, H5S_ALL, H5P_DEFAULT, two_theta);
  H5Sclose(dataspace);
  H5Tclose(datatype);
  H5Pclose(dataprop);

```

```
  /*
   * set the units attribute
   */
  atttype = H5Tcopy(H5T_C_S1);
  H5Tset_size(atttype, strlen("degrees"));
  atts = H5Screate(H5S_SCALAR);
  attid = H5Acreate(dataid,"units", atttype, atts, H5P_DEFAULT);
  H5Awrite(attid, atttype, (char *)"degrees");
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);
 /*
   * set the target attribute for linking
   */
  atttype = H5Tcopy(H5T_C_S1);
  H5Tset_size(atttype, strlen("/scan/data/two_theta"));
  atts = H5Screate(H5S_SCALAR);
  attid = H5Acreate(dataid,"target", atttype, atts, H5P_DEFAULT);
  H5Awrite(attid, atttype, (char *)"/scan/data/two_theta");
  H5Sclose(atts);
  H5Tclose(atttype);
  H5Aclose(attid);


  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
   */
  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 datatype, 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");
  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);

  /*
   * open and read the two_theta data set
   */
  dataid = H5Dopen(fid,"/scan/data/two_theta");
  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 Python Examples using `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.

The data shown plotted in the next figure will be written to the NeXus HDF5 file using the only two required NeXus objects `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. 2.1: data structure, (from Introduction)

our h5py example

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

two-column data for our *mr_scan*

```
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
```

[1] *h5py*: http://code.google.com/p/h5py
[2] *SPEC*: http://certif.com/spec.html

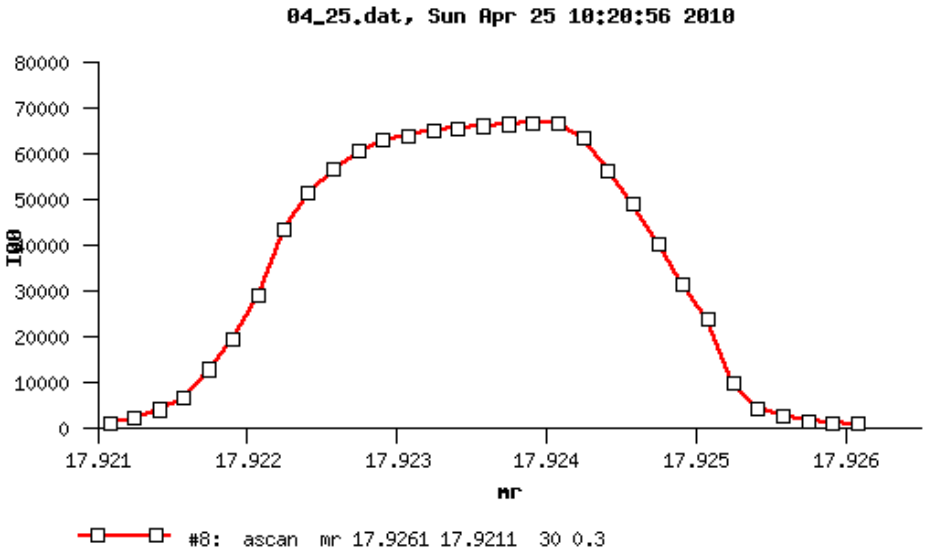


Fig. 2.2: plot of our *mr_scan*

```
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 the simplest data using `h5py`

These two examples show how to write the simplest data (above). One example writes the data directly to the *NXdata* group while the other example writes the data to `NXinstrument/NXdetector/data` and then creates a soft link to that data in `NXdata`.

`h5py` example writing the simplest NeXus data 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.* NeXus also requires that each `NXentry` group must contain at least one *NXdata* group. `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.

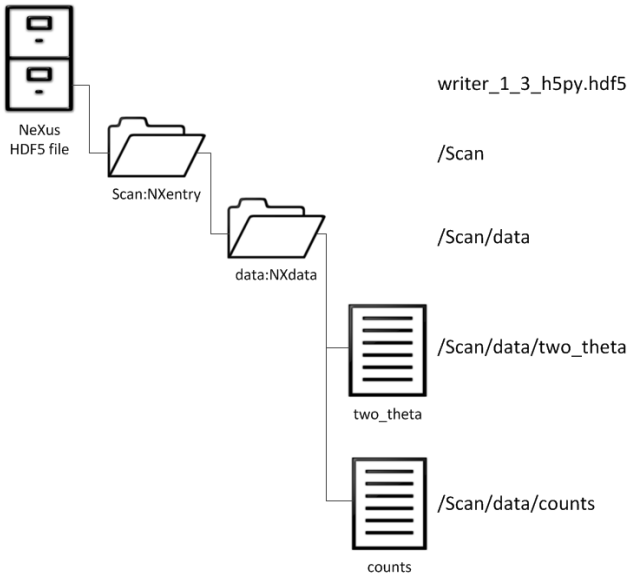


Fig. 2.3: Simple Example

In the *above figure*, the data file (`writer_1_3_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 (`writer_1_3_h5py.py`) is run from the command line (and there are no problems), the `writer_1_3_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
'''

import h5py
import numpy

buffer = numpy.loadtxt('input.dat').T
tthData = buffer[0]                             # float[]
countsData = numpy.asarray(buffer[1],'int32')   # int[]

f = h5py.File('writer_1_3.hdf5', "w")  # 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"

f.close()    # be CERTAIN to close the file
```

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 writer_1_3.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 *h5toText* 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:

```
writer_1_3.hdf5 : NeXus data file
  Scan:NXentry
    @NX_class = NXentry
    data:NXdata
      @NX_class = NXdata
      @signal = counts
      @axes = two_theta
      @two_theta_indices = [0]
      counts:NX_INT32[31] = [1037, 1318, 1704, '...', 1321]
        @units = counts
      two_theta:NX_FLOAT64[31] = [17.926079999999999, 17.925909999999998, 17.
↪925750000000001, '...', 17.92108]
        @units = degrees
```

As the data files in these examples become more complex, you will appreciate the information density provided by *h5toText*.

downloads

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

file	description
`writer_1_3.py`	python code to write example *writer_1_3*
`writer_1_3.hdf5`	NeXus file written by this code
`writer_1_3_h5dump.txt`	*h5dump* analysis of the NeXus file
`writer_1_3_structure.txt`	*h5toText* analysis of the NeXus file

[1] *h5toText* : http://spec2nexus.readthedocs.org/en/latest/h5toText.html

`h5py` example writing a simple NeXus data file with links

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. Still, NeXus requires 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 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.

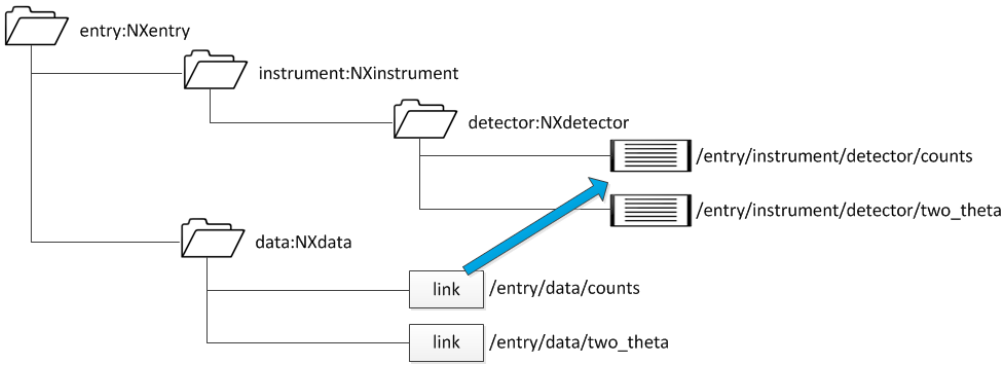


Fig. 2.4: h5py example showing linking in a NeXus file

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

```
#!/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
'''

import h5py
import numpy

buffer = numpy.loadtxt('input.dat').T
tthData = buffer[0]                                 # float[]
countsData = numpy.asarray(buffer[1],'int32')    # int[]

f = h5py.File('writer_2_1.hdf5', "w")  # 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'

# 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._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._id.link(source_addr, target_addr, h5py.h5g.LINK_HARD)

f.close()    # be CERTAIN to close the file
```

It is interesting to compare the output of the `h5dump` of the data file `writer_2_1.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 `h5toText.py` tool knows about the additional NeXus `target` attribute and shows the data to appear in its original location, in the `NXdetector` group.

```
writer_2_1.hdf5
  @default = entry
  entry:NXentry
    @NX_class = NXentry
    @default = data
    data:NXdata
      @NX_class = NXdata
      @signal = counts
      @axes = two_theta
      @two_theta_indices = [0]
      counts --> /entry/instrument/detector/counts
      two_theta --> /entry/instrument/detector/two_theta
```

```
    instrument:NXinstrument
      @NX_class = NXinstrument
      detector:NXdetector
        @NX_class = NXdetector
        counts:int32[31] = [1037, 1318, 1704, '...', 1321]
          @units = counts
          @target = /entry/instrument/detector/counts
        two_theta:float64[31] = [17.926079999999999, 17.925909999999998, 17.
↪925750000000001, '...', 17.92108]
          @units = degrees
          @target = /entry/instrument/detector/two_theta
```

downloads

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

file	description
`writer_2_1.py`	python code to write example *writer_2_1*
`writer_2_1.hdf5`	NeXus file written by this code
`writer_2_1_h5dump.txt`	*h5dump* analysis of the NeXus file
`writer_2_1_structure.txt`	*h5toText* analysis of the NeXus file

Complete `h5py` example writing and reading a NeXus data file

Writing the HDF5 file using h5py

In the main code section of *BasicWriter.py*, a current time stamp is written in the format of *ISO 8601* (`yyyy-mm-ddTHH:MM:SS`). For simplicity of this code example, we use a text string for the time, rather than computing it directly from Python support library calls. It is easier this way to see the exact type of string formatting for the time. When using the Python `datetime` package, one way to write the time stamp is:

```
timestamp = "T".join( str( datetime.datetime.now() ).split() )
```

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/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.

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

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

import h5py    # HDF5 support
import numpy

print "Write a NeXus HDF5 file"
fileName = "prj_test.nexus.hdf5"
timestamp = "2010-10-18T17:17:04-0500"

# load data from two column format
data = numpy.loadtxt('input.dat').T
mr_arr = data[0]
i00_arr = numpy.asarray(data[1],'int32')

# create the HDF5 NeXus file
f = h5py.File(fileName, "w")
# 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']          = 'BasicWriter.py'
f.attrs['NeXus_version']    = '4.3.0'
f.attrs['HDF5_Version']     = h5py.version.hdf5_version
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

f.close()    # be CERTAIN to close the file

print "wrote file:", fileName
```

Reading the HDF5 file using h5py

The file reader, *BasicReader.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`.

BasicReader.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 = "prj_test.nexus.hdf5"
f = h5py.File(fileName,  "r")
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])
f.close()
```

Output from `BasicReader.py` is shown next.

Output from `BasicReader.py`

```
file_name: prj_test.nexus.hdf5
file_time: 2010-10-18T17:17:04-0500
creator: BasicWriter.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
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
```

Plotting the HDF5 file

Now that we are certain our file conforms to the NeXus standard, let's plot it using the `NeXpy` [3] client tool. To help label the plot, we added the `long_name` attributes to each of our datasets. We also added metadata to the root level of our HDF5 file similar to that written by the NAPI. It seemed to be a useful addition. Compare this with *plot of our mr_scan* 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.

Links to Data in External HDF5 Files

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.

[3] *NeXpy*: http://nexpy.github.io/nexpy/

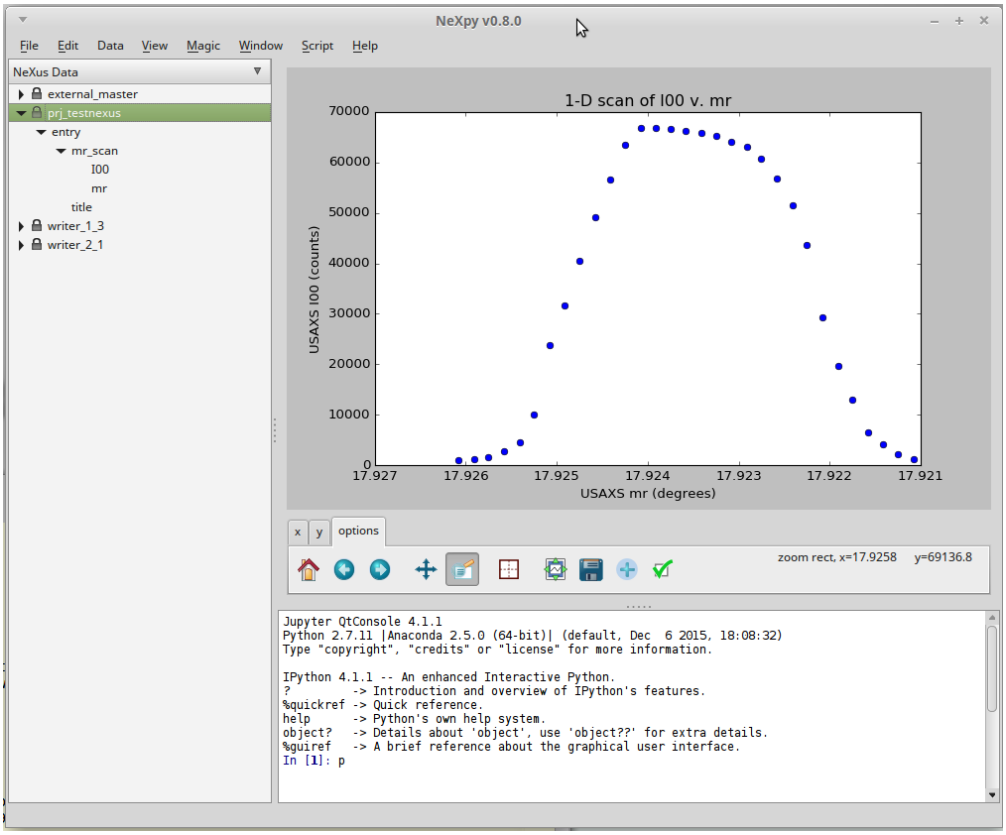


Fig. 2.5: plot of our *mr_scan* using NeXpy

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. [4] 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"
```

Note: The file `external_counts.hdf5` is not a complete NeXus file since it does not contain an NXdata group containing a dataset named by the NXdata group `signal` attributed.

---

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 containing a *NXdata* group containing a dataset that is named as the value of the group attribute `signal={dataset_name}`. 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
```

[4] see these URLs for further guidance on HDF5 external links: http://www.hdfgroup.org/HDF5/doc/RM/RM_H5L.html#Link-CreateExternal, http://www.h5py.org/docs-1.3/guide/group.html#external-links

```
    @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"
```

source code: externalExample.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.

externalExample.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``.
'''

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
buffer = numpy.loadtxt('input.dat').T
tthData = buffer[0]                             # float[]
countsData = numpy.asarray(buffer[1],'int32')   # int[]

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


# put the detector counts in an external HDF5 data file
# with *incomplete* NeXus structure (no NXdata group)
f = h5py.File(FILE_HDF5_COUNTS, "w")
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')
f.close()

# create a master NeXus HDF5 file
f = h5py.File(FILE_HDF5_MASTER, "w")
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')

f.close()
```

downloads

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

file	description
`input.dat`	2-column ASCII data used in this section
`BasicReader.py`	python code to read example *prj_test.nexus.hdf5*
`BasicWriter.py`	python code to write example *prj_test.nexus.hdf5*
`external_angles_h5dump.txt`	*h5dump* analysis of *external_angles.hdf5*
`external_angles.hdf5`	HDF5 file written by *externalExample*
`external_angles_structure.txt`	*h5toText* analysis of *external_angles.hdf5*
`external_counts_h5dump.txt`	*h5dump* analysis of *external_counts.hdf5*
`external_counts.hdf5`	HDF5 file written by *externalExample*
`external_counts_structure.txt`	*h5toText* analysis of *external_counts.hdf5*
`externalExample.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 *externalExample*
`external_master_structure.txt`	*h5toText* analysis of *external_master.hdf5*
`prj_test.nexus_h5dump.txt`	*h5dump* analysis of the NeXus file
`prj_test.nexus.hdf5`	NeXus file written by *BasicWriter*
`prj_test.nexus_structure.txt`	*h5toText* analysis of the NeXus file

2.1.3 MATLAB Examples

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 *Python Examples using 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');


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');

% 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);

% 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

2.1.4 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. For this example, we will start with a conversion of that original data file into *HDF5* format.

HDF4 http://svn.nexusformat.org/definitions/exampledata/IPNS/LRMECS/lrcs3701.nxs

HDF5 http://svn.nexusformat.org/definitions/exampledata/IPNS/LRMECS/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`):

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 (http://www.hdfgroup.org/hdf-java-html/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.

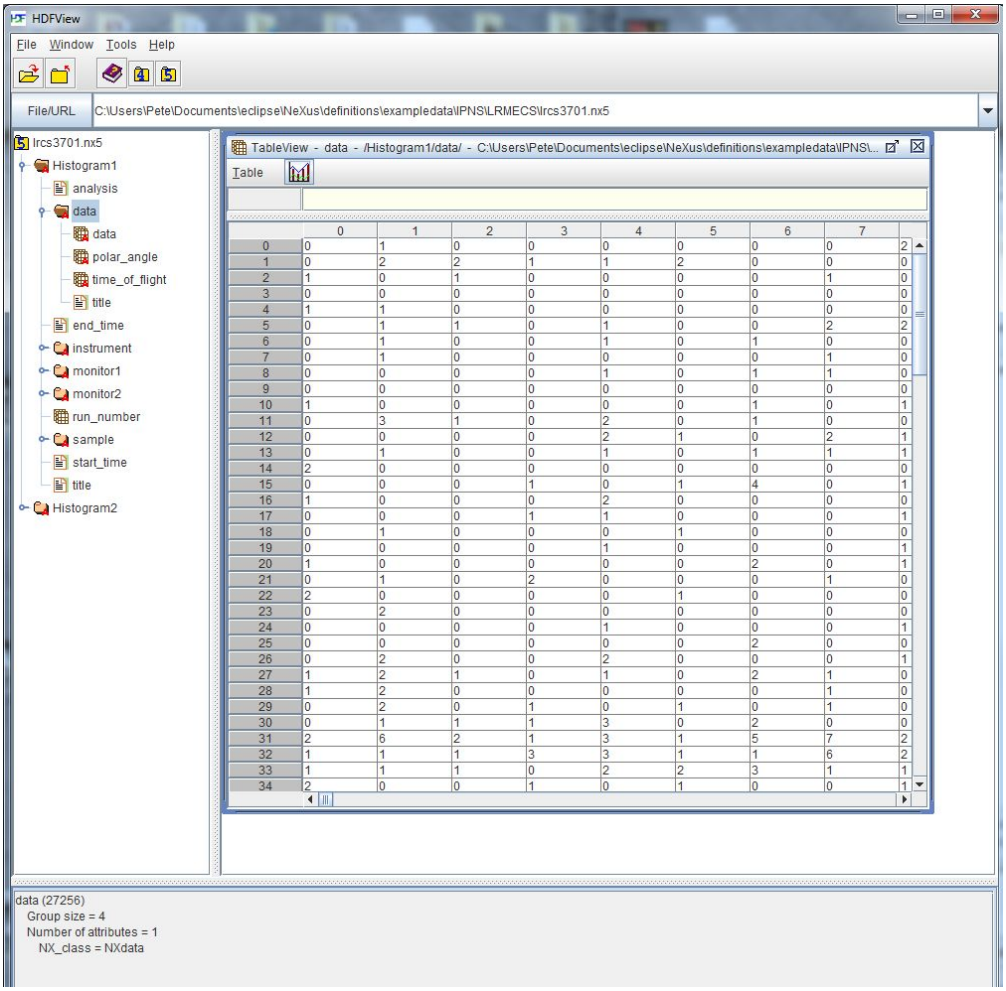


Fig. 2.6: LRMECS `lrcs3701` data: *HDFview*

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.

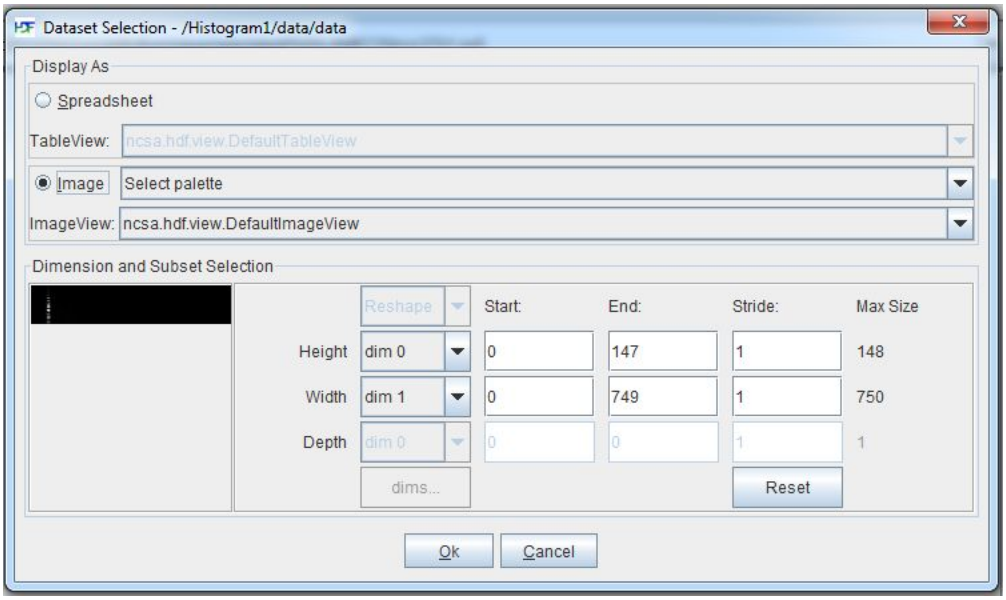


Fig. 2.7: LRMECS `lrcs3701` data: *HDFview Open As* dialog

LRMECS `lrcs3701` data: image

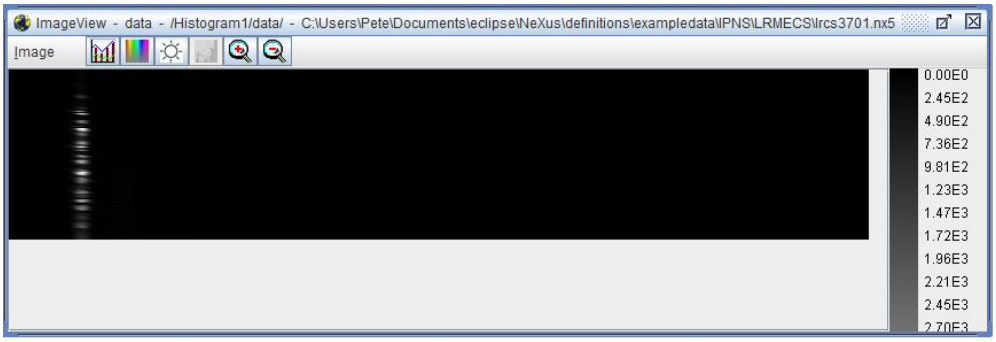


Fig. 2.8: LRMECS `lrcs3701` data: *HDFview* Image

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.

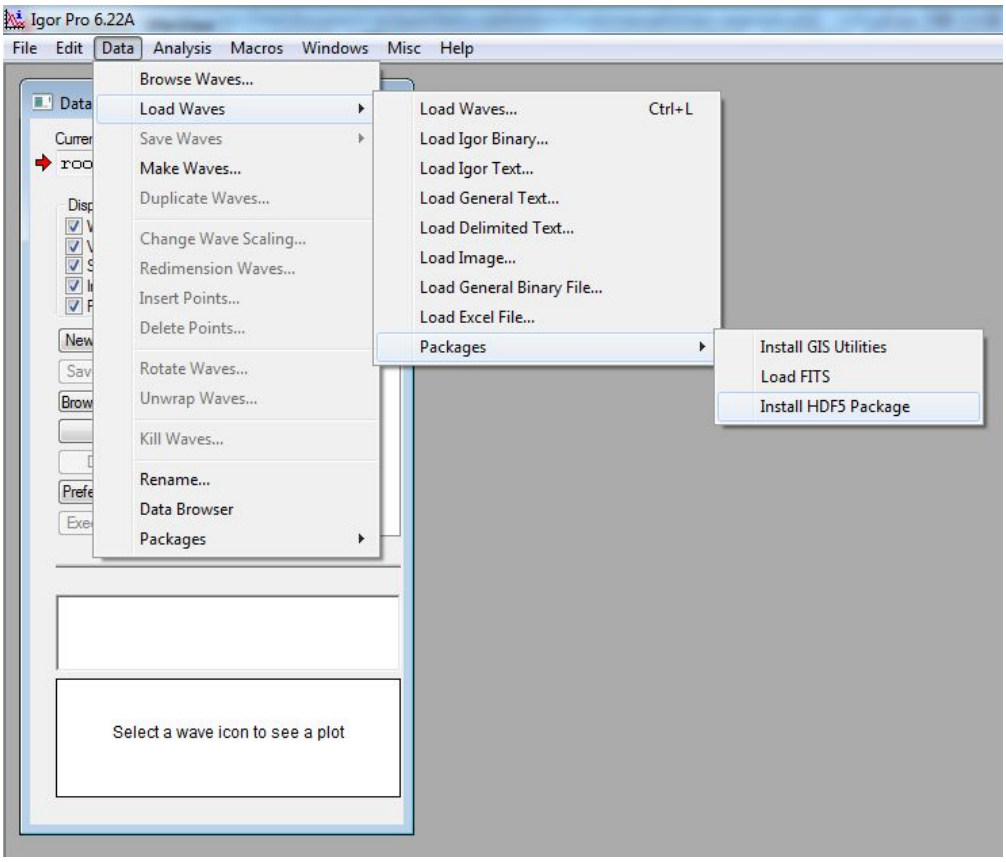


Fig. 2.9: LRMECS `lrcs3701` data: *IgorPro* install HDF5 Browser

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

2.2 Code Examples that use the NeXus API (NAPI)

These examples illustrate the use of the NAPI *NAPI: NeXus Application Programmer Interface (frozen)*. Please refer to the linked section in the manual for the status of NAPI.

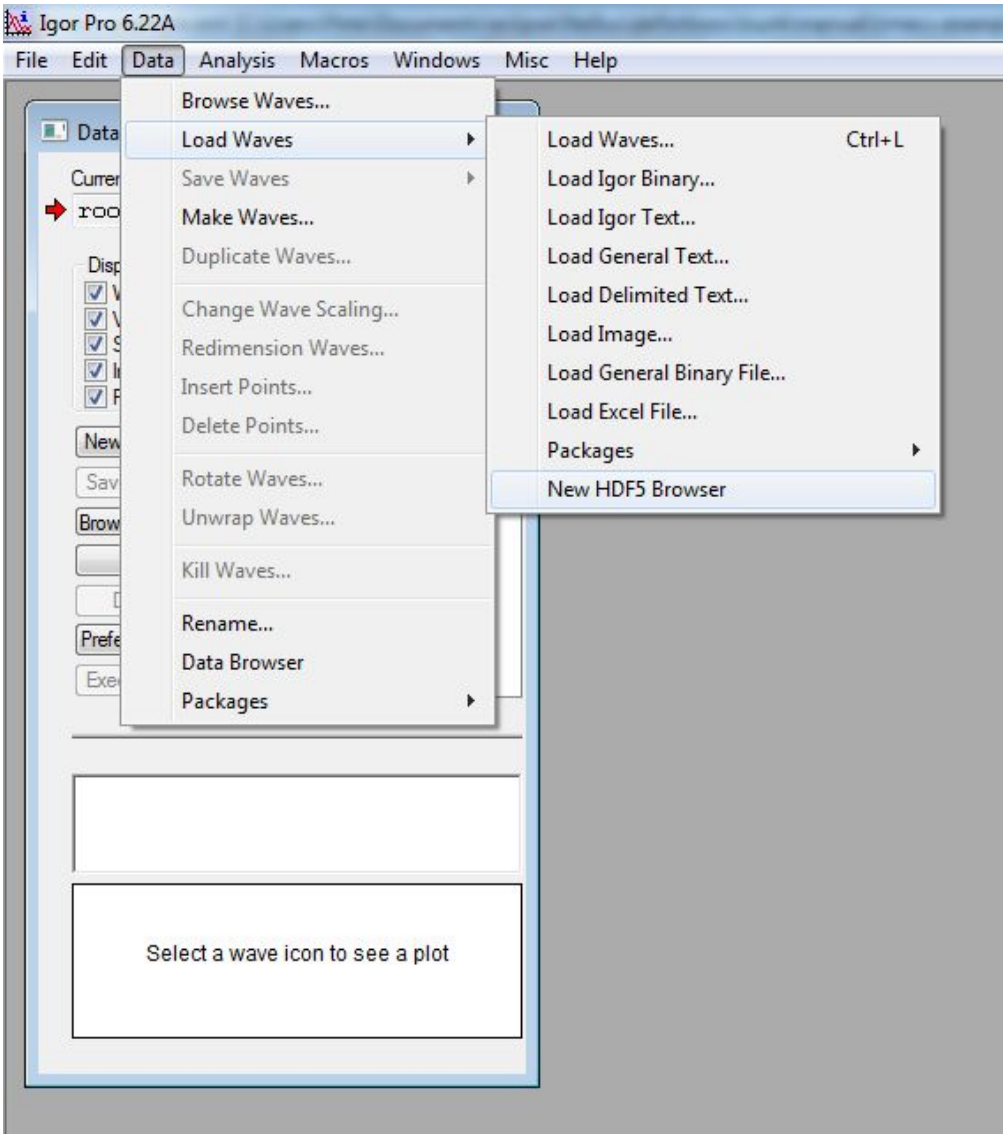


Fig. 2.10: LRMECS `lrcs3701` data: *IgorPro HDFBrowser* dialog

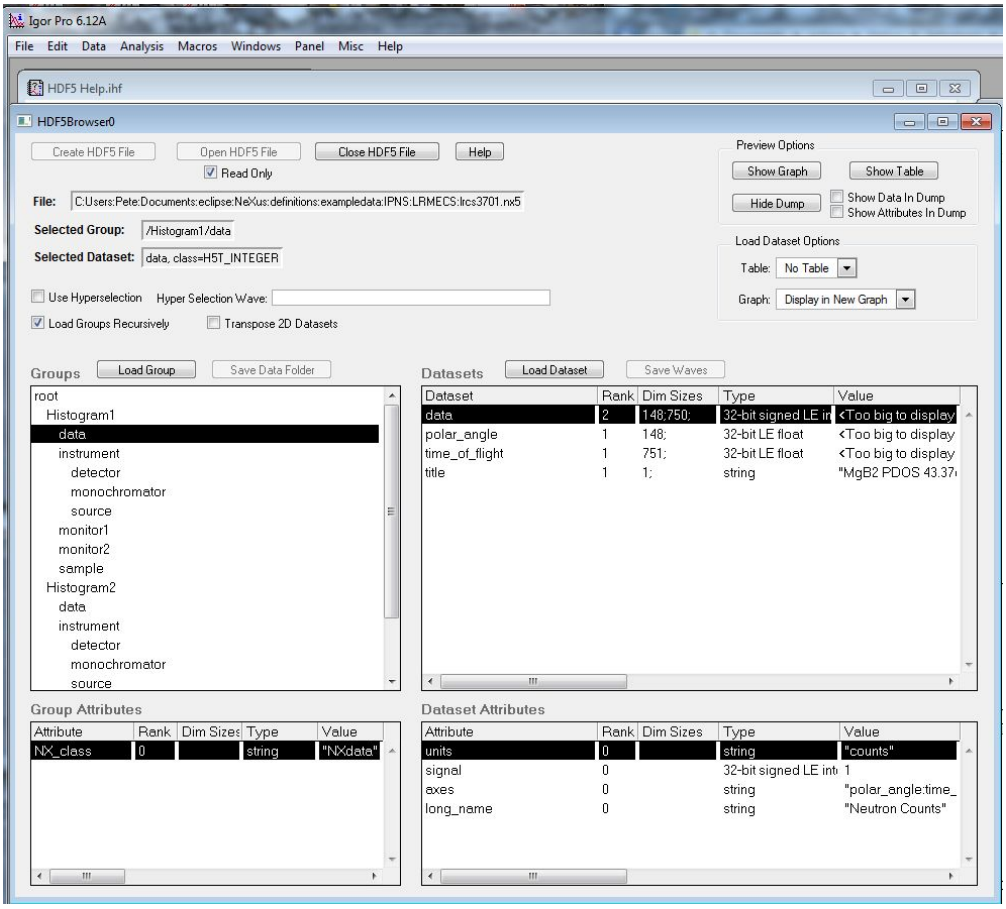


Fig. 2.11: LRMECS `lrcs3701` data: *IgorPro HDFBrowser* dialog

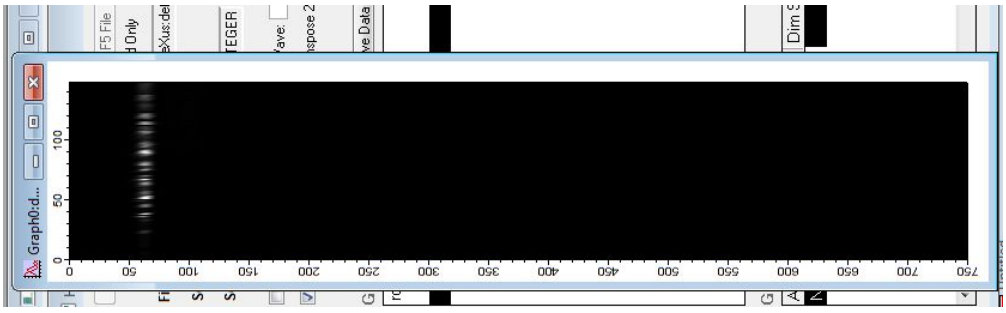


Fig. 2.12: LRMECS `lrcs3701` data: *IgorPro* Image

2.2.1 Example NeXus programs using NAPI

NAPI Simple 2-D Write Example (C, F77, F90)

Code examples are provided in this section that write 2-D data to a NeXus HDF5 file in C, F77, and F90 languages using the NAPI.

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. The same code is provided in C, F77, and F90 versions. Compare these code examples with *Example NeXus C programs using native HDF5 commands*.

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*.

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

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')
              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
         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
```

NAPI Python Simple 3-D Write Example

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.

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)
nf.closedata()

nf.closegroup() # NXdata
nf.closegroup() # NXentry

nf.close()

exit
```

View a NeXus HDF5 file using *h5dump*

For the purposes of an example, it is instructive to view the content of the NeXus HDF5 file produced by the above program. Since HDF5 is a binary file format, we cannot show the contents of the file directly in this manual. Instead, we first we view the content by showing the output from the `h5dump` tool provided as part of the HDF5 tool kit: `h5dump simple3D.h5`

NAPI Python Example: `h5dump` output of NeXus HDF5 file

```
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;
            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,
            (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
               }
            }
         }
      }
   }
}
}
```

View a NeXus HDF5 file using *h5toText.py*

The output of `h5dump` contains a lot of structural information about the HDF5 file that can distract us from the actual content we added to the file. Next, we show the output from a custom Python tool (`h5toText.py`) built for this

documentation and later moved into the **spec2nexus** package. [1] This tool was developed to show the actual data content of an HDF5 file that we create.

NAPI Python Example: `h5toText` output of NeXus HDF5 file

```
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] **spec2nexus** : http://spec2nexus.readthedocs.org

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:

1. **short table**:
   - the current version of the NXDL specification used for the class
   - the category of the class (base class / application definition / contributed definition)
   - The NeXus class extended by this class. Most NeXus base classes only extend the base class definition (NXDL).
   - any other base classes (groups) cited by this class
2. **symbol list:** keywords used to designate array dimensions. At present, this list is not guaranteed to be complete (some array dimension names appear only in the description column of the class member table, and not here)
3. **source:** a link to the authorative NXDL source
4. **tree outline:** hierarchical list of members.

5. **member table:** list of top-level members with natural-language annotations.
6. **supplementary member tables as needed:** member tables of subgroups.

3.1.2 Tree outlines

A compact listing 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.

3.1.3 Member tables

Member tables provide basic information about each field or group in the class. An example of the varieties of specifications are given in the following table using items found in various NeXus base classes.

Name	Type	Units	Description (and Occurrences)
`program_name`	NX_CHAR		Name of program used to generate this file
`@version`	NX_CHAR		Program version number Occurences: 1 : *default*
`@configuration`	NX_CHAR		configuration of the program
`thumbnail`	*NXnote*		A small image that is representative of the entry. An example of this is a 640x480 JPEG image automatically produced by a low resolution plot of the NXdata.
`@mime_type`	NX_CHAR		expected: *mime_type="image/*"*
	NXgeometry		describe the geometry of this class
`distance`	NX_FLOAT	NX_LENGTH	Distance from sample
`mode`	"Single Bunch" | "Multi Bunch"		source operating mode
`target_material`	Ta | W | depleted_U | enriched_U | Hg | Pb | C		Pulsed source target material

The columns in the table are described as follows:

Name (and attributes) Name of the field. 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 two attributes: `version` (required) and `configuration` (optional) while the `thumbnail` element has one attribute: `mime_type` (optional).

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.

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 and this is *shown in parentheses* in the *Type* column. See *NXdata* for examples.

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

Description (and Occurrences) A simple text description of the field. No markup or formatting is allowed. The absence of *Occurrences* in the item description signifies that both `minOccurs` and `maxOccurs` have the default values. If the number of occurrences of an item are specified in the NXDL (through `@minOccurs` and `@maxOccurs` attributes), they will be reported in the Description column similar to the example shown above. Default values for occurrences are shown in the following table. The `NXDL element type` is either a group (such as a NeXus base class), a field (that specifies the name and type of a variable), or an attribute of a field or group. The number of times an item can appear ranges between `minOccurs` and `maxOccurs`. A default `minOccurs` of zero means the item is optional. For attributes, `maxOccurs` cannot be greater than 1.

NXDL element type	minOccurs	maxOccurs
group	0	unbounded
field	0	unbounded
attribute	0	1

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.

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 Elements and Data Types

The documentation in this section has been obtained directly from the NXDL Schema file: *nxdl.xsd*. First, the basic elements are defined in alphabetical order. Attributes to an element are indicated immediately following the element and are preceded with an "@" symbol, such as **@attribute**. Then, the common data types used within the NXDL specification are defined. Pay particular attention to the rules for *validItemName* and *validNXClassName*.

NXDL Elements

attribute

An `attribute` element can *only* be a child of a `field` or `group` element. It is used to define *attribute* elements to be used and their data types and possibly an enumeration of allowed values.

For more details, see: *attributeType*

definition

A `definition` element can *only* be used at the root level of an NXDL specification. Note: Due to the large number of attributes of the `definition` element, they have been omitted from the figure below.

For more details, see: *definition*, *definitionType*, and *definitionTypeAttr*

dimensions

The `dimensions` element describes the *shape* of an array. It is used *only* as a child of a `field` element.

For more details, see: *dimensionsType*

[1] For example *XML Copy Editor* (http://xml-copy-editor.sourceforge.net/)
[2] http://en.wikipedia.org/wiki/XML#Well-formedness_and_error-handling

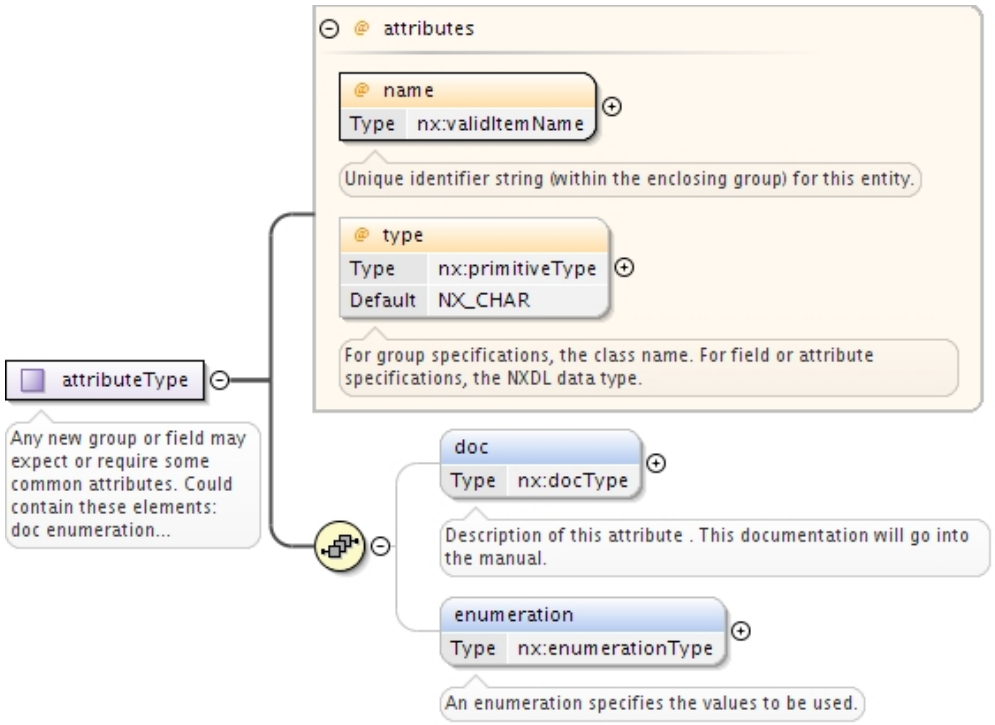


Fig. 3.1: Graphical representation of the NXDL `attribute` element

doc

A `doc` element can be a child of most NXDL elements. In most cases, the content of the `doc` element will also become part of the NeXus manual.

element {any}:

In documentation, it may be useful to use an element that is not directly specified by the NXDL language. The *any* element here says that one can use any element at all in a `doc` element and NXDL will not process it but pass it through.

For more details, see: *docType*

enumeration

An `enumeration` element can *only* be a child of a `field` or `attribute` element. It is used to restrict the available choices to a predefined list, such as to control varieties in spelling of a controversial word (such as *metre* vs. *meter*).

For more details, see: *enumerationType*

field

The `field` element provides the value of a named item. Many different attributes are available to further define the `field`. Some of the attributes are not allowed to be used together (such as `axes` and `axis`); see the documentation of each for details. It is used *only* as a child of a `group` element.

For more details, see: *fieldType*

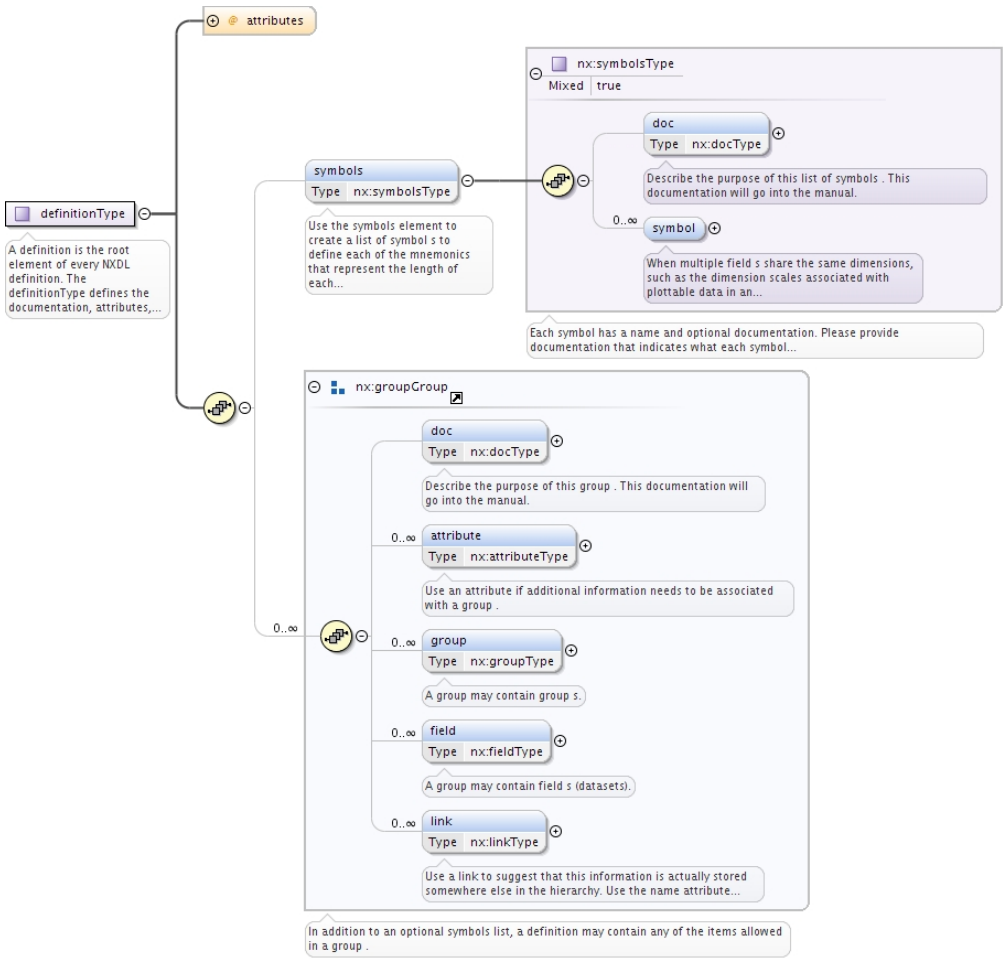


Fig. 3.2: Graphical representation of the NXDL `definition` element

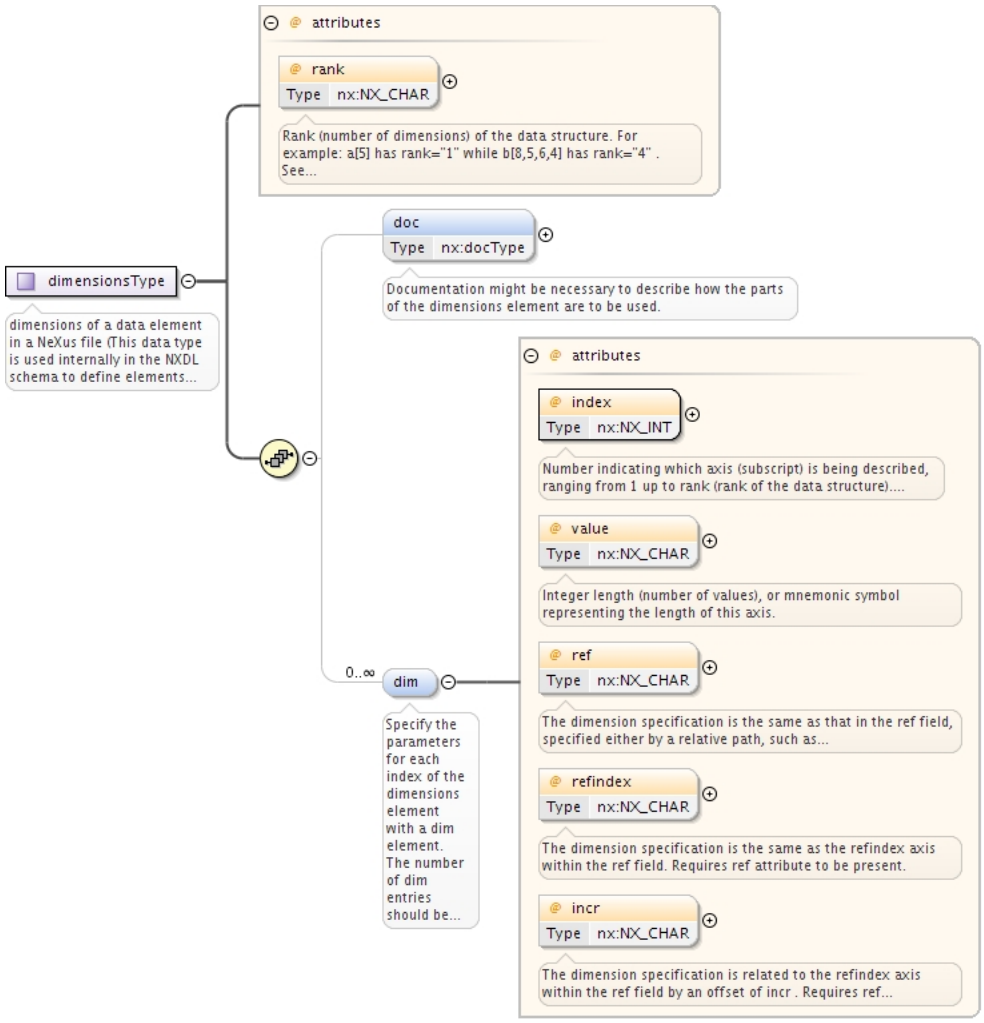


Fig. 3.3: Graphical representation of the NXDL `dimensions` element

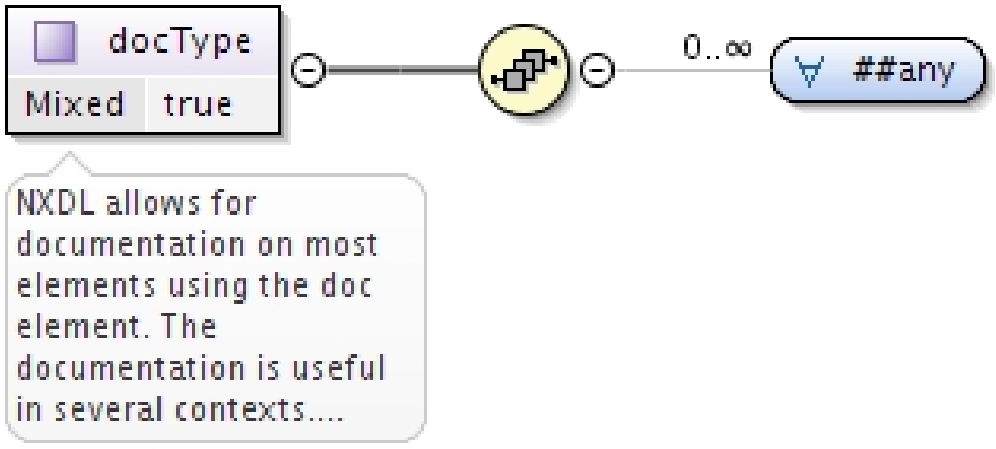


Fig. 3.4: Graphical representation of the NXDL `doc` element

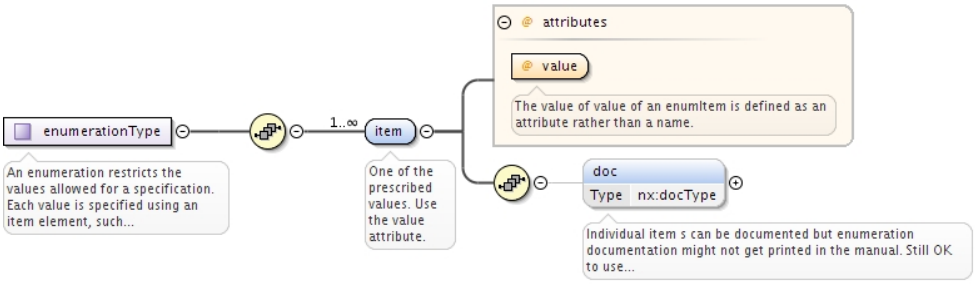


Fig. 3.5: Graphical representation of the NXDL `enumeration` element

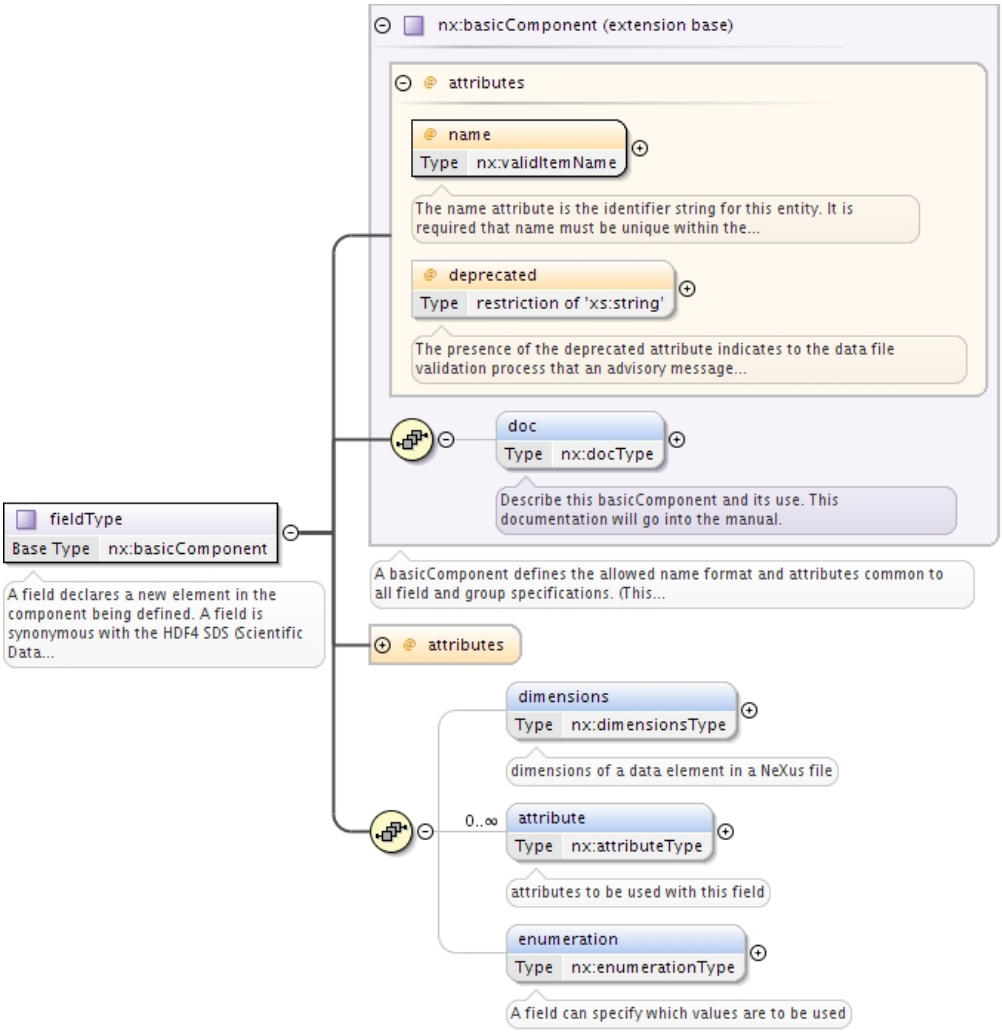


Fig. 3.6: Graphical representation of the NXDL `field` element

group

A `group` element can *only* be a child of a `definition` or `group` element. It describes a common level of organization in a NeXus data file, similar to a subdirectory in a file directory tree.

For more details, see: *groupType*

link

A `link` element can *only* be a child of a `definition`, `field`, or `group` element. It describes the path to the original source of the parent `definition`, `field`, or `group`.

For more details, see: *linkType*

symbols

A `symbols` element can *only* be a child of a `definition` element. It defines the array index symbols to be used when defining arrays as `field` elements with common dimensions and lengths.

For more details, see: *symbolsType*

NXDL Data Types (internal)

Data types that define the NXDL language are described here. These data types are defined in the XSD Schema (`nxdl.xsd`) and are used in various parts of the Schema to define common structures or to simplify a complicated entry. While the data types are not intended for use in NXDL specifications, they define structures that may be used in NXDL specifications.

attributeType

Any new group or field may expect or require some common attributes.

(This data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

Attributes of attributeType

@name

Name of the attribute (unique within the enclosing group).

@optional

Is this attribute *optional* (if **true**) or *required* (if **false**)?

@type

Type of the attribute. For `group` specifications, the class name. For `field` or `attribute` specifications, the NXDL data type.

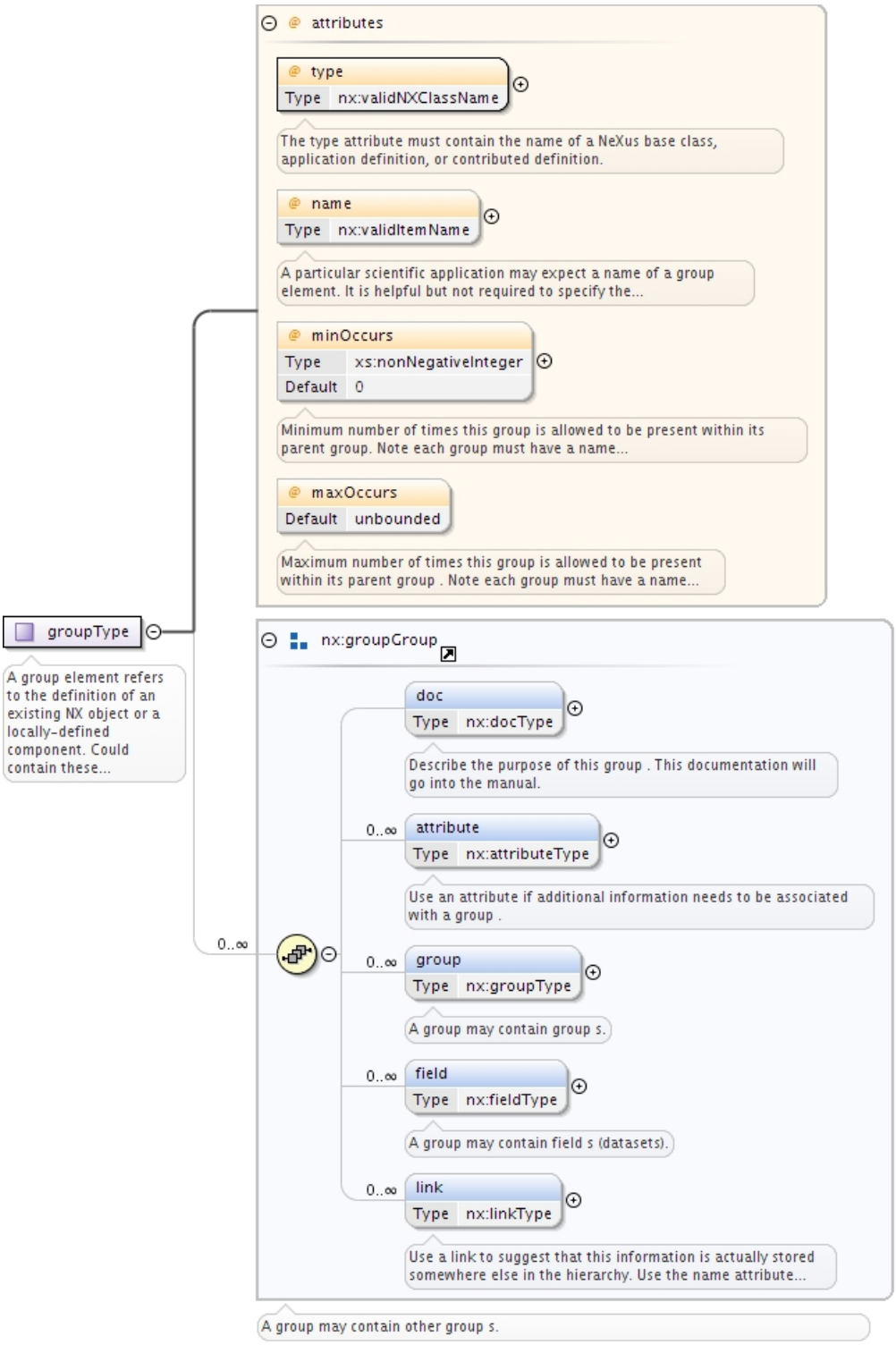


Fig. 3.7: Graphical representation of the NXDL `group` element

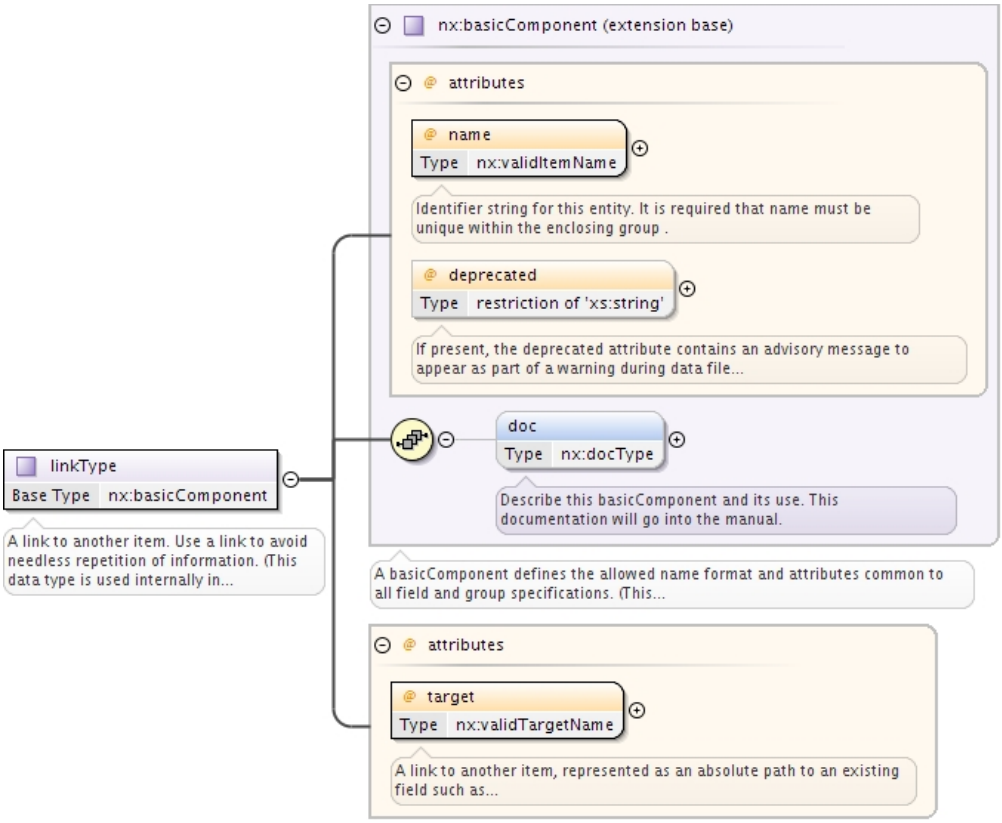


Fig. 3.8: Graphical representation of the NXDL `link` element

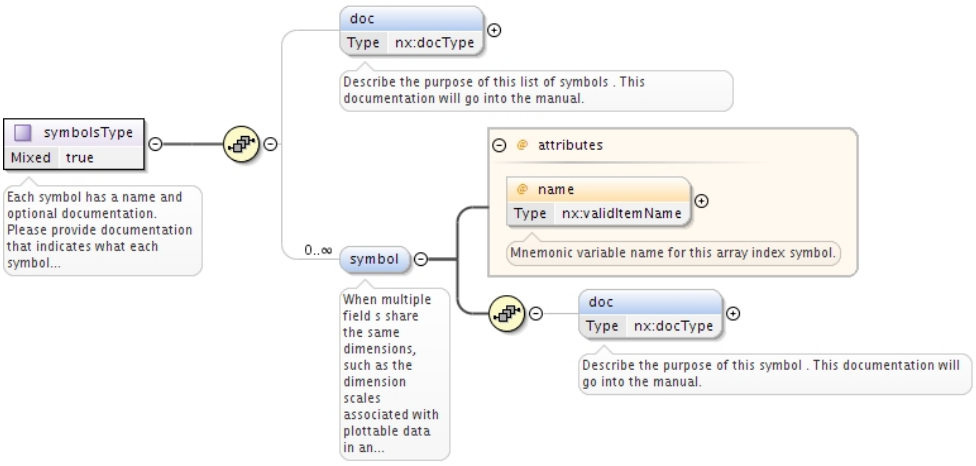


Fig. 3.9: Graphical representation of the NXDL `symbols` element

Elements of attributeType

doc

Description of this `attribute`. This documentation will go into the manual.

enumeration

An enumeration specifies the values to be used.

definition

A `definition` element is the `group` at the root of every NXDL specification. It may *only* appear at the root of an NXDL file and must only appear **once** for the NXDL to be *well-formed*.

definitionType

A `definition` is the root element of every NXDL definition. It may *only* appear at the root of an NXDL file and must only appear **once** for the NXDL to be *well-formed*.

The `definitionType` defines the documentation, attributes, fields, and groups that will be used as children of the `definition` element. Could contain these elements:

- `attribute`
- `doc`
- `field`
- `group`
- `link`

Note that a `definition` element also includes the definitions of the `basicComponent` data type. (The `definitionType` data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

Note that the first line of text in a `doc` element in a `definition` is used as a summary in the manual. Follow the pattern as shown in the base class NXDL files.

Attributes of definitionType

@category

NXDL `base` definitions define the dictionary of terms to use for these components. All terms in a `base` definition are optional. NXDL `application` definitions define what is required for a scientific interest. All terms in an `application` definition are required. NXDL `contributed` definitions may be considered either base or applications. Contributed definitions <emphasis>must</emphasis> indicate their intended use, either as a base class or as an application definition.

@extends

The `extends` attribute allows this definition to *subclass* from another NXDL, otherwise `extends="NXobject"` should be used.

@ignoreExtraAttributes

Only validate known attributes; do not not warn about unknowns. The `ignoreExtraAttributes` attribute is a flag to the process of validating NeXus data files. By setting `ignoreExtraAttributes="true"`, presence of any undefined attributes in this class will not generate warnings during validation. Normally, validation will check all the attributes against their definition in the NeXus base classes and application definitions. Any items found that do not match the definition in the NXDL will generate a warning message.

The `ignoreExtraAttributes` attribute should be used sparingly!

@ignoreExtraFields

Only validate known fields; do not not warn about unknowns. The `ignoreExtraFields` attribute is a flag to the process of validating NeXus data files. By setting `ignoreExtraFields="true"`, presence of any undefined fields in this class will not generate warnings during validation. Normally, validation will check all the fields against their definition in the NeXus base classes and application definitions. Any items found that do not match the definition in the NXDL will generate a warning message.

The `ignoreExtraFields` attribute should be used sparingly!

@ignoreExtraGroups

Only validate known groups; do not not warn about unknowns. The `ignoreExtraGroups` attribute is a flag to the process of validating NeXus data files. By setting `ignoreExtraGroups="true"`, presence of any undefined groups in this class will not generate warnings during validation. Normally, validation will check all the groups against their definition in the NeXus base classes and application definitions. Any items found that do not match the definition in the NXDL will generate a warning message.

The `ignoreExtraGroups` attribute should be used sparingly!

@name

The `name` of this NXDL file (without the file extensions). The name must be unique amongst all the NeXus base class, application, and contributed definitions. For the class to be adopted by the NIAC, the first two letters must be "`NX`" (in uppercase). Any other use must *not* begin with "`NX`" in any combination of upper or lower case.

@restricts

The `restricts` attribute is a flag to the data validation. When `restricts="1"`, any non-standard component found (and checked for validity aginst this NXDL specification) in a NeXus data file will be flagged as an error. If the `restricts` attribute is not present, any such situations will produce a warning.

@svnid

(2014-08-19: deprecated since switch to GitHub version control) The identifier string from the subversion revision control system. This reports the time stamp and the revision number of this file. (Updated automatically, unlike the `version` attribute.)

@type

Must be `type="group"`

@version

Version of *this* NXDL definition. Each NXDL specification may have a different version to facilitate software maintenance. This value is modified by the person who edits this file when this NXDL specification has changed significantly (in a way that downstream software should be aware).

Elements of definitionType

symbols

Use a `symbols` list to define each of the mnemonics that represent the length of each dimension in a vector or array.

Groups under definitionType

In addition to an optional `symbols` list, a `definition` may contain any of the items allowed in a `group`.

definitionTypeAttr

Prescribes the allowed values for `definition type` attribute. (This data type is used internally in the NXDL schema to define a data type.)

The value may be any one from this list only:

- `group`
- `definition`

dimensionsType

dimensions of a data element in a NeXus file (This data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

Attributes of dimensionsType

@rank

Rank (number of dimensions) of the data structure. For example: `a[5]` has `rank="1"` while `b[8,5,6,4]` has `rank="4"`. See http://en.wikipedia.org/wiki/Rank_(computer_programming) for more details.

Elements of dimensionsType

dim

Specify the parameters for each index of the `dimensions` element with a `dim` element. The number of `dim` entries should be equal to the `rank` of the array. For example, these terms describe a 2-D array with lengths (`nsurf`, `nwl`):

```
<dimensions rank="2">
    <dim index="1" value="nsurf"/>
    <dim index="2" value="nwl"/>
</dimensions>
```

The `value` attribute is used by NXDL and also by the NeXus data file validation tools to associate and coordinate the same array length across multiple fields in a group.

@incr

The dimension specification is related to the `refindex` axis within the `ref` field by an offset of `incr`. Requires `ref` and `refindex` attributes to be present.

@index

Number or symbol indicating which axis (subscript) is being described, ranging from 1 up to `rank` (rank of the data structure). For example, given an array `A[i,j,k]`, `index="1"` would refer to the `i` axis (subscript). (`NXdata` uses `index="0"` to indicate a situation when the specific index is not known *a priori*.)

@ref

The dimension specification is the same as that in the `ref` field, specified either by a relative path, such as `polar_angle` or `../Qvec` or absolute path, such as `/entry/path/to/follow/to/ref/field`.

@refindex

The dimension specification is the same as the `refindex` axis within the `ref` field. Requires `ref` attribute to be present.

@value

Integer length (number of values), or mnemonic symbol representing the length of this axis.

doc

Documentation might be necessary to describe how the parts of the `dimensions` element are to be used.

docType

NXDL allows for documentation on most elements using the `doc` element. The documentation is useful in several contexts. The documentation will be rendered in the manual. Documentation, is provided as tooltips by some XML editors when editing NXDL files. Simple documentation can be typed directly in the NXDL:

```
<field name="name">
        <doc>Descriptive name of sample</doc>
</field>
```

This is suitable for basic descriptions that do not need extra formatting such as a bullet-list or a table. For more advanced control, use the rules of restructured text, such as in the *NXdetector* specification. Refer to examples in the NeXus base class NXDL files such as *NXdata*.

Could contain these elements:

- *any*

(This data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

Note: For documentation of `definition` elements, the first line of text in a `doc` is used as a summary in the manual. Follow the pattern as shown in the base class NXDL files.

enumerationType

An `enumeration` restricts the values allowed for a specification. Each value is specified using an `item` element, such as: `<item value="Synchrotron X-ray Source"/>`. Could contain these elements:

- `doc`
- `item`

(This data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

```
<field name="mode">
        <doc>source operating mode</doc>
        <enumeration>
                <item value="Single Bunch"><doc>for storage rings</doc></item>
                <item value="Multi Bunch"><doc>for storage rings</doc></item>
                <!-- other sources could add to this -->
        </enumeration>
</field>
```

Elements of enumerationType

item

One of the prescribed values. Use the `value` attribute.

@value

The value of `value` of an `enumItem` is defined as an attribute rather than a name.

doc

Individual items can be documented but this documentation might not be printed in the *NeXus Reference Guide*.

fieldType

A `field` declares a new element in the component being defined. A `field` is synonymous with the HDF4 SDS (Scientific Data Set) and the HDF5 *dataset* terms. Could contain these elements:

- `attribute`
- `dimensions`
- `doc`
- `enumeration`

Note that a `field` element also includes the definitions of the `basicComponent` data type. (The `fieldType` data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

@axes

NOTE: Use of this attribute is discouraged. It is for legacy support. You should use the axes attribute on the NXdata group instead.

Presence of the `axes` attribute means this field is an ordinate.

This attribute contains a colon (or comma in legacy files) delimited list of the names of independent axes when plotting this field. Each name in this list must exist as a field in the same group. <!– perhaps even discourage use of square brackets in axes attribute? –> (Optionally, the list can be enclosed by square brackets but this is not common.) The regular expression for this rule is:

```
[A-Za-z_][\w_]*([ :][A-Za-z_][\w_]*)*
```

@axis

NOTE: Use of this attribute is discouraged. It is for legacy support. You should use the axes attribute on the NXdata group instead.

Presence of the `axis` attribute means this field is an abcissa.

The attribute value is an integer indicating this field as an axis that is part of the data set. The data set is a field with the attribute `signal=1` in the same group. The value can range from 1 up to the number of independent axes (abcissae) in the data set.

A value of `axis=1`" indicates that this field contains the data for the first independent axis. For example, the X axis in an XY data set.

A value of `axis=2` indicates that this field contains the data for the second independent axis. For example, the Y axis in a 2-D data set.

A value of `axis=3` indicates that this field contains the data for the third independent axis. For example, the Z axis in a 3-D data set.

A field with an `axis` attribute should not have a `signal` attribute.

@data_offset

The `stride` and `data_offset` attributes are used together to index the array of data items in a multi-dimensional array. They may be used as an alternative method to address a data array that is not stored in the standard NeXus method of "C" order.

The `data_offset` attribute determines the starting coordinates of the data array for each dimension.

See http://davis.lbl.gov/Manuals/HDF5-1.4.3/Tutor/phypereg.html or *4. Dataspace Selection Operations* in http://www.hdfgroup.org/HDF5/doc1.6/Dataspaces.html.

The `data_offset` attribute contains a comma-separated list of integers. (In addition to the required comma delimiter, whitespace is also allowed to improve readability.) The number of items in the list is equal to the rank of the data being stored. The value of each item is the offset in the array of the first data item of that subscript of the array.

@interpretation

This instructs the consumer of the data what the last dimensions of the data are. It allows plotting software to work out the natural way of displaying the data.

For example a single-element, energy-resolving, fluorescence detector with 512 bins should have `interpretation="spectrum"`. If the detector is scanned over a 512 x 512 spatial grid, the data reported will be of dimensions: 512 x 512 x 512. In this example, the initial plotting representation should default to data of the same dimensions of a 512 x 512 pixel `image` detector where the images where taken at 512 different pressure values.

In simple terms, the allowed values mean:

- `scaler` = 0-D data to be plotted
- `spectrum` = 1-D data to be plotted
- `image` = 2-D data to be plotted
- `vertex` = 3-D data to be plotted

@long_name

Descriptive name for this field (may include whitespace and engineering units). Often, the long_name (when defined) will be used as the axis label on a plot.

@maxOccurs

Defines the maximum number of times this element may be used. Its value is confined to zero or greater. Must be greater than or equal to the value for the "minOccurs" attribute. A value of "unbounded" is allowed.

@minOccurs

Defines the minimum number of times this element may be used. Its value is confined to zero or greater. Must be less than or equal to the value for the "maxOccurs" attribute.

@nameType

This interprets the name attribute as: * `specified` = use as specified * `any` = can be any name not already used in group

@primary

Integer indicating the priority of selection of this field for plotting (or visualization) as an axis.

Presence of the `primary` attribute means this field is an abcissa.

@signal

Presence of the `signal` attribute means this field is an ordinate.

Integer marking this field as plottable data (ordinates). The value indicates the priority of selection or interest. Some facilities only use `signal=1` while others use `signal=2` to indicate plottable data of secondary interest. Higher numbers are possible but not common and interpretation is not standard.

A field with a `signal` attribute should not have an `axis` attribute.

@stride

The `stride` and `data_offset` attributes are used together to index the array of data items in a multi-dimensional array. They may be used as an alternative method to address a data array that is not stored in the standard NeXus method of "C" order.

The `stride` list chooses array locations from the data array with each value in the `stride` list determining how many elements to move in each dimension. Setting a value in the `stride` array to 1 moves to each element in that dimension of the data array, while setting a value of 2 in a location in the `stride` array moves to every other element in that dimension of the data array. A value in the `stride` list may be positive to move forward or negative to step backward. A value of zero will not step (and is of no particular use).

See http://davis.lbl.gov/Manuals/HDF5-1.4.3/Tutor/phypereg.html or *4. Dataspace Selection Operations* in http://www.hdfgroup.org/HDF5/doc1.6/Dataspaces.html.

The `stride` attribute contains a comma-separated list of integers. (In addition to the required comma delimiter, whitespace is also allowed to improve readability.) The number of items in the list is equal to the rank of the data being stored. The value of each item is the spacing of the data items in that subscript of the array.

@type

Defines the type of the element as allowed by the NAPI (NeXus Application Programmer Interface). See elsewhere for the complete list of allowed NAPI types.

@units

String describing the engineering units. The string should be appropriate for the value and should conform to the NeXus rules for units. Conformance is not validated at this time.

attribute

attributes to be used with this field

dimensions

dimensions of a data element in a NeXus file

enumeration

A field can specify which values are to be used

groupType

A group element refers to the definition of an existing NX object or a locally-defined component. Could contain these elements:

- `attribute`
- `doc`
- `field`
- `group`
- `link`

Note that a `group` element also includes the definitions of the `basicComponent` data type. (The `groupType` data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

Attributes of groupType

@maxOccurs

Maximum number of times this `group` is allowed to be present within its parent `group`. Note each `group` must have a `name` attribute that is unique among all `group` and `field` declarations within a common parent `group`.

@minOccurs

Minimum number of times this `group` is allowed to be present within its parent group. Note each `group` must have a `name` attribute that is unique among all `group` and `field` declarations within a common parent group.

@name

A particular scientific application may expect a name of a `group` element. It is helpful but not required to specify the `name` attribute in the NXDL file. It is suggested to always specify a `name` to avoid ambiguity. It is also suggested to derive the `name` from the type, using an additional number suffix as necessary. For example, consider a data file with only one `NXentry`. The suggested default `name` would be `entry`. For a data file with two or more `NXentry` groups, the suggested names would be `entry1`, `entry2`, ... Alternatively, a scientific application such as small-angle scattering might require a different naming procedure; two different `NXaperture` groups might be given the names `beam-defining slit` and `scatter slit`.

@type

The `type` attribute *must* contain the name of a NeXus base class, application definition, or contributed definition.

linkType

A link to another item. Use a link to avoid needless repetition of information. (This data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

@target

A link to another item, represented as an absolute path to an existing field or group such as `/NXentry/NXinstrument/NXdetector/polar_angle` or `/NXentry/NXdata`. Could contain these elements:

- `doc`

Matching regular expression:

```
(/[a-zA-Z_][\w_]*(:[a-zA-Z_][\w_]*)?)+
```

symbolsType

Each `symbol` has a `name` and optional documentation. Please provide documentation that indicates what each symbol represents. For example:

```
<symbols>
    <symbol name="nsurf"><doc>number of reflecting surfaces</doc></symbol>
    <symbol name="nwl"><doc>number of wavelengths</doc></symbol>
</symbols>
```

Elements of symbolsType

doc

Describe the purpose of this list of `symbols`. This documentation will go into the manual.

symbol

When multiple `field` elements share the same dimensions, such as the dimension scales associated with plottable data in an `NXdata` group, the length of each dimension written in a NeXus data file should be something that can be tested by the data file validation process.

@name

Mnemonic variable name for this array index symbol.

doc

Describe the purpose of the parent `symbol`. This documentation will go into the manual.

basicComponent

A `basicComponent` defines the allowed name format and attributes common to all `field` and `group` specifications. (This data type is used internally in the NXDL schema to define elements and attributes to be used by users in NXDL specifications.)

Attributes of basicComponent

@name

The `name` attribute is the identifier string for this entity. It is required that `name` must be unique within the enclosing `group`. The rule (`validItemName`) is defined to only allow names that can be represented as valid variable names in most computer languages.

Elements of basicComponent

doc

Describe this `basicComponent` and its use. This documentation will go into the manual.

validItemName

Used for allowed names of elements and attributes. Need to be restricted to valid program variable names. Note: This means no "-" or "." characters can be allowed and you cannot start with a number. HDF4 had a 64 character limit on names (possibly including NULL) and NeXus enforces this via the `NX_MAXNAMELEN` variable with a **64**

character limit (which may be 63 on a practical basis if one considers a NULL terminating byte). (This data type is used internally in the NXDL schema to define a data type.)

The value may be any `xs:token` that *also* matches the regular expression:

```
[A-Za-z_][\w_]*
```

validNXClassName

Used for allowed names of NX class types (e.g. NXdetector) not the instance (e.g. bank1) which is covered by validItemName. (This data type is used internally in the NXDL schema to define a data type.)

The value may be any `nx:validItemName` that *also* matches the regular expression:

```
NX.+
```

validTargetName

This is a valid link target - currently it must be an absolute path made up of valid names with the / character delimiter. But we may want to consider allowing "`..`" (parent of directory) at some point. If the `name` attribute is helpful, then use it in the path with the syntax of *name:type* as in these examples:

```
/NXentry/NXinstrument/analyzer:NXcrystal/ef
/NXentry/NXinstrument/monochromator:NXcrystal/ei
/NX_other
```

Must also consider use of `name` attribute in resolving `link` targets. (This data type is used internally in the NXDL schema to define a data type.)

From the HDF5 documentation (http://www.hdfgroup.org/HDF5/doc/UG/UG_frame09Groups.html):

> *Note that relative path names in HDF5 do not employ the ''../'' notation, the UNIX notation indicating a parent directory, to indicate a parent group.*

Thus, if we only consider the case of `[name:]type`, the matching regular expression syntax is written: `/[a-zA-Z_][\w_]*(:[a-zA-Z_][\w_]*)?)+`. Note that HDF5 also permits relative path names, such as: `GroupA/GroupB/Dataset1` but this is not permitted in the matching regular expression and not supported in NAPI.

The value may be any `xs:token` that *also* matches the regular expression:

```
(/[a-zA-Z_][\w_]*(:[a-zA-Z_][\w_]*)?)+
```

nonNegativeUnbounded

A `nonNegativeUnbounded` allows values including all positive integers, zero, and the string `unbounded`. (This data type is used internally in the NXDL schema to define a data type.)

The `xs:string` data type The `xs:string` data type can contain characters, line feeds, carriage returns, and tab characters. See http://www.w3schools.com/Schema/schema_dtypes_string.asp for more details.

The `xs:token` data type The `xs:string` data type is derived from the `xs:string` data type.

The `xs:token` data type also contains characters, but the XML processor will remove line feeds, carriage returns, tabs, leading and trailing spaces, and multiple spaces. See http://www.w3schools.com/Schema/schema_dtypes_string.asp for more details.

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. 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.

ISO8601 ISO 8601 date and time representation (http://www.w3.org/TR/NOTE-datetime)

NX_BINARY any representation of binary data - if text, line terminator is [CR][LF]

NX_BOOLEAN true/false value (true | 1 | false | 0)

NX_CHAR any string representation

NX_DATE_TIME alias of ISO8601

NX_FLOAT any representation of a floating point number

NX_INT any representation of an integer number

NX_NUMBER any valid NeXus number representation

NX_POSINT any representation of a positive integer number (greater than zero)

NX_UINT any representation of an unsigned integer number (includes zero)

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.

NX_ANGLE example: degrees or radians or arcminutes or

NX_ANY usage: things like logs that aren't picky on units

NX_AREA example: m2 or barns

NX_CHARGE example: pC or C

NX_CROSS_SECTION example: barns

NX_CURRENT example: A

NX_DIMENSIONLESS for fields where the units cancel out, example: "" or mm/mm (NOTE: not the same as NX_UNITLESS)

NX_EMITTANCE emittance (`length * angle`) of a radiation source, example: `nm*rad`

NX_ENERGY example: J or keV

NX_FLUX example: s-1 cm-2

NX_FREQUENCY example: Hz

NX_LENGTH example: m

NX_MASS example: g

NX_MASS_DENSITY example: g cm-3

NX_MOLECULAR_WEIGHT example: g mol-1

NX_PERIOD (alias to NX_TIME) period of pulsed source, example: microseconds

NX_PER_AREA example: cm-2

NX_PER_LENGTH example: cm-1

NX_POWER example: W

NX_PRESSURE example: Pa

NX_PULSES (alias to NX_NUMBER) clock pulses

NX_SCATTERING_LENGTH_DENSITY example: cm-2

NX_SOLID_ANGLE example: sr | steradian

NX_TEMPERATURE example: K

NX_TIME example: s

NX_TIME_OF_FLIGHT (alias to NX_TIME) example: s

NX_UNITLESS for fields that don't have a unit (e.g. hkl) so that they don't inherit the wrong units (NOTE: not the same as NX_DIMENSIONLESS)

NX_VOLTAGE example: V

NX_VOLUME example: m3

NX_WAVELENGTH example: angstrom

NX_WAVENUMBER units for Q, example: angstrom-1 or nm-1

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)

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.

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.

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.

3.3.1 Base Class Definitions

A description of each NeXus base class definition is given. 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.

NXaperture A beamline aperture.

NXattenuator A device that reduces the intensity of a beam by attenuation.

NXbeam Properties of the neutron or X-ray beam at a given location.

NXbeam_stop A device that blocks the beam completely, usually to protect a detector.

NXbending_magnet A bending magnet

NXcapillary A capillary lens to focus the X-ray beam.

NXcharacterization legacy only - not intended for new use - may be removed in the future

NXcite A literature reference

NXcollection An unvalidated set of terms, such as the description of a beam line.

NXcollimator A beamline collimator.

NXcrystal A crystal monochromator or analyzer.

NXdata (**required**) *NXdata* describes the plottable data and related dimension scales.

NXdetector A detector, detector bank, or multidetector.

NXdetector_group Logical grouping of detector elements.

NXdetector_module Geometry and logical description of a detector module.

NXdisk_chopper A device blocking the beam in a temporal periodic pattern.

NXentry (**required**) *NXentry* describes the measurement.

NXenvironment Parameters for controlling external conditions

NXevent_data Time-of-flight events

NXfermi_chopper A Fermi chopper, possibly with curved slits.

NXfilter For band pass beam filters.

NXflipper A spin flipper.

NXfresnel_zone_plate A fresnel zone plate

NXgeometry legacy class - recommend to use *NXtransformations* now

NXgrating A diffraction grating, as could be used in a soft X-ray monochromator

NXguide A neutron optical element to direct the path of the beam.

NXinsertion_device An insertion device, as used in a synchrotron light source.

NXinstrument Collection of the components of the instrument or beamline.

NXlog Information recorded as a function of time.

NXmirror A beamline mirror or supermirror.

NXmoderator A neutron moderator

NXmonitor A monitor of incident beam data.

NXmonochromator A wavelength defining device.

NXnote Any additional freeform information not covered by the other base classes.

NXobject This is the base object of NeXus

NXorientation legacy class - recommend to use *NXtransformations* now

NXparameters Container for parameters, usually used in processing or analysis.

NXpinhole A simple pinhole.

NXpolarizer A spin polarizer.

NXpositioner A generic positioner such as a motor or piezo-electric transducer.

NXprocess Document an event of data processing, reconstruction, or analysis for this data.

NXroot Definition of the root NeXus group.

NXsample Any information on the sample.

NXsensor A sensor used to monitor an external condition

NXshape legacy class - (used by *NXgeometry*) - the shape and size of a component.

NXslit A simple slit.

NXsource The neutron or x-ray storage ring/facility.

NXsubentry Group of multiple application definitions for "multi-modal" (e.g. SAXS/WAXS) measurements.

NXtransformations Collection of translations and rotations to describe a geometry

NXtranslation legacy class - (used by *NXgeometry*) - general spatial location of a component.

NXuser Contact information for a user.

NXvelocity_selector A neutron velocity selector

NXxraylens An X-ray lens, typically at a synchrotron X-ray beam line.

NXaperture

Status:

base class, extends *NXobject*, version 1.0

Description:

A beamline aperture.

Symbols:

No symbol table

Groups cited: *NXgeometry*, *NXnote*

Structure:

material: *NX_CHAR*

Absorbing material of the aperture

description: *NX_CHAR*

Description of aperture

(geometry): *NXgeometry*

location and shape of aperture

(geometry): *NXgeometry*

location and shape of each blade

(note): *NXnote*

describe an additional information in a note*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXaperture.nxdl.xml

NXattenuator

Status:

base class, extends *NXobject*, version 1.0

Description:

A device that reduces the intensity of a beam by attenuation.

If uncertain whether to use *NXfilter* (band-pass filter) or *NXattenuator* (reduces beam intensity), then choose *NXattenuator*.

Symbols:

No symbol table

Groups cited: none

Structure:

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Distance from sample

type: *NX_CHAR*

Type or composition of attenuator, e.g. polythene

thickness: *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of attenuator along beam direction

scattering_cross_section: *NX_FLOAT* {units=*NX_CROSS_SECTION*}

Scattering cross section (coherent+incoherent)

absorption_cross_section: *NX_FLOAT* {units=*NX_CROSS_SECTION*}

Absorption cross section

attenuator_transmission: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

The nominal amount of the beam that gets through (transmitted intensity)/(incident intensity)

status: *NX_CHAR*

In or out or moving of the beam

Any of these values: `in` | `out` | `moving`

@time: *NX_DATE_TIME*

time stamp for this observation

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXattenuator.nxdl.xml

NXbeam

Status:

base class, extends *NXobject*, version 1.0

Description:

Properties of the neutron or X-ray beam at a given location.

It will be referenced by beamline component groups within the *NXinstrument* group or by the *NXsample* group. Note that variables such as the incident energy could be scalar values or arrays. This group is especially valuable in storing the results of instrument simulations in which it is useful to specify the beam profile, time distribution etc. at each beamline component. Otherwise, its most likely use is in the *NXsample* group in which it defines the results of the neutron scattering by the sample, e.g., energy transfer, polarizations.

Symbols:

No symbol table

Groups cited: *NXdata*

Structure:

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Distance from sample

incident_energy[i]: *NX_FLOAT* {units=*NX_ENERGY*}

Energy on entering beamline component

final_energy[i]: *NX_FLOAT* {units=*NX_ENERGY*}

Energy on leaving beamline component

energy_transfer[i]: *NX_FLOAT* {units=*NX_ENERGY*}

Energy change caused by beamline component

incident_wavelength[i]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Wavelength on entering beamline component

incident_wavelength_spread[i]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Wavelength spread FWHM on entering component

incident_beam_divergence[2, j]: *NX_FLOAT* {units=*NX_ANGLE*}

Divergence of beam entering this component

final_wavelength[i]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Wavelength on leaving beamline component

incident_polarization[2, j]: *NX_FLOAT* {units=*NX_ANY*}

Polarization vector on entering beamline component

final_polarization[2, j]: *NX_FLOAT* {units=*NX_ANY*}

Polarization vector on leaving beamline component

final_wavelength_spread[i]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Wavelength spread FWHM of beam leaving this component

final_beam_divergence[2, j]: *NX_FLOAT* {units=*NX_ANGLE*}

Divergence FWHM of beam leaving this component

flux[i]: *NX_FLOAT* {units=*NX_FLUX*}

flux incident on beam plane area

(data): *NXdata*

Distribution of beam with respect to relevant variable e.g. wavelength. This is mainly useful for simulations which need to store plottable information at each beamline component.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXbeam.nxdl.xml

NXbeam_stop

Status:

base class, extends *NXobject*, version 1.0

Description:

A device that blocks the beam completely, usually to protect a detector.

Beamstops and their positions are important for SANS and SAXS experiments.

Symbols:

No symbol table

Groups cited: *NXgeometry*

Structure:

description: *NX_CHAR*

description of beamstop

Any of these values: `circular` | `rectangular`

size: *NX_FLOAT* {units=*NX_LENGTH*}

size of beamstop

x: *NX_FLOAT* {units=*NX_LENGTH*}

x position of the beamstop in relation to the detector

y: *NX_FLOAT* {units=*NX_LENGTH*}

y position of the beamstop in relation to the detector

distance_to_detector: *NX_FLOAT* {units=*NX_LENGTH*}

distance of the beamstop to the detector

status: *NX_CHAR*

Any of these values: `in` | `out`

(geometry): *NXgeometry*

engineering shape, orientation and position of the beam stop.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXbeam_stop.nxdl.xml

NXbending_magnet

Status:

base class, extends *NXobject*, version 1.0

Description:

A bending magnet

Symbols:

No symbol table

Groups cited: *NXdata*, *NXgeometry*

Structure:

critical_energy: *NX_FLOAT* {units=*NX_ENERGY*}

bending_radius: *NX_FLOAT* {units=*NX_LENGTH*}

magnetic_field: *NX_FLOAT* {units=*NX_CURRENT*}

strength of magnetic field of dipole magnets

accepted_photon_beam_divergence: *NX_FLOAT* {units=*NX_LENGTH*}

An array of four numbers giving X+, X-, Y+ and Y- half divergence

source_distance_x: *NX_FLOAT* {units=*NX_LENGTH*}

Distance of source point from particle beam waist in X (horizontal) direction.

source_distance_y: *NX_FLOAT* {units=*NX_LENGTH*}

Distance of source point from particle beam waist in Y (vertical) direction.

divergence_x_plus: *NX_FLOAT* {units=*NX_ANGLE*}

Accepted photon beam divergence in X+ (horizontal outboard) direction. Note that divergence_x_plus+divergence_x_minus is the total horizontal beam divergence.

divergence_x_minus: *NX_FLOAT* {units=*NX_ANGLE*}

Accepted photon beam divergence in X- (horizontal inboard) direction. Note that divergence_x_plus+divergence_x_minus is the total horizontal beam divergence.

divergence_y_plus: *NX_FLOAT* {units=*NX_ANGLE*}

Accepted photon beam divergence in Y+ (vertical upward) direction. Note that divergence_y_plus+divergence_y_minus is the total vertical beam divergence.

divergence_y_minus: *NX_FLOAT* {units=*NX_ANGLE*}

Accepted photon beam divergence in Y- (vertical downward) direction. Note that divergence_y_plus+divergence_y_minus is the total vertical beam divergence.

spectrum: *NXdata*

bending magnet spectrum

(**geometry**): *NXgeometry*

“Engineering” position of bending magnet

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXbending_magnet.nxdl.xml

NXcapillary

Status:

base class, extends *NXobject*, version 1.0

Description:

A capillary lens to focus the X-ray beam.

Based on information provided by Gerd Wellenreuther (DESY).

Symbols:

No symbol table

Groups cited: *NXdata*

Structure:

type: *NX_CHAR*

Type of the capillary

Any of these values:

- `single_bounce`
- `polycapillary`
- `conical_capillary`

manufacturer: *NX_CHAR*

The manufacturer of the capillary. This is actually important as it may have an impact on performance.

maximum_incident_angle: *NX_FLOAT* {units=*NX_ANGLE*}

accepting_aperture: *NX_FLOAT* {units=*NX_ANGLE*}

working_distance: *NX_FLOAT* {units=*NX_LENGTH*}

focal_size: *NX_FLOAT*

The focal size in FWHM

gain: *NXdata*

The gain of the capillary as a function of energy

transmission: *NXdata*

The transmission of the capillary as a function of energy

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXcapillary.nxdl.xml

NXcharacterization

Status:

base class, extends *NXobject*, version 1.0

Description:

legacy only - not intended for new use - may be removed in the future

Note: This base class may be removed in future releases of NXDL. If you have a use for this base class, please provide a description of your intended use to the NIAC (nexus-committee@nexusformat.org).

Symbols:

No symbol table

Groups cited: none

Structure:

@source: *NX_CHAR*

If missing, the source file is the current file

@location: *NX_CHAR*

@mime_type: *NX_CHAR*

If missing, the source file is NAPI readable

definition: *NX_CHAR*

@version: *NX_CHAR*

@URL: *NX_CHAR*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXcharacterization.nxdl.xml

NXcite

Status:

base class, extends *NXobject*, version 1.0

Description:

A literature reference

Definition to include references for example for detectors, manuals, instruments, acquisition or analysis software used.

The idea would be to include this in the relevant NeXus object: *NXdetector* for detectors, *NXinstrument* for instruments, etc.

Symbols:

No symbol table

Groups cited: none

Structure:

description: *NX_CHAR*

This should describe the reason for including this reference. For example: The dataset in this group was normalised using the method which is described in detail in this reference.

url: *NX_CHAR*

URL referencing the document or data.

doi: *NX_CHAR*

DOI referencing the document or data.

endnote: *NX_CHAR*

Bibliographic reference data in EndNote format.

bibtex: *NX_CHAR*

Bibliographic reference data in BibTeX format.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXcite.nxdl.xml

NXcollection

Status:

base class, extends *NXobject*, version 1.0

Description:

An unvalidated set of terms, such as the description of a beam line.

Use *NXcollection* to gather together any set of terms. The original suggestion is to use this as a container class for the description of a beamline.

For NeXus validation, *NXcollection* will always generate a warning since it is always an optional group. Anything (groups, fields, or attributes) placed in an *NXcollection* group will not be validated.

Symbols:

No symbol table

Groups cited: none

Structure:

beamline: *NX_CHAR*

name of the beamline for this collection

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXcollection.nxdl.xml

NXcollimator

Status:

base class, extends *NXobject*, version 1.0

Description:

A beamline collimator.

Symbols:

No symbol table

Groups cited: *NXgeometry*, *NXlog*

Structure:

type: *NX_CHAR*

Any of these values: `Soller` | `radial` | `oscillating` | `honeycomb`

soller_angle: *NX_FLOAT* {units=*NX_ANGLE*}

Angular divergence of Soller collimator

divergence_x: *NX_FLOAT* {units=*NX_ANGLE*}

divergence of collimator in local x direction

divergence_y: *NX_FLOAT* {units=*NX_ANGLE*}

divergence of collimator in local y direction

frequency: *NX_FLOAT* {units=*NX_FREQUENCY*}

Frequency of oscillating collimator

blade_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

blade thickness

blade_spacing: *NX_FLOAT* {units=*NX_LENGTH*}

blade spacing

absorbing_material: *NX_CHAR*

name of absorbing material

transmitting_material: *NX_CHAR*

name of transmitting material

(geometry): *NXgeometry*

position, shape and size

frequency_log: *NXlog*

Log of frequency

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXcollimator.nxdl.xml

NXcrystal

Status:

base class, extends *NXobject*, version 1.0

Description:

A crystal monochromator or analyzer.

Permits double bent monochromator comprised of multiple segments with anisotropic Gaussian mosaic.

If curvatures are set to zero or are absent, array is considered to be flat.

Scattering vector is perpendicular to surface. Crystal is oriented parallel to beam incident on crystal before rotation, and lies in vertical plane.

Symbols:

These symbols will be used below to coordinate dimensions with the same lengths.

n_comp: number of different unit cells to be described

i: number of wavelengths

Groups cited: *NXdata*, *NXgeometry*, *NXlog*, *NXshape*

Structure:

usage: *NX_CHAR*

How this crystal is used. Choices are in the list.

Any of these values:

- `Bragg`: reflection geometry
- `Laue`: The chemical formula specified using CIF conventions. Abbreviated version of CIF standard: * Only recognized element symbols may be used. * Each element symbol is followed by a 'count' number. A count of '1' may be omitted. * A space or parenthesis must separate each cluster of (element symbol + count). * Where a group of elements is enclosed in parentheses, the multiplier for the group must follow the closing parentheses. That is, all element and group multipliers are assumed to be printed as subscripted numbers. * Unless the elements are ordered in a manner that corresponds to their chemical structure, the order of the elements within any group or moiety depends on whether or not carbon is present. * If carbon is present, the order should be: C, then H, then the other elements in alphabetical order of their symbol. If carbon is not present, the elements are listed purely in alphabetic order of their symbol. This is the *Hill* system used by Chemical Abstracts. See, for example: http://www.iucr.org/__data/iucr/cif/standard/cifstd15.html, http://www.cas.org/training/stneasytips/subinforformula1.html, or http://www.indiana.edu/~cheminfo/courses/471cnfs.html.

type: *NX_CHAR*

Type or material of monochromating substance. Chemical formula can be specified separately. Use the "reflection" field to indicate the (hkl) orientation. Use the "d_spacing" field to record the lattice plane spacing.

This field was changed (2010-11-17) from an enumeration to a string since common usage showed a wider variety of use than a simple list. These are the items in the list at the time of the change: PG (Highly Oriented Pyrolytic Graphite) | Ge | Si | Cu | Fe3Si | CoFe | Cu2MnAl (Heusler) | Multilayer | Diamond.

chemical_formula: *NX_CHAR*

The chemical formula specified using CIF conventions. Abbreviated version of CIF standard:

- Only recognized element symbols may be used.
- Each element symbol is followed by a 'count' number. A count of '1' may be omitted.
- A space or parenthesis must separate each cluster of (element symbol + count).
- Where a group of elements is enclosed in parentheses, the multiplier for the group must follow the closing parentheses. That is, all element and group multipliers are assumed to be printed as subscripted numbers.
- Unless the elements are ordered in a manner that corresponds to their chemical structure, the order of the elements within any group or moiety depends on whether or not carbon is present.
- If carbon is present, the order should be: C, then H, then the other elements in alphabetical order of their symbol. If carbon is not present, the elements are listed purely in alphabetic order of their symbol.
- This is the *Hill* system used by Chemical Abstracts.

order_no: *NX_INT*

A number which describes if this is the first, second,.. n^{th} crystal in a multi crystal monochromator

cut_angle: *NX_FLOAT* {units=*NX_ANGLE*}

Cut angle of reflecting Bragg plane and plane of crystal surface

space_group: *NX_CHAR*

Space group of crystal structure

unit_cell[n_comp, 6]: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell parameters (lengths and angles)

unit_cell_a: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell lattice parameter: length of side a

unit_cell_b: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell lattice parameter: length of side b

unit_cell_c: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell lattice parameter: length of side c

unit_cell_alpha: *NX_FLOAT* {units=*NX_ANGLE*}

Unit cell lattice parameter: angle alpha

unit_cell_beta: *NX_FLOAT* {units=*NX_ANGLE*}

Unit cell lattice parameter: angle beta

unit_cell_gamma: *NX_FLOAT* {units=*NX_ANGLE*}

Unit cell lattice parameter: angle gamma

unit_cell_volume: *NX_FLOAT* {units=*NX_VOLUME*}

Volume of the unit cell

orientation_matrix[3, 3]: *NX_FLOAT*

Orientation matrix of single crystal sample using Busing-Levy convention: W. R. Busing and H. A. Levy (1967). Acta Cryst. 22, 457-464

wavelength[i]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Optimum diffracted wavelength

d_spacing: *NX_FLOAT* {units=*NX_LENGTH*}

spacing between crystal planes of the reflection

scattering_vector: *NX_FLOAT* {units=*NX_WAVENUMBER*}

Scattering vector, Q, of nominal reflection

reflection[3]: *NX_INT* {units=*NX_UNITLESS*}

Miller indices (hkl) values of nominal reflection

thickness: *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of the crystal. (Required for Laue orientations - see “usage” field)

density: *NX_NUMBER* {units=*NX_MASS_DENSITY*}

mass density of the crystal

segment_width: *NX_FLOAT* {units=*NX_LENGTH*}

Horizontal width of individual segment

segment_height: *NX_FLOAT* {units=*NX_LENGTH*}

Vertical height of individual segment

segment_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of individual segment

segment_gap: *NX_FLOAT* {units=*NX_LENGTH*}

Typical gap between adjacent segments

segment_columns: *NX_FLOAT* {units=*NX_LENGTH*}

number of segment columns in horizontal direction

segment_rows: *NX_FLOAT* {units=*NX_LENGTH*}

number of segment rows in vertical direction

mosaic_horizontal: *NX_FLOAT* {units=*NX_ANGLE*}

horizontal mosaic Full Width Half Maximum

mosaic_vertical: *NX_FLOAT* {units=*NX_ANGLE*}

vertical mosaic Full Width Half Maximum

curvature_horizontal: *NX_FLOAT* {units=*NX_ANGLE*}

Horizontal curvature of focusing crystal

curvature_vertical: *NX_FLOAT* {units=*NX_ANGLE*}

Vertical curvature of focusing crystal

is_cylindrical: *NX_BOOLEAN*

Is this crystal bent cylindrically?

cylindrical_orientation_angle: *NX_NUMBER* {units=*NX_ANGLE*}

If cylindrical: cylinder orientation angle

polar_angle[i]: *NX_FLOAT* {units=*NX_ANGLE*}

Polar (scattering) angle at which crystal assembly is positioned. Note: some instrument geometries call this term 2theta.

azimuthal_angle[i]: *NX_FLOAT* {units=*NX_ANGLE*}

Azimuthal angle at which crystal assembly is positioned

bragg_angle[i]: *NX_FLOAT* {units=*NX_ANGLE*}

Bragg angle of nominal reflection

temperature: *NX_FLOAT* {units=*NX_TEMPERATURE*}

average/nominal crystal temperature

temperature_coefficient: *NX_FLOAT* {units=*NX_ANY*}

how lattice parameter changes with temperature

(geometry): *NXgeometry*

Position of crystal

temperature_log: *NXlog*

log file of crystal temperature

reflectivity: *NXdata*

crystal reflectivity versus wavelength

transmission: *NXdata*

crystal transmission versus wavelength

shape: *NXshape*

A NXshape group describing the shape of the crystal arrangement

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXcrystal.nxdl.xml

NXdata

Status:

base class, extends *NXobject*, version 1.0

Description:

(**required**) *NXdata* describes the plottable data and related dimension scales.

It is mandatory that there is at least one *NXdata* group in each *NXentry* group. Note that the `variable` and `data` can be defined with different names. The `signal` and `axes` attributes of the `data` group define which items are plottable data and which are *dimension scales*, respectively.

NXdata is used to implement one of the basic motivations in NeXus, to provide a default plot for the data of this *NXentry*. The actual data might be stored in another group and (hard) linked to the *NXdata* group.

- Each *NXdata* group will define only one data set containing plottable data, dimension scales, and possibly associated standard deviations. Other data sets may be present in the group.
- The plottable data may be of arbitrary rank up to a maximum of `NX_MAXRANK=32`.
- The plottable data will be named as the value of the group `signal` attribute, such as:

```
data:NXdata
  @signal = "counts"
  @axes = "mr"
  @mr_indices = 0
  counts: float[100]  --> the default dependent data
  mr: float[100]  --> the default independent data
```

  The field named in the `signal` attribute **must** exist, either directly as a dataset or defined through a link.

- The group `axes` attribute will name the *dimension scale* associated with the plottable data.

If available, the standard deviations of the data are to be stored in a data set of the same rank and dimensions, with the name `errors`.

- For each data dimension, there should be a one-dimensional array of the same length.
- These one-dimensional arrays are the *dimension scales* of the data, *i.e.* the values of the independent variables at which the data is measured, such as scattering angle or energy transfer.

The preferred method to associate each data dimension with its respective dimension scale is to specify the field name of each dimension scale in the group `axes` attribute as a string list. Here is an example for a 2-D data set *data* plotted against *time*, and *pressure*. (An additional *temperature* data set is provided and could be selected as an alternate for the *pressure* axis.):

```
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]
```

Old methods to identify the plottable data

There are two older methods of associating each data dimension to its respective dimension scale. Both are now out of date and should not be used when writing new data files. However, client software should expect to see data files written with any of these methods.

- One method uses the `axes` attribute to specify the names of each *dimension scale*.
- The oldest method uses the `axis` attribute on each *dimension scale* to identify with an integer the axis whose value is the number of the dimension.

Symbols:

These symbols will be used below to coordinate datasets with the same shape.

dataRank: rank of the `data` field

n: length of the `variable` field

nx: length of the `x` field

ny: length of the `y` field

nz: length of the `z` field

Groups cited: none

Structure:

@signal: *NX_CHAR*

Declares which dataset is the default. The value is the name of the dataset to be plotted. A field of this name *must* exist (either as dataset or as a link to a dataset).

It is recommended (as of NIAC2014) to use this attribute rather than adding a signal attribute to the dataset. See http://wiki.nexusformat.org/2014_How_to_find_default_data for a summary of the discussion.

@axes: *NX_CHAR*

String array that defines the independent data fields used in the default plot for all of the dimensions of the *signal* field (the *signal* field is the field in this group that is named by the `signal` attribute of this group). 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. Such as:

```
@I_axes="time", ".", "."
```

Since there are three items in the list, the the *signal* field must must be a three-dimensional array (rank=3). The first dimension is described by the values of a one-dimensional array named `time` while the other two dimensions have no fields to be used as dimension scales.

See examples provided on the NeXus wiki: http://wiki.nexusformat.org/2014_axes_and_uncertainties

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

@AXISNAME_indices: *NX_CHAR*

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 that defines the indices of the *signal* field (that field will be a multidimensional array) which need to be used in the *AXISNAME* dataset 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".

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

---

variable[n]: *NX_NUMBER*

Dimension scale defining an axis of the data. Client is responsible for defining the dimensions of the data. The name of this field may be changed to fit the circumstances. Standard NeXus client tools will use the attributes to determine how to use this field.

@long_name: *NX_CHAR*

Axis label

@distribution: *NX_BOOLEAN*

`0|false`: single value, `1|true`: multiple values

@first_good: *NX_INT*

Index of first good value

@last_good: *NX_INT*

Index of last good value

@axis: *NX_POSINT*

Index (positive integer) identifying this specific set of numbers.

N.B. The `axis` attribute is the old way of designating a link. Do not use the `axes` attribute with the `axis` attribute. The `axes` *group* attribute is now preferred.

variable_errors[n]: *NX_NUMBER*

Errors (uncertainties) associated with axis `variable`. Client is responsible for defining the dimensions of the data. The name of this field may be changed to fit the circumstances but is matched with the *variable* field with `_errors` appended.

data[n]: *NX_NUMBER*

This field contains the data values to be used as the NeXus *plottable data*. Client is responsible for defining the dimensions of the data. The name of this field may be changed to fit the circumstances. Standard NeXus client tools will use the attributes to determine how to use this field.

@signal: *NX_POSINT*

Plottable (independent) axis, indicate index number. Only one field in a *NXdata* group may have the `signal=1` attribute. Do not use the `signal` attribute with the `axis` attribute.

@axes: *NX_CHAR*

Defines the names of the dimension scales (independent axes) for this data set as a colon-delimited array. NOTE: The `axes` attribute is the preferred method of designating a link. Do not use the `axes` attribute with the `axis` attribute.

@uncertainties: *NX_CHAR*

Specify the name (or names) of the uncertainties (errors) of the dependent axes as plottable data. NOTE: The `uncertainties` attribute uses the same syntax as the `axes` attribute, a string or an array of strings for multiple uncertainties.

Examples:

```
@I_uncertainties="Idev"

@Q_uncertainties="dQw", "dQl"
```

@long_name: *NX_CHAR*

data label

errors[n]: *NX_NUMBER*

Standard deviations of data values - the data array is identified by the group attribute `signal`. The `errors` array must have the same dimensions as `data`. Client is responsible for defining the dimensions of the data.

scaling_factor: *NX_FLOAT*

The elements in data are usually float values really. For efficiency reasons these are usually stored as integers after scaling with a scale factor. This value is the scale factor. It is required to get the actual physical value, when necessary.

offset: *NX_FLOAT*

An optional offset to apply to the values in data.

x[nx]: *NX_FLOAT* {units=*NX_ANY*}

This is an array holding the values to use for the x-axis of data. The units must be appropriate for the measurement.

y[ny]: *NX_FLOAT* {units=*NX_ANY*}

This is an array holding the values to use for the y-axis of data. The units must be appropriate for the measurement.

z[nz]: *NX_FLOAT* {units=*NX_ANY*}

This is an array holding the values to use for the z-axis of data. The units must be appropriate for the measurement.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXdata.nxdl.xml

NXdetector

Status:

base class, extends *NXobject*, version 1.1

Description:

A detector, detector bank, or multidetector.

Symbols:

These symbols will be used below to coordinate datasets with the same shape.

np: number of scan points (only present in scanning measurements)

i: number of detector pixels in the first (X, slowest) direction

j: number of detector pixels in the second (Y, faster) direction

k: number of detector pixels in the third (Z, if necessary, fastest) direction

tof: number of bins in the time-of-flight histogram

Groups cited: *NXcharacterization*, *NXcollection*, *NXdata*, *NXdetector_module*, *NXgeometry*, *NXnote*

Structure:

time_of_flight[tof+1]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

Total time of flight

@axis: *NX_POSINT*

Obligatory value: `3`

@primary: *NX_POSINT*

Obligatory value: `1`

@long_name: *NX_CHAR*

Total time of flight

raw_time_of_flight[tof+1]: *NX_INT* {units=*NX_PULSES*}

In DAQ clock pulses

@frequency: *NX_NUMBER*

Clock frequency in Hz

detector_number[i, j]: *NX_INT*

Identifier for detector

data[np, i, j, tof]: *NX_NUMBER* {units=*NX_ANY*}

Data values from the detector.

@long_name: *NX_CHAR*

Title of measurement

@check_sum: *NX_INT*

Integral of data as check of data integrity

data_error[np, i, j, tof]: *NX_NUMBER* {units=*NX_ANY*}

The best estimate of the uncertainty in the data value. Where possible, this should be the standard deviation, which has the same units as the data.

x_pixel_offset[i, j]: *NX_FLOAT* {units=*NX_LENGTH*}

Offset from the detector center in x-direction. Can be multidimensional when needed.

@axis: *NX_POSINT*

Obligatory value: `1`

@primary: *NX_POSINT*

Obligatory value: `1`

@long_name: *NX_CHAR*

x-axis offset from detector center

y_pixel_offset[i, j]: *NX_FLOAT* {units=*NX_LENGTH*}

Offset from the detector center in the y-direction. Can be multidimensional when different values are required for each pixel.

@axis: *NX_POSINT*

Obligatory value: `2`

@primary: *NX_POSINT*

Obligatory value: `1`

@long_name: *NX_CHAR*

y-axis offset from detector center

distance[np, i, j]: *NX_FLOAT* {units=*NX_LENGTH*}

This is the distance to the previous component in the instrument; most often the sample. The usage depends on the nature of the detector: Most often it is the distance of the detector assembly. But there are irregular detectors. In this case the distance must be specified for each detector pixel.

polar_angle[np, i, j]: *NX_FLOAT* {units=*NX_ANGLE*}

This is the polar angle of the detector towards the previous component in the instrument; most often the sample. The usage depends on the nature of the detector. Most often it is the polar_angle of the detector assembly. But there are irregular detectors. In this case, the polar_angle must be specified for each detector pixel.

azimuthal_angle[np, i, j]: *NX_FLOAT* {units=*NX_ANGLE*}

This is the azimuthal angle angle of the detector towards the previous component in the instrument; most often the sample. The usage depends on the nature of the detector. Most often it is the azimuthal_angle of the detector assembly. But there are irregular detectors. In this case, the azimuthal_angle must be specified for each detector pixel.

description: *NX_CHAR*

name/manufacturer/model/etc. information

local_name: *NX_CHAR*

Local name for the detector

solid_angle[i, j]: *NX_FLOAT* {units=*NX_SOLID_ANGLE*}

Solid angle subtended by the detector at the sample

x_pixel_size[i, j]: *NX_FLOAT* {units=*NX_LENGTH*}

Size of each detector pixel. If it is scalar all pixels are the same size.

y_pixel_size[i, j]: *NX_FLOAT* {units=*NX_LENGTH*}

Size of each detector pixel. If it is scalar all pixels are the same size

dead_time[np, i, j]: *NX_FLOAT* {units=*NX_TIME*}

Detector dead time

gas_pressure[i, j]: *NX_FLOAT* {units=*NX_PRESSURE*}

Detector gas pressure

detection_gas_path: *NX_FLOAT* {units=*NX_LENGTH*}

maximum drift space dimension

crate[i, j]: *NX_INT*

Crate number of detector

@local_name: *NX_CHAR*

Equivalent local term

slot[i, j]: *NX_INT*

Slot number of detector

@local_name: *NX_CHAR*

Equivalent local term

input[i, j]: *NX_INT*

Input number of detector

@local_name: *NX_CHAR*

Equivalent local term

type: *NX_CHAR*

Description of type such as He3 gas cylinder, He3 PSD, scintillator, fission chamber, proportion counter, ion chamber, ccd, pixel, image plate, CMOS, ...

calibration_date: *NX_DATE_TIME*

date of last calibration (geometry and/or efficiency) measurements

layout: *NX_CHAR*

How the detector is represented

Any of these values: `point` | `linear` | `area`

count_time[np]: *NX_NUMBER* {units=*NX_TIME*}

Elapsed actual counting time

sequence_number[nBrightFrames]: *NX_CHAR*

In order to properly sort the order of the images taken in (for example) a tomography experiment, a sequence number is stored with each image.

beam_center_x: *NX_FLOAT* {units=*NX_LENGTH*}

This is the x position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

beam_center_y: *NX_FLOAT* {units=*NX_LENGTH*}

This is the y position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

frame_start_number: *NX_INT*

This is the start number of the first frame of a scan. In PX one often scans a couple of frames on a give sample, then does something else, then returns to the same sample and scans some more frames. Each time with a new data file. This number helps concatenating such measurements.

diameter: *NX_FLOAT* {units=*NX_LENGTH*}

The diameter of a cylindrical detector

acquisition_mode: *NX_CHAR*

The acquisition mode of the detector.

Any of these values:

- `gated`
- `triggered`
- `summed`
- `event`
- `histogrammed`
- `decimated`

angular_calibration_applied: *NX_BOOLEAN*

True when the angular calibration has been applied in the electronics, false otherwise.

angular_calibration[i, j]: *NX_FLOAT*

Angular calibration data.

flatfield_applied: *NX_BOOLEAN*

True when the flat field correction has been applied in the electronics, false otherwise.

flatfield[i, j]: *NX_FLOAT*

Flat field correction data.

flatfield_error[i, j]: *NX_FLOAT*

Errors of the flat field correction data.

pixel_mask_applied: *NX_BOOLEAN*

True when the pixel mask correction has been applied in the electronics, false otherwise.

pixel_mask[i, j]: *NX_INT*

The 32-bit pixel mask for the detector. Contains a bit field for each pixel to signal dead, blind or high or otherwise unwanted or undesirable pixels. They have the following meaning:

- bit 0: gap (pixel with no sensor)
- bit 1: dead
- bit 2: under responding
- bit 3: over responding
- bit 4: noisy
- bit 5: -undefined-
- bit 6: pixel is part of a cluster of problematic pixels (bit set in addition to others)
- bit 7: -undefined-
- bit 8: user defined mask (e.g. around beamstop)
- bits 9-30: -undefined-
- bit 31: virtual pixel (corner pixel with interpolated value)

The normal data analysis software would not take pixels into account when a bit in (mask & 0x00FF) is set. Tag bit in the upper two bytes would indicate special pixel properties that normally would not be a sole reason to reject the intensity value (unless lower bits are also set).

countrate_correction__applied: *NX_BOOLEAN*

True when a count-rate correction has already been applied in the electronics, false otherwise.

bit_depth_readout: *NX_INT*

How many bits the electronics reads per pixel. With CCD's and single photon counting detectors, this must not align with traditional integer sizes. This can be 4, 8, 12, 14, 16, ...

detector_readout_time: *NX_FLOAT* {units=*NX_TIME*}

Time it takes to read the detector (typically milliseconds). This is important to know for time resolved experiments.

trigger_delay_time: *NX_FLOAT* {units=*NX_TIME*}

Time it takes to start exposure after a trigger signal has been received. This is the reaction time of the detector firmware after receiving the trigger signal to when the detector starts to acquire the exposure, including any user set delay.. This is important to know for time resolved experiments.

trigger_delay_time_set: *NX_FLOAT* {units=*NX_TIME*}

User-specified trigger delay.

trigger_internal_delay_time: *NX_FLOAT* {units=*NX_TIME*}

Time it takes to start exposure after a trigger signal has been received. This is the reaction time of the detector hardware after receiving the trigger signal to when the detector starts to acquire the exposure. It forms the lower boundary of the trigger_delay_time when the user does not request an additional delay.

trigger_dead_time: *NX_FLOAT* {units=*NX_TIME*}

Time during which no new trigger signal can be accepted. Typically this is the trigger_delay_time + exposure_time + readout_time. This is important to know for time resolved experiments.

frame_time[NP]: *NX_FLOAT* {units=*NX_TIME*}

This is time for each frame. This is exposure_time + readout time.

gain_setting: *NX_CHAR*

The gain setting of the detector. This influences background etc.

Any of these values: `high` | `standard` | `fast` | `auto`

saturation_value: *NX_INT*

The value at which the detector goes into saturation. Especially common to CCD detectors, the data is known to be invalid above this value.

number_of_cycles: *NX_INT*

CCD images are sometimes constructed by summing together multiple short exposures in the electronics. This reduces background etc. This is the number of short exposures used to sum images for an image.

sensor_material: *NX_CHAR*

At times, radiation is not directly sensed by the detector. Rather, the detector might sense the output from some converter like a scintillator. This is the name of this converter material.

sensor_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

At times, radiation is not directly sensed by the detector. Rather, the detector might sense the output from some converter like a scintillator. This is the thickness of this converter material.

threshold_energy: *NX_FLOAT* {units=*NX_ENERGY*}

Single photon counter detectors can be adjusted for a certain energy range in which they work optimally. This is the energy setting for this.

(geometry): *NXgeometry*

Position and orientation of detector

efficiency: *NXdata*

Spectral efficiency of detector with respect to e.g. wavelength

@signal: *NX_CHAR*

Obligatory value: `efficiency`

@axes: *NX_CHAR*

Any of these values: `.` | `.` | `.` | `.` | `.` | `.` | `.` | `.` | `wavelength`

@wavelength_indices: *NX_CHAR*

Obligatory value: `0`

efficiency[i, j, k]: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

efficiency of the detector

wavelength[i, j, k]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

This field can be two things:

1. For a pixel detector it provides the nominal wavelength for which the detector has been calibrated.
2. For other detectors this field has to be seen together with the efficiency field above. For some detectors, the efficiency is wavelength dependent. Thus this field provides the wavelength axis for the efficiency field. In this use case, the efficiency and wavelength arrays must have the same dimensionality.

start_time[np]: *NX_FLOAT* {units=*NX_TIME*}

start time for each frame, with the `start` attribute as absolute reference

@start: *NX_DATE_TIME*

stop_time[np]: *NX_FLOAT* {units=*NX_TIME*}

stop time for each frame, with the `start` attribute as absolute reference

@start: *NX_DATE_TIME*

real_time[i, j, k]: *NX_NUMBER* {units=*NX_TIME*}

real-time of the exposure (use this if exposure time varies for each array element, otherwise use `count_time` field)

calibration_method: *NXnote*

summary of conversion of array data to pixels (e.g. polynomial approximations) and location of details of the calibrations

data_file: *NXnote*

(characterization): *NXcharacterization*

DEPRECATED: use *NXcollection* instead

see https://github.com/nexusformat/definitions/issues/177

(collection): *NXcollection*

Use this group to provide other data related to this NXdetector group.

(detector_module): *NXdetector_module*

For use in special cases where the data in NXdetector is represented in several parts, each with a separate geometry.

Use one or more instances of the NXdetector_module group to declare regions of interest or some other subdivision of a detector.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXdetector.nxdl.xml

NXdetector_group

Status:

base class, extends *NXobject*, version 1.0

Description:

Logical grouping of detector elements.

This class is used to allow a logical grouping of detector elements (e.g. which tube, bank or group of banks) to be recorded in the file. As well as allowing you to e.g just select the "left" or "east" detectors, it may also be useful for determining which elements belong to the same PSD tube and hence have e.g. the same dead time.

For example, if we had "bank1" composed of "tube1", "tube2" and "tube3" then group_names would be the string "bank1, bank1/tube1, bank1/tube2,bank1/tube3" group_index would be {1,2,3,4} group_parent would be {-1,1,1,1}

The mapping array is interpreted as group 1 is a top level group containing groups 2, 3 and 4

A `group_index` array in `NXdetector` gives the base group for a detector element.

Symbols:

No symbol table

Groups cited: none

Structure:

group_names: *NX_CHAR*

Comma separated list of name

group_index[i]: *NX_INT*

Unique ID for group. A group_index array in `NXdetector` gives the base group for a detector element.

group_parent[ref(group_index)]: *NX_INT*

Index of group parent in the hierarchy: -1 means no parent (i.e. a top level) group

group_type[ref(group_index)]: *NX_INT*

Code number for group type, e.g. bank=1, tube=2 etc.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXdetector_group.nxdl.xml

NXdetector_module

Status:

base class, extends *NXobject*, version 1.0

Description:

Geometry and logical description of a detector module.

Many detectors consist of multiple smaller modules. Sometimes it is important to know the exact position of such modules. This is the purpose of this group. It is a child group to NXdetector.

Symbols:

No symbol table

Groups cited: none

Structure:

data_origin: *NX_INT*

A two value field which gives the index of the start of the modules data in the main area detector image in the underlying NXdetector module.

data_size: *NX_INT*

Two values for the size of the module in pixels in each direction.

module_offset: *NX_NUMBER* {units=*NX_LENGTH*}

Offset of the module in regards to the origin of the detector in an arbitrary direction.

@transformation_type: *NX_CHAR*

Obligatory value: `translation`

@vector: *NX_NUMBER*

Three values that define the axis for this transformation

@offset: *NX_NUMBER*

A fixed offset applied before the transformation (three vector components).

@offset_units: *NX_CHAR*

Units of the offset.

@depends_on: *NX_CHAR*

Points to the path of the next element in the geometry chain.

fast_pixel_direction: *NX_NUMBER* {units=*NX_LENGTH*}

Values along the direction of fastest varying pixel direction.The direction itself is given through the vector attribute

@transformation_type: *NX_CHAR*

Obligatory value: `translation`

@vector: *NX_NUMBER*

Three values that define the axis for this transformation

@offset: *NX_NUMBER*

A fixed offset applied before the transformation (three vector components).

@offset_units: *NX_CHAR*

Units of the offset.

@depends_on: *NX_CHAR*

Points to the path of the next element in the geometry chain.

slow_pixel_direction: *NX_NUMBER* {units=*NX_LENGTH*}

Values along the direction of slow varying pixel direction. The direction itself is given through the vector attribute

@transformation_type: *NX_CHAR*

Obligatory value: `translation`

@vector: *NX_NUMBER*

Three values that define the axis for this transformation

@offset: *NX_NUMBER*

A fixed offset applied before the transformation (three vector components).

@offset_units: *NX_CHAR*

Units of the offset.

@depends_on: *NX_CHAR*

Points to the path of the next element in the geometry chain.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXdetector_module.nxdl.xml

NXdisk_chopper

Status:

base class, extends *NXobject*, version 1.0

Description:

A device blocking the beam in a temporal periodic pattern.

TODO: need documentation

Symbols:

No symbol table

Groups cited: *NXgeometry*

Structure:

type: *NX_CHAR*

Type of the disk-chopper: only one from the enumerated list (match text exactly)

Any of these values:

- `Chopper type single`
- `contra_rotating_pair`
- `synchro_pair`

rotation_speed: *NX_FLOAT* {units=*NX_FREQUENCY*}

chopper rotation speed

slits: *NX_INT*

Number of slits

slit_angle: *NX_FLOAT* {units=*NX_ANGLE*}

angular opening

pair_separation: *NX_FLOAT* {units=*NX_LENGTH*}

disc spacing in direction of beam

radius: *NX_FLOAT* {units=*NX_LENGTH*}

radius to centre of slit

slit_height: *NX_FLOAT* {units=*NX_LENGTH*}

total slit height

phase: *NX_FLOAT* {units=*NX_ANGLE*}

chopper phase angle

ratio: *NX_INT*

pulse reduction factor of this chopper in relation to other choppers/fastest pulse in the instrument

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Effective distance to the origin

wavelength_range[2]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

low and high values of wavelength range transmitted

(geometry): *NXgeometry*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXdisk_chopper.nxdl.xml

NXentry

Status:

base class, extends *NXobject*, version 1.0

Description:

(**required**) *NXentry* describes the measurement.

The top-level NeXus group which contains all the data and associated information that comprise a single measurement. It is mandatory that there is at least one group of this type in the NeXus file.

Symbols:

No symbol table

Groups cited: *NXcharacterization*, *NXcollection*, *NXdata*, *NXinstrument*, *NXmonitor*, *NXnote*, *NXparameters*, *NXprocess*, *NXsample*, *NXsubentry*, *NXuser*

Structure:

@default: *NX_CHAR*

Declares which *NXdata* (or *NXsubentry*) group contains the data to be shown by default. It is needed to resolve ambiguity when more than one *NXdata* group exists. The value is the name of the default *NXdata* group.

It is recommended (as of NIAC2014) to use this attribute to help define the path to the default dataset to be plotted. See http://wiki.nexusformat.org/2014_How_to_find_default_data for a summary of the discussion.

@IDF_Version: *NX_CHAR*

ISIS Muon IDF_Version

title: *NX_CHAR*

Extended title for entry

experiment_identifier: *NX_CHAR*

Unique identifier for the experiment, defined by the facility, possibly linked to the proposals

experiment_description: *NX_CHAR*

Brief summary of the experiment, including key objectives.

collection_identifier: *NX_CHAR*

User or Data Acquisition defined group of NeXus files or NXentry

collection_description: *NX_CHAR*

Brief summary of the collection, including grouping criteria.

entry_identifier: *NX_CHAR*

unique identifier for the measurement, defined by the facility.

features: *NX_CHAR*

Reserved for furture use by NIAC.

See https://github.com/nexusformat/definitions/issues/382

definition: *NX_CHAR*

(alternate use: see same field in *NXsubentry* for preferred)

Official NeXus NXDL schema to which this entry conforms.

This field is provided so that *NXentry* can be the overlay position in a NeXus data file for an application definition and its set of groups, fields, and attributes.

It is advised to use *NXsubentry*, instead, as the overlay position.

@version: *NX_CHAR*

NXDL version number

@URL: *NX_CHAR*

URL of NXDL file

definition_local: *NX_CHAR*

DEPRECATED: see same field in *NXsubentry* for preferred use

Local NXDL schema extended from the entry specified in the `definition` field. This contains any locally-defined, additional fields in the entry.

@version: *NX_CHAR*

NXDL version number

@URL: *NX_CHAR*

URL of NXDL file

start_time: *NX_DATE_TIME*

Starting time of measurement

end_time: *NX_DATE_TIME*

Ending time of measurement

duration: *NX_INT* {units=*NX_TIME*}

Duration of measurement

collection_time: *NX_FLOAT* {units=*NX_TIME*}

Time transpired actually collecting data i.e. taking out time when collection was suspended due to e.g. temperature out of range

run_cycle: *NX_CHAR*

Such as “2007-3”. Some user facilities organize their beam time into run cycles.

program_name: *NX_CHAR*

Name of program used to generate this file

@version: *NX_CHAR*

Program version number

@configuration: *NX_CHAR*

configuration of the program

revision: *NX_CHAR*

Revision id of the file due to re-calibration, reprocessing, new analysis, new instrument definition format, ...

@comment: *NX_CHAR*

pre_sample_flightpath: *NX_FLOAT* {units=*NX_LENGTH*}

This is the flightpath before the sample position. This can be determined by a chopper, by the moderator or the source itself. In other words: it the distance to the component which gives the T0 signal to the detector electronics. If another component in the NXinstrument hierarchy provides this information, this should be a link.

(data): *NXdata*

The required data group

experiment_documentation: *NXnote*

Description of the full experiment (document in pdf, latex, ...)

notes: *NXnote*

Notes describing entry

thumbnail: *NXnote*

A small image that is representative of the entry. An example of this is a 640x480 jpeg image automatically produced by a low resolution plot of the NXdata.

@type: *NX_CHAR*

The mime type should be an `image/*`

Obligatory value: `image/*`

(characterization): *NXcharacterization*

(user): *NXuser*

(sample): *NXsample*

(instrument): *NXinstrument*

(collection): *NXcollection*

(monitor): *NXmonitor*

(parameters): *NXparameters*

(process): *NXprocess*

(subentry): *NXsubentry*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXentry.nxdl.xml

NXenvironment

Status:

base class, extends *NXobject*, version 1.0

Description:

Parameters for controlling external conditions

Symbols:

No symbol table

Groups cited: *NXgeometry*, *NXnote*, *NXsensor*

Structure:

name: *NX_CHAR*

Apparatus identification code/model number; e.g. OC100 011

short_name: *NX_CHAR*

Alternative short name, perhaps for dashboard display like a present Seblock name

type: *NX_CHAR*

Type of apparatus. This could be the SE codes in scheduling database; e.g. OC/100

description: *NX_CHAR*

Description of the apparatus; e.g. 100mm bore orange cryostat with Roots pump

program: *NX_CHAR*

Program controlling the apparatus; e.g. LabView VI name

position: *NXgeometry*

The position and orientation of the apparatus

(note): *NXnote*

Additional information, LabView logs, digital photographs, etc

(sensor): *NXsensor*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXenvironment.nxdl.xml

NXevent_data

Status:

base class, extends *NXobject*, version 1.0

Description:

Time-of-flight events

Symbols:

No symbol table

Groups cited: none

Structure:

time_of_flight[i]: *NX_INT* {units=*NX_TIME_OF_FLIGHT*}

A list of time of flight for each event as it comes in. This list is for all pulses with information to attach to a particular pulse located in events_per_pulse.

pixel_number[i]: *NX_INT* {units=*NX_DIMENSIONLESS*}

There will be extra information in the NXdetector to convert pixel_number to detector_number. This list is for all pulses with information to attach to a particular pulse located in events_per_pulse.

pulse_time[j]: *NX_INT* {units=*NX_TIME*}

The time that each pulse started with respect to the offset

@offset: *NX_DATE_TIME*

ISO8601

events_per_pulse[j]: *NX_INT* {units=*NX_DIMENSIONLESS*}

This connects the index "i" to the index "j". The jth element is the number of events in "i" that occurred during the jth pulse.

pulse_height[i, k]: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

If voltages from the ends of the detector are read out this is where they go. This list is for all events with information to attach to a particular pulse height. The information to attach to a particular pulse is located in events_per_pulse.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXevent_data.nxdl.xml

NXfermi_chopper

Status:

base class, extends *NXobject*, version 1.0

Description:

A Fermi chopper, possibly with curved slits.

Symbols:

No symbol table

Groups cited: *NXgeometry*

Structure:

type: *NX_CHAR*

Fermi chopper type

rotation_speed: *NX_FLOAT* {units=*NX_FREQUENCY*}

chopper rotation speed

radius: *NX_FLOAT* {units=*NX_LENGTH*}

radius of chopper

slit: *NX_FLOAT* {units=*NX_LENGTH*}

width of an individual slit

r_slit: *NX_FLOAT* {units=*NX_LENGTH*}

radius of curvature of slits

number: *NX_INT* {units=*NX_UNITLESS*}

number of slits

height: *NX_FLOAT* {units=*NX_LENGTH*}

input beam height

width: *NX_FLOAT* {units=*NX_LENGTH*}

input beam width

distance: *NX_FLOAT* {units=*NX_LENGTH*}

distance

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Wavelength transmitted by chopper

energy: *NX_FLOAT* {units=*NX_ENERGY*}

energy selected

absorbing_material: *NX_CHAR*

absorbing material

transmitting_material: *NX_CHAR*

transmitting material

(geometry): *NXgeometry*

geometry of the fermi chopper

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXfermi_chopper.nxdl.xml

NXfilter

Status:

base class, extends *NXobject*, version 1.0

Description:

For band pass beam filters.

If uncertain whether to use *NXfilter* (band-pass filter) or *NXattenuator* (reduces beam intensity), then use *NXattenuator*.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXgeometry*, *NXlog*, *NXsensor*

Structure:

description: *NX_CHAR*

Composition of the filter. Chemical formula can be specified separately.

This field was changed (2010-11-17) from an enumeration to a string since common usage showed a wider variety of use than a simple list. These are the items in the list at the time of the change: Beryllium | Pyrolytic Graphite | Graphite | Sapphire | Silicon | Supermirror.

status: *NX_CHAR*

position with respect to in or out of the beam (choice of only “in” or “out”)

Any of these values:

- `in`: in the beam
- `out`: out of the beam

temperature: *NX_FLOAT* {units=*NX_TEMPERATURE*}

average/nominal filter temperature

thickness: *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of the filter

density: *NX_NUMBER* {units=*NX_MASS_DENSITY*}

mass density of the filter

chemical_formula: *NX_CHAR*

The chemical formula specified using CIF conventions. Abbreviated version of CIF standard:

- Only recognized element symbols may be used.
- Each element symbol is followed by a ‘count’ number. A count of ‘1’ may be omitted.
- A space or parenthesis must separate each cluster of (element symbol + count).
- Where a group of elements is enclosed in parentheses, the multiplier for the group must follow the closing parentheses. That is, all element and group multipliers are assumed to be printed as subscripted numbers.
- Unless the elements are ordered in a manner that corresponds to their chemical structure, the order of the elements within any group or moiety depends on whether or not carbon is present.
- If carbon is present, the order should be:
  - C, then H, then the other elements in alphabetical order of their symbol.
  - If carbon is not present, the elements are listed purely in alphabetic order of their symbol.
- This is the *Hill* system used by Chemical Abstracts.

unit_cell_a: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell lattice parameter: length of side a

unit_cell_b: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell lattice parameter: length of side b

unit_cell_c: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell lattice parameter: length of side c

unit_cell_alpha: *NX_FLOAT* {units=*NX_ANGLE*}

Unit cell lattice parameter: angle alpha

unit_cell_beta: *NX_FLOAT* {units=*NX_ANGLE*}

Unit cell lattice parameter: angle beta

unit_cell_gamma: *NX_FLOAT* {units=*NX_ANGLE*}

Unit cell lattice parameter: angle gamma

unit_cell_volume[n_comp]: *NX_FLOAT* {units=*NX_VOLUME*}

Unit cell

orientation_matrix[n_comp, 3, 3]: *NX_FLOAT*

Orientation matrix of single crystal filter using Busing-Levy convention: W. R. Busing and H. A. Levy (1967). Acta Cryst. 22, 457-464

m_value: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

m value of supermirror filter

substrate_material: *NX_CHAR*

substrate material of supermirror filter

substrate_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

substrate thickness of supermirror filter

coating_material: *NX_CHAR*

coating material of supermirror filter

substrate_roughness: *NX_FLOAT* {units=*NX_LENGTH*}

substrate roughness (RMS) of supermirror filter

coating_roughness[nsurf]: *NX_FLOAT* {units=*NX_LENGTH*}

coating roughness (RMS) of supermirror filter

(geometry): *NXgeometry*

Geometry of the filter

transmission: *NXdata*

Wavelength transmission profile of filter

temperature_log: *NXlog*

Linked temperature_log for the filter

sensor_type: *NXsensor*

Sensor(s)used to monitor the filter temperature

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXfilter.nxdl.xml

NXflipper

Status:

base class, extends *NXobject*, version 1.0

Description:

A spin flipper.

Symbols:

No symbol table

Groups cited: none

Structure:

type: *NX_CHAR*

Any of these values: `coil` | `current-sheet`

flip_turns: *NX_FLOAT* {units=*NX_PER_LENGTH*}

Linear density of turns (such as number of turns/cm) in flipping field coils

comp_turns: *NX_FLOAT* {units=*NX_PER_LENGTH*}

Linear density of turns (such as number of turns/cm) in compensating field coils

guide_turns: *NX_FLOAT* {units=*NX_PER_LENGTH*}

Linear density of turns (such as number of turns/cm) in guide field coils

flip_current: *NX_FLOAT* {units=*NX_CURRENT*}

Flipping field coil current in "on" state"

comp_current: *NX_FLOAT* {units=*NX_CURRENT*}

Compensating field coil current in "on" state"

guide_current: *NX_FLOAT* {units=*NX_CURRENT*}

Guide field coil current in "on" state"

thickness: *NX_FLOAT* {units=*NX_LENGTH*}

thickness along path of neutron travel

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXflipper.nxdl.xml

NXfresnel_zone_plate

Status:

base class, extends *NXobject*, version 1.0

Description:

A fresnel zone plate

Symbols:

No symbol table

Groups cited: *NXtransformations*

Structure:

focus_parameters[]: *NX_FLOAT*

list of polynomial coefficients describing the focal length of the zone plate, in increasing powers of photon energy, that describes the focal length of the zone plate (in microns) at an X-ray photon energy (in electron volts).

outer_diameter: *NX_FLOAT* {units=*NX_LENGTH*}

outermost_zone_width: *NX_FLOAT* {units=*NX_LENGTH*}

central_stop_diameter: *NX_FLOAT* {units=*NX_LENGTH*}

fabrication: *NX_CHAR*

how the zone plate was manufactured

Any of these values: `etched` | `plated` | `zone doubled` | `other`

zone_height: *NX_FLOAT* {units=*NX_LENGTH*}

zone_material: *NX_CHAR*

Material of the zones themselves

zone_support_material: *NX_CHAR*

Material present between the zones. This is usually only present for the "zone doubled" fabrication process

central_stop_material: *NX_CHAR*

central_stop_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

mask_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

mask_material: *NX_CHAR*

If no mask is present, set mask_thickness to 0 and omit the mask_material field

support_membrane_material: *NX_CHAR*

support_membrane_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

(transformations): *NXtransformations*

"Engineering" position of the fresnel zone plate

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXfresnel_zone_plate.nxdl.xml

NXgeometry

Status:

base class, extends *NXobject*, version 1.0

DEPRECATED: as decided at 2014 NIAC meeting, convert to use *NXtransformations*

Description:

legacy class - recommend to use *NXtransformations* now

It is recommended that instances of *NXgeometry* be converted to use *NXtransformations*.

This is the description for a general position of a component. It is recommended to name an instance of *NXgeometry* as "geometry" to aid in the use of the definition in simulation codes such as McStas. Also, in HDF, linked items must share the same name. However, it might not be possible or practical in all situations.

Symbols:

No symbol table

Groups cited: *NXorientation*, *NXshape*, *NXtranslation*

Structure:

description: *NX_CHAR*

Optional description/label. Probably only present if we are an additional reference point for components rather than the location of a real component.

component_index: *NX_INT*

Position of the component along the beam path. The sample is at 0, components upstream have negative component_index, components downstream have positive component_index.

(shape): *NXshape*

shape/size information of component

(translation): *NXtranslation*

translation of component

(orientation): *NXorientation*

orientation of component

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXgeometry.nxdl.xml

NXgrating

Status:

base class, extends *NXobject*, version 1.0

Description:

A diffraction grating, as could be used in a soft X-ray monochromator

Symbols:

No symbol table

Groups cited: *NXdata*, *NXshape*, *NXtransformations*

Structure:

angles[2]: *NX_FLOAT* {units=*NX_ANGLE*}

Blaze or trapezoidal angles, with the angle of the upstream facing edge listed first. Blazed gratings can be identified by the low value of the first-listed angle.

period[]: *NX_FLOAT* {units=*NX_LENGTH*}

List of polynomial coefficients describing the spatial separation of lines/grooves as a function of position along the grating, in increasing powers of position. Gratings which do not have variable line spacing will only have a single coefficient (constant).

duty_cycle: *NX_FLOAT* {units=*NX_UNITLESS*}

depth: *NX_FLOAT* {units=*NX_LENGTH*}

diffraction_order: *NX_INT* {units=*NX_UNITLESS*}

deflection_angle: *NX_FLOAT* {units=*NX_ANGLE*}

Angle between the incident beam and the utilised outgoing beam.

interior_atmosphere: *NX_CHAR*

Any of these values: `vacuum|helium|argon`

substrate_material: *NX_CHAR*

substrate_density: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

substrate_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

coating_material: *NX_CHAR*

substrate_roughness: *NX_FLOAT* {units=*NX_LENGTH*}

coating_roughness: *NX_FLOAT* {units=*NX_LENGTH*}

layer_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

An array describing the thickness of each layer

shape: *NXshape*

A NXshape group describing the shape of the mirror

figure_data: *NXdata*

Numerical description of the surface figure of the mirror.

(transformations): *NXtransformations*

"Engineering" position of the grating

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXgrating.nxdl.xml

NXguide

Status:

base class, extends *NXobject*, version 1.0

Description:

A neutron optical element to direct the path of the beam.

NXguide is used by neutron instruments to describe a guide consists of several mirrors building a shape through which neutrons can be guided or directed. The simplest such form is box shaped although elliptical guides are gaining in popularity. The individual parts of a guide usually have common characteristics but there are cases where they are different. For example, a neutron guide might consist of 2 or 4 coated walls or a supermirror bender with multiple, coated vanes.

To describe polarizing supermirrors such as used in neutron reflection, it may be necessary to revise this definition of *NXguide* to include *NXpolarizer* and/or *NXmirror*.

When even greater complexity exists in the definition of what constitutes a *guide*, it has been suggested that *NXguide* be redefined as a *NXcollection* of *NXmirror* each having their own *NXgeometry* describing their location(s).

For the more general case when describing mirrors, consider using *NXmirror*.

NOTE: The NeXus International Advisory Committee welcomes comments for revision and improvement of this definition of *NXguide*.

Symbols:

nsurf: number of reflecting surfaces

nwl: number of wavelengths

Groups cited: *NXdata*, *NXgeometry*

Structure:

description: *NX_CHAR*

A description of this particular instance of `NXguide`.

incident_angle: *NX_FLOAT* {units=*NX_ANGLE*}

TODO: documentation needed

bend_angle_x: *NX_FLOAT* {units=*NX_ANGLE*}

TODO: documentation needed

bend_angle_y: *NX_FLOAT* {units=*NX_ANGLE*}

TODO: documentation needed

interior_atmosphere: *NX_CHAR*

Any of these values: `vacuum|helium|argon`

external_material: *NX_CHAR*

external material outside substrate

m_value[nsurf]: *NX_FLOAT*

The `m` value for a supermirror, which defines the supermirror regime in multiples of the critical angle of Nickel.

substrate_material[nsurf]: *NX_FLOAT*

TODO: documentation needed

substrate_thickness[nsurf]: *NX_FLOAT* {units=*NX_LENGTH*}

TODO: documentation needed

coating_material[nsurf]: *NX_FLOAT*

TODO: documentation needed

substrate_roughness[nsurf]: *NX_FLOAT* {units=*NX_LENGTH*}

TODO: documentation needed

coating_roughness[nsurf]: *NX_FLOAT* {units=*NX_LENGTH*}

TODO: documentation needed

number_sections: *NX_INT* {units=*NX_UNITLESS*}

number of substrate sections (also called `nsurf` as an index in the `NXguide` specification)

(geometry): *NXgeometry*

TODO: Explain what this NXgeometry group means. What is intended here?

reflectivity: *NXdata*

Reflectivity as function of reflecting surface and wavelength

@signal: *NX_CHAR*

Obligatory value: `data`

@axes: *NX_CHAR*

Obligatory value: `surface wavelength`

@surface_indices: *NX_CHAR*

Obligatory value: `0`

@wavelength_indices: *NX_CHAR*

Obligatory value: `1`

data[nsurf, nwl]: *NX_NUMBER*

reflectivity of each surface as a function of wavelength

surface[nsurf]: *NX_NUMBER* {units=*NX_ANY*}

List of surfaces. Probably best to use index numbers but the specification is very loose.

wavelength[nwl]: *NX_NUMBER* {units=*NX_WAVELENGTH*}

wavelengths at which reflectivity was measured

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXguide.nxdl.xml

NXinsertion_device

Status:

base class, extends *NXobject*, version 1.0

Description:

An insertion device, as used in a synchrotron light source.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXgeometry*

Structure:

type: *NX_CHAR*

Any of these values: `undulator|wiggler`

gap: *NX_FLOAT* {units=*NX_LENGTH*}

separation between opposing pairs of magnetic poles

taper: *NX_FLOAT* {units=*NX_ANGLE*}

angular of gap difference between upstream and downstream ends of the insertion device

phase: *NX_FLOAT* {units=*NX_ANGLE*}

poles: *NX_INT* {units=*NX_UNITLESS*}

number of poles

magnetic_wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

k: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

beam displacement parameter

length: *NX_FLOAT* {units=*NX_LENGTH*}

length of insertion device

power: *NX_FLOAT* {units=*NX_POWER*}

total power delivered by insertion device

energy: *NX_FLOAT* {units=*NX_ENERGY*}

energy of peak intensity in output spectrum

bandwidth: *NX_FLOAT* {units=*NX_ENERGY*}

bandwidth of peak energy

harmonic: *NX_INT* {units=*NX_UNITLESS*}

harmonic number of peak

spectrum: *NXdata*

spectrum of insertion device

(geometry): *NXgeometry*

"Engineering" position of insertion device

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXinsertion_device.nxdl.xml

NXinstrument

Status:

base class, extends *NXobject*, version 1.0

Description:

Collection of the components of the instrument or beamline.

Template of instrument descriptions comprising various beamline components. Each component will also be a NeXus group defined by its distance from the sample. Negative distances represent beamline components that are before the sample while positive distances represent components that are after the sample. This device allows the unique identification of beamline components in a way that is valid for both reactor and pulsed instrumentation.

Symbols:

No symbol table

Groups cited: *NXaperture*, *NXattenuator*, *NXbeam_stop*, *NXbeam*, *NXbending_magnet*, *NXcapillary*, *NXcollection*, *NXcollimator*, *NXcrystal*, *NXdetector_group*, *NXdetector*, *NXdisk_chopper*, *NXevent_data*, *NXfermi_chopper*, *NXfilter*, *NXflipper*, *NXguide*, *NXinsertion_device*, *NXmirror*, *NXmoderator*, *NXmonochromator*, *NXpolarizer*, *NXpositioner*, *NXsource*, *NXvelocity_selector*, *NXxraylens*

Structure:

name: *NX_CHAR*

Name of instrument

@short_name: *NX_CHAR*

short name for instrument, perhaps the acronym

(aperture): *NXaperture*

(attenuator): *NXattenuator*

(beam): *NXbeam*

(beam_stop): *NXbeam_stop*

(bending_magnet): *NXbending_magnet*

(collimator): *NXcollimator*

(collection): *NXcollection*

(capillary): *NXcapillary*

(crystal): *NXcrystal*

(detector): *NXdetector*

(detector_group): *NXdetector_group*

(disk_chopper): *NXdisk_chopper*

(event_data): *NXevent_data*

(fermi_chopper): *NXfermi_chopper*

(filter): *NXfilter*

(flipper): *NXflipper*

(guide): *NXguide*

(insertion_device): *NXinsertion_device*

(mirror): *NXmirror*

(moderator): *NXmoderator*

(monochromator): *NXmonochromator*

(polarizer): *NXpolarizer*

(positioner): *NXpositioner*

(source): *NXsource*

(velocity_selector): *NXvelocity_selector*

(xraylens): *NXxraylens*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXinstrument.nxdl.xml

NXlog

Status:

base class, extends *NXobject*, version 1.0

Description:

Information recorded as a function of time.

Description of information that is recorded against time, such as information monitored during the run. It contains the logged values and the times at which they were measured as elapsed time since a starting time recorded in ISO8601 format. This method of storing logged data helps to distinguish instances in which a variable is a dimension scale of the data, in which case it is stored in an *NXdata* group, and instances in which it is logged during the run, when it should be stored in an *NXlog* group. Note: When using multiple *NXlog* groups, it is suggested to place them inside a *NXcollection* group. In such cases, when *NXlog* is used in another class, *NXcollection*/*NXlog* is then constructed.

Symbols:

No symbol table

Groups cited: none

Structure:

time: *NX_FLOAT* {units=*NX_TIME*}

Time of logged entry. The times are relative to the "start" attribute and in the units specified in the "units" attribute.

@start: *NX_DATE_TIME*

value: *NX_NUMBER* {units=*NX_ANY*}

Array of logged value, such as temperature

raw_value: *NX_NUMBER* {units=*NX_ANY*}

Array of raw information, such as thermocouple voltage

description: *NX_CHAR*

Description of logged value

average_value: *NX_FLOAT* {units=*NX_ANY*}

average_value_error: *NX_FLOAT* {units=*NX_ANY*}

estimated uncertainty (often used: standard deviation) of average_value

minimum_value: *NX_FLOAT* {units=*NX_ANY*}

maximum_value: *NX_FLOAT* {units=*NX_ANY*}

duration: *NX_FLOAT* {units=*NX_ANY*}

Total time log was taken

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXlog.nxdl.xml

NXmirror

Status:

base class, extends *NXobject*, version 1.0

Description:

A beamline mirror or supermirror.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXgeometry*, *NXshape*

Structure:

type: *NX_CHAR*

Any of these values:

- `single`: mirror with a single material as a reflecting surface
- `multi`: mirror with stacked, multiple layers as a reflecting surface

description: *NX_CHAR*

description of this mirror

incident_angle: *NX_FLOAT* {units=*NX_ANGLE*}

bend_angle_x: *NX_FLOAT* {units=*NX_ANGLE*}

bend_angle_y: *NX_FLOAT* {units=*NX_ANGLE*}

interior_atmosphere: *NX_CHAR*

Any of these values: `vacuum|helium|argon`

external_material: *NX_CHAR*

external material outside substrate

m_value: *NX_FLOAT* {units=*NX_UNITLESS*}

The m value for a supermirror, which defines the supermirror regime in multiples of the critical angle of Nickel.

substrate_material: *NX_CHAR*

substrate_density: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

substrate_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

coating_material: *NX_CHAR*

substrate_roughness: *NX_FLOAT* {units=*NX_LENGTH*}

coating_roughness: *NX_FLOAT* {units=*NX_LENGTH*}

even_layer_material: *NX_CHAR*

even_layer_density: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

odd_layer_material: *NX_CHAR*

odd_layer_density: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

layer_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

An array describing the thickness of each layer

(geometry): *NXgeometry*

reflectivity: *NXdata*

Reflectivity as function of wavelength

shape: *NXshape*

A NXshape group describing the shape of the mirror

figure_data: *NXdata*

Numerical description of the surface figure of the mirror.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXmirror.nxdl.xml

NXmoderator

Status:

base class, extends *NXobject*, version 1.0

Description:

A neutron moderator

Symbols:

No symbol table

Groups cited: *NXdata*, *NXgeometry*, *NXlog*

Structure:

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Effective distance as seen by measuring radiation

type: *NX_CHAR*

Any of these values:

- `H20`
- `D20`
- `Liquid H2`
- `Liquid CH4`
- `Liquid D2`
- `Solid D2`
- `C`
- `Solid CH4`
- `Solid H2`

poison_depth: *NX_FLOAT* {units=*NX_LENGTH*}

coupled: *NX_BOOLEAN*

whether the moderator is coupled

coupling_material: *NX_CHAR*

The material used for coupling. Usually Cd.

poison_material: *NX_CHAR*

Any of these values: `Gd` | `Cd`

temperature: *NX_FLOAT* {units=*NX_TEMPERATURE*}

average/nominal moderator temperature

(geometry): *NXgeometry*

"Engineering" position of moderator

temperature_log: *NXlog*

log file of moderator temperature

pulse_shape: *NXdata*

moderator pulse shape

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXmoderator.nxdl.xml

NXmonitor

Status:

base class, extends *NXobject*, version 1.0

Description:

A monitor of incident beam data.

It is similar to the *NXdata* groups containing monitor data and its associated dimension scale, e.g. time_of_flight or wavelength in pulsed neutron instruments. However, it may also include integrals, or scalar monitor counts, which are often used in both in both pulsed and steady-state instrumentation.

Symbols:

No symbol table

Groups cited: *NXgeometry*, *NXlog*

Structure:

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor` | `timer`

start_time: *NX_DATE_TIME*

Starting time of measurement

end_time: *NX_DATE_TIME*

Ending time of measurement

preset: *NX_NUMBER* {units=*NX_ANY*}

preset value for time or monitor

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Distance of monitor from sample

range[2]: *NX_FLOAT* {units=*NX_ANY*}

Range (X-axis, Time-of-flight, etc.) over which the integral was calculated

nominal: *NX_NUMBER* {units=*NX_ANY*}

Nominal reading to be used for normalisation purposes.

integral: *NX_NUMBER* {units=*NX_ANY*}

Total integral monitor counts

type: *NX_CHAR*

Any of these values: `Fission Chamber` | `Scintillator`

time_of_flight[ref(efficiency)]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

Time-of-flight

efficiency[ref(i)]: *NX_NUMBER* {units=*NX_DIMENSIONLESS*}

Monitor efficiency

data[n]: *NX_NUMBER* {units=*NX_ANY*}

Monitor data

sampled_fraction: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

Proportion of incident beam sampled by the monitor (0<x<1)

count_time: *NX_FLOAT* {units=*NX_TIME*}

Elapsed actual counting time, can be an array of size `np` when scanning. This is not the difference of the calendar time but the time the instrument was really counting, without pauses or times lost due beam unavailability

integral_log: *NXlog*

Time variation of monitor counts

(geometry): *NXgeometry*

Geometry of the monitor

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXmonitor.nxdl.xml

NXmonochromator

Status:

base class, extends *NXobject*, version 1.0

Description:

A wavelength defining device.

This is a base class for everything which selects a wavelength or energy, be it a monochromator crystal, a velocity selector, an undulator or whatever.

The expected units are:

- wavelength: angstrom
- energy: eV

Symbols:

No symbol table

Groups cited: *NXcrystal*, *NXdata*, *NXgeometry*, *NXgrating*, *NXvelocity_selector*

Structure:

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

wavelength selected

wavelength_error: *NX_FLOAT* {units=*NX_WAVELENGTH*}

wavelength standard deviation

energy: *NX_FLOAT* {units=*NX_ENERGY*}

energy selected

energy_error: *NX_FLOAT* {units=*NX_ENERGY*}

energy standard deviation

distribution: *NXdata*

geometry: *NXgeometry*

(crystal): *NXcrystal*

Use as many crystals as necessary to describe

(velocity_selector): *NXvelocity_selector*

(grating): *NXgrating*

For diffraction grating based monochromators

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXmonochromator.nxdl.xml

NXnote

Status:

base class, extends *NXobject*, version 1.0

Description:

Any additional freeform information not covered by the other base classes.

This class can be used to store additional information in a NeXus file e.g. pictures, movies, audio, additional text logs

Symbols:

No symbol table

Groups cited: none

Structure:

author: *NX_CHAR*

Author or creator of note

date: *NX_DATE_TIME*

Date note created/added

type: *NX_CHAR*

Mime content type of note data field e.g. image/jpeg, text/plain, text/html

file_name: *NX_CHAR*

Name of original file name if note was read from an external source

description: *NX_CHAR*

Title of an image or other details of the note

sequence_index: *NX_POSINT*

Sequence index of note, for placing a sequence of multiple **NXnote** groups in an order. Starts with 1.

data: *NX_BINARY*

Binary note data - if text, line terminator is [CR][LF].

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXnote.nxdl.xml

NXobject

Status:

base class, extends none, version 1.0

Description:

This is the base object of NeXus

Symbols:

No symbol table

Groups cited: none

Structure:

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXobject.nxdl.xml

NXorientation

Status:

base class, extends *NXobject*, version 1.0

Description:

legacy class - recommend to use *NXtransformations* now

Description for a general orientation of a component - used by *NXgeometry*

Symbols:

No symbol table

Groups cited: *NXgeometry*

Structure:

value[numobj, 6]: *NX_FLOAT* {units=*NX_UNITLESS*}

The orientation information is stored as direction cosines. The direction cosines will be between the local coordinate directions and the reference directions (to origin or relative NXgeometry). Calling the local unit vectors (x',y',z') and the reference unit vectors (x,y,z) the six numbers will be [x' dot x, x' dot y, x' dot z, y' dot x, y' dot y, y' dot z] where "dot" is the scalar dot product (cosine of the angle between the unit vectors). The unit vectors in both the local and reference coordinates are right-handed and orthonormal.

The pair of groups NXtranslation and NXorientation together describe the position of a component.

(geometry): *NXgeometry*

Link to another object if we are using relative positioning, else absent

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXorientation.nxdl.xml

NXparameters

Status:

base class, extends *NXobject*, version 1.0

Description:

Container for parameters, usually used in processing or analysis.

Symbols:

No symbol table

Groups cited: none

Structure:

term: *NX_CHAR*

A parameter (also known as a term) that is used in or results from processing.

@units: *NX_CHAR*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXparameters.nxdl.xml

NXpinhole

Status:

base class, extends *NXobject*, version 1.0

Description:

A simple pinhole.

For more complex geometries, *NXaperture* should be used.

Symbols:

No symbol table

Groups cited: none

Structure:

depends_on: *NX_CHAR*

Points to the path of the last element in the geometry chain that places this object in space. When followed through that chain is supposed to end at an element depending on "." i.e. the origin of the coordinate system. If desired the location of the slit can also be described relative to an NXbeam, which will allow a simple description of a non-centred pinhole.

diameter: *NX_NUMBER* {units=*NX_LENGTH*}

Size of the circular hole defining the transmitted beam size.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXpinhole.nxdl.xml

NXpolarizer

Status:

base class, extends *NXobject*, version 1.0

Description:

A spin polarizer.

Symbols:

No symbol table

Groups cited: none

Structure:

type: *NX_CHAR*

one of these values: "crystal", "supermirror", "3He"

composition: *NX_CHAR*

description of the composition of the polarizing material

reflection[3]: *NX_INT* {units=*NX_UNITLESS*}

[hkl] values of nominal reflection

efficiency: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

polarizing efficiency

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXpolarizer.nxdl.xml

NXpositioner

Status:

base class, extends *NXobject*, version 1.0

Description:

A generic positioner such as a motor or piezo-electric transducer.

It is used to document the current information of a piece of beam line equipment. Note: When using multiple *NXpositioner* groups, it is suggested to place them inside a *NXcollection* group. In such cases, when *NXpositioner* is used in another class, *NXcollection*/*NXpositioner* is then constructed.

Symbols:

No symbol table

Groups cited: none

Structure:

name: *NX_CHAR*

symbolic or mnemonic name (one word)

description: *NX_CHAR*

description of positioner

value[n]: *NX_NUMBER* {units=*NX_ANY*}

best known value of positioner - need [n] as may be scanned

raw_value[n]: *NX_NUMBER* {units=*NX_ANY*}

raw value of positioner - need [n] as may be scanned

target_value[n]: *NX_NUMBER* {units=*NX_ANY*}

targeted (commanded) value of positioner - need [n] as may be scanned

tolerance[n]: *NX_NUMBER* {units=*NX_ANY*}

maximum allowable difference between target_value and value

soft_limit_min: *NX_NUMBER* {units=*NX_ANY*}

minimum allowed limit to set value

soft_limit_max: *NX_NUMBER* {units=*NX_ANY*}

maximum allowed limit to set value

velocity: *NX_NUMBER* {units=*NX_ANY*}

velocity of the positioner (distance moved per unit time)

acceleration_time: *NX_NUMBER* {units=*NX_ANY*}

time to ramp the velocity up to full speed

controller_record: *NX_CHAR*

Hardware device record, e.g. EPICS process variable, taco/tango ...

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXpositioner.nxdl.xml

NXprocess

Status:

base class, extends *NXobject*, version 1.0

Description:

Document an event of data processing, reconstruction, or analysis for this data.

Symbols:

No symbol table

Groups cited: *NXnote*

Structure:

program: *NX_CHAR*

Name of the program used

sequence_index: *NX_POSINT*

Sequence index of processing, for determining the order of multiple **NXprocess** steps. Starts with 1.

version: *NX_CHAR*

Version of the program used

date: *NX_DATE_TIME*

Date and time of processing.

(note): *NXnote*

The note will contain information about how the data was processed or anything about the data provenance. The contents of the note can be anything that the processing code can understand, or simple text.

The name will be numbered to allow for ordering of steps.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXprocess.nxdl.xml

NXroot

Status:

base class, extends *NXobject*, version 1.0

Description:

Definition of the root NeXus group.

Symbols:

No symbol table

Groups cited: *NXentry*

Structure:

@NX_class: *NX_CHAR*

The root of any NeXus data file is an `NXroot` class (no other choice is allowed for a valid NeXus data file). This attribute cements that definition.

Obligatory value: `NXroot`

@file_time: *NX_CHAR*

Date and time file was originally created

@file_name: *NX_CHAR*

File name of original NeXus file

@file_update_time: *NX_CHAR*

Date and time of last file change at close

@NeXus_version: *NX_CHAR*

Version of NeXus API used in writing the file.

Only used when the NAPI has written the file. Note that this is different from the version of the base class or application definition version number.

@HDF_version: *NX_CHAR*

Version of HDF (version 4) library used in writing the file

@HDF5_Version: *NX_CHAR*

Version of HDF5 library used in writing the file.

Note this attribute is spelled with uppercase "V", different than other version attributes.

@XML_version: *NX_CHAR*

Version of XML support library used in writing the XML file

@h5py_version: *NX_CHAR*

Version of h5py Python package used in writing the file

@creator: *NX_CHAR*

facility or program where file originated

@default: *NX_CHAR*

Declares which *NXentry* group contains the data to be shown by default. It is needed to resolve ambiguity when more than one *NXentry* group exists. The value is the name of the default *NXentry* group.

It is recommended (as of NIAC2014) to use this attribute to help define the path to the default dataset to be plotted. See http://wiki.nexusformat.org/2014_How_to_find_default_data for a summary of the discussion.

(entry): *NXentry*

entries

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXroot.nxdl.xml

NXsample

Status:

base class, extends *NXobject*, version 1.0

Description:

Any information on the sample.

This could include scanned variables that are associated with one of the data dimensions, e.g. the magnetic field, or logged data, e.g. monitored temperature vs elapsed time.

Symbols:

symbolic array lengths to be coordinated between various fields

n_comp: number of compositions

n_Temp: number of temperatures

n_eField: number of values in applied electric field

n_mField: number of values in applied magnetic field

n_pField: number of values in applied pressure field

n_sField: number of values in applied stress field

Groups cited: *NXbeam*, *NXdata*, *NXenvironment*, *NXgeometry*, *NXlog*, *NXpositioner*

Structure:

name: *NX_CHAR*

Descriptive name of sample

chemical_formula: *NX_CHAR*

The chemical formula specified using CIF conventions. Abbreviated version of CIF standard:

- Only recognized element symbols may be used.
- Each element symbol is followed by a 'count' number. A count of '1' may be omitted.
- A space or parenthesis must separate each cluster of (element symbol + count).
- Where a group of elements is enclosed in parentheses, the multiplier for the group must follow the closing parentheses. That is, all element and group multipliers are assumed to be printed as subscripted numbers.
- Unless the elements are ordered in a manner that corresponds to their chemical structure, the order of the elements within any group or moiety depends on whether or not carbon is present.
- If carbon is present, the order should be:
  - C, then H, then the other elements in alphabetical order of their symbol.
  - If carbon is not present, the elements are listed purely in alphabetic order of their symbol.
- This is the *Hill* system used by Chemical Abstracts.

temperature[n_Temp]: *NX_FLOAT* {units=*NX_TEMPERATURE*}

Sample temperature. This could be a scanned variable

electric_field[n_eField]: *NX_FLOAT* {units=*NX_VOLTAGE*}

Applied electric field

@direction: *NX_CHAR*

Any of these values: `x` | `y` | `z`

magnetic_field[n_mField]: *NX_FLOAT* {units=*NX_ANY*}

Applied magnetic field

@direction: *NX_CHAR*

Any of these values: `x` | `y` | `z`

stress_field[n_sField]: *NX_FLOAT* {units=*NX_ANY*}

Applied external stress field

@direction: *NX_CHAR*

Any of these values: `x` | `y` | `z`

pressure[n_pField]: *NX_FLOAT* {units=*NX_PRESSURE*}

Applied pressure

changer_position: *NX_INT* {units=*NX_UNITLESS*}

Sample changer position

unit_cell[n_comp, 6]: *NX_FLOAT* {units=*NX_LENGTH*}

Unit cell parameters (lengths and angles)

unit_cell_volume[n_comp]: *NX_FLOAT* {units=*NX_VOLUME*}

Volume of the unit cell

sample_orientation[3]: *NX_FLOAT* {units=*NX_ANGLE*}

This will follow the Busing-Levy convention: W. R. Busing and H. A. Levy (1967). Acta Cryst. 22, 457-464

orientation_matrix[n_comp, 3, 3]: *NX_FLOAT*

Orientation matrix of single crystal sample using Busing-Levy convention: W. R. Busing and H. A. Levy (1967). Acta Cryst. 22, 457-464

mass[n_comp]: *NX_FLOAT* {units=*NX_MASS*}

Mass of sample

density[n_comp]: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

Density of sample

relative_molecular_mass[n_comp]: *NX_FLOAT* {units=*NX_MASS*}

Relative Molecular Mass of sample

type: *NX_CHAR*

Any of these values:

- `sample`
- `sample+can`
- `can`
- `calibration sample`

- `normalisation sample`
- `simulated data`
- `none`
- `sample environment`

situation: *NX_CHAR*

The atmosphere will be one of the components, which is where its details will be stored; the relevant components will be indicated by the entry in the sample_component member.

Any of these values:

- `air`
- `vacuum`
- `inert atmosphere`
- `oxidising atmosphere`
- `reducing atmosphere`
- `sealed can`
- `other`

description: *NX_CHAR*

Description of the sample

preparation_date: *NX_DATE_TIME*

Date of preparation of the sample

component[n_comp]: *NX_CHAR*

Details of the component of the sample and/or can

sample_component[n_comp]: *NX_CHAR*

Type of component

Any of these values: `sample` | `can` | `atmosphere` | `kit`

concentration[n_comp]: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

Concentration of each component

volume_fraction[n_comp]: *NX_FLOAT*

Volume fraction of each component

scattering_length_density[n_comp]: *NX_FLOAT* {units=*NX_SCATTERING_LENGTH_DENSITY*}

Scattering length density of each component

unit_cell_class[n_comp]: *NX_CHAR*

In case it is all we know and we want to record/document it

Any of these values:

- `cubic`
- `tetragonal`
- `orthorhombic`

- `monoclinic`
- `triclinic`

unit_cell_group[n_comp]: *NX_CHAR*

Crystallographic point or space group

path_length: *NX_FLOAT* {units=*NX_LENGTH*}

Path length through sample/can for simple case when it does not vary with scattering direction

path_length_window: *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of a beam entry/exit window on the can (mm) - assumed same for entry and exit

thickness: *NX_FLOAT* {units=*NX_LENGTH*}

sample thickness

external_DAC: *NX_FLOAT* {units=*NX_ANY*}

value sent to user's sample setup

short_title: *NX_CHAR*

20 character fixed length sample description for legends

rotation_angle: *NX_FLOAT* {units=*NX_ANGLE*}

Optional rotation angle for the case when the powder diagram has been obtained through an omega-2theta scan like from a traditional single detector powder diffractometer

x_translation: *NX_FLOAT* {units=*NX_LENGTH*}

Translation of the sample along the X-direction of the laboratory coordinate system

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Translation of the sample along the Z-direction of the laboratory coordinate system

geometry: *NXgeometry*

The position and orientation of the center of mass of the sample

(beam): *NXbeam*

Details of beam incident on sample - used to calculate sample/beam interaction point

transmission: *NXdata*

As a function of Wavelength

temperature_log: *NXlog*

temperature_log.value is a link to e.g. temperature_env.sensor1.value_log.value

temperature_env: *NXenvironment*

Additional sample temperature environment information

magnetic_field_log: *NXlog*

magnetic_field_log.value is a link to e.g. magnetic_field_env.sensor1.value_log.value

magnetic_field_env: *NXenvironment*

Additional sample magnetic environment information

external_ADC: *NXlog*

logged value (or logic state) read from user's setup

(positioner): *NXpositioner*

Any positioner (motor, PZT, ...) used to locate the sample

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXsample.nxdl.xml

NXsensor

Status:

base class, extends *NXobject*, version 1.0

Description:

A sensor used to monitor an external condition

The condition itself is described in *NXenvironment*.

Symbols:

No symbol table

Groups cited: *NXgeometry*, *NXlog*, *NXorientation*

Structure:

model: *NX_CHAR*

Sensor identification code/model number

name: *NX_CHAR*

Name for the sensor

short_name: *NX_CHAR*

Short name of sensor used e.g. on monitor display program

attached_to: *NX_CHAR*

where sensor is attached to ("sample" | "can")

measurement: *NX_CHAR*

name for measured signal

Any of these values:

- `temperature`
- `pH`
- `magnetic_field`
- `electric_field`
- `conductivity`
- `resistance`
- `voltage`
- `pressure`
- `flow`
- `stress`

- `strain`
- `shear`
- `surface_pressure`

type: *NX_CHAR*

The type of hardware used for the measurement. Examples (suggestions but not restrictions):

Temperature J | K | T | E | R | S | Pt100 | Rh/Fe

pH Hg/Hg2Cl2 | Ag/AgCl | ISFET

Ion selective electrode specify species; e.g. Ca2+

Magnetic field Hall

Surface pressure wilhelmy plate

run_control: *NX_BOOLEAN*

Is data collection controlled or synchronised to this quantity: 1=no, 0=to "value", 1=to "value_deriv1", etc.

high_trip_value: *NX_FLOAT* {units=*NX_ANY*}

Upper control bound of sensor reading if using run_control

low_trip_value: *NX_FLOAT* {units=*NX_ANY*}

Lower control bound of sensor reading if using run_control

value[n]: *NX_FLOAT* {units=*NX_ANY*}

nominal setpoint or average value - need [n] as may be a vector

value_deriv1[ref(value)]: *NX_FLOAT* {units=*NX_ANY*}

Nominal/average first derivative of value e.g. strain rate - same dimensions as "value" (may be a vector)

value_deriv2[ref(value)]: *NX_FLOAT* {units=*NX_ANY*}

Nominal/average second derivative of value - same dimensions as "value" (may be a vector)

external_field_brief: *NX_CHAR*

Any of these values:

- `along beam`
- `across beam`
- `transverse`
- `solenoidal`
- `flow shear gradient`
- `flow vorticity`

geometry: *NXgeometry*

Defines the axes for logged vector quantities if they are not the global instrument axes

value_log: *NXlog*

Time history of sensor readings

value_deriv1_log: *NXlog*

Time history of first derivative of sensor readings

value_deriv2_log: *NXlog*

Time history of second derivative of sensor readings

external_field_full: *NXorientation*

For complex external fields not satisfied by External_field_brief

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXsensor.nxdl.xml

NXshape

Status:

base class, extends *NXobject*, version 1.0

Description:

legacy class - (used by *NXgeometry*) - the shape and size of a component.

This is the description of the general shape and size of a component, which may be made up of `numobj` separate elements - it is used by the *NXgeometry* class

Symbols:

No symbol table

Groups cited: none

Structure:

shape: *NX_CHAR*

general shape of a component

Any of these values:

- `nxflat`
- `nxcylinder`
- `nxbox`
- `nxsphere`
- `nxcone`
- `nxelliptical`
- `nxtoroidal`
- `nxparabolic`
- `nxpolynomial`

size[numobj, nshapepar]: *NX_FLOAT* {units=*NX_LENGTH*}

physical extent of the object along its local axes (after NXorientation) with the center of mass at the local origin (after NXtranslation). The meaning and location of these axes will vary according to the value of the "shape" variable. `nshapepar` defines how many parameters:

- For "nxcylinder" type the parameters are (diameter,height) and a three value orientation vector of the cylinder.
- For the "nxbox" type the parameters are (length,width,height).

- For the "nxsphere" type the parameters are (diameter).
- For nxcone cone half aperture
- For nxelliptical, semi-major axis, semi-minor-axis, angle of major axis and pole
- For nxtoroidal, major radius, minor radius
- For nxparabolic, parabolic parameter a
- For nxpolynomial, an array of polynom coefficients, the dimension of the array encodes the degree of the polynom

direction: *NX_CHAR*

Any of these values: `concave` | `convex`

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXshape.nxdl.xml

NXslit

Status:

base class, extends *NXobject*, version 1.0

Description:

A simple slit.

For more complex geometries, *NXaperture* should be used.

Symbols:

No symbol table

Groups cited: none

Structure:

depends_on: *NX_CHAR*

Points to the path of the last element in the geometry chain that places this object in space. When followed through that chain is supposed to end at an element depending on "." i.e. the origin of the coordinate system. If desired the location of the slit can also be described relative to an NXbeam, which will allow a simple description of a non-centred slit.

x_gap: *NX_NUMBER* {units=*NX_LENGTH*}

Size of the gap opening in the first dimension of the local coordinate system.

y_gap: *NX_NUMBER* {units=*NX_LENGTH*}

Size of the gap opening in the second dimension of the local coordinate system.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXslit.nxdl.xml

NXsource

Status:

base class, extends *NXobject*, version 1.0

Description:

The neutron or x-ray storage ring/facility.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXgeometry*, *NXnote*

Structure:

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Effective distance from sample Distance as seen by radiation from sample. This number should be negative to signify that it is upstream of the sample.

name: *NX_CHAR*

Name of source

@short_name: *NX_CHAR*

short name for source, perhaps the acronym

type: *NX_CHAR*

type of radiation source (pick one from the enumerated list and spell exactly)

Any of these values:

- `Spallation Neutron Source`
- `Pulsed Reactor Neutron Source`
- `Reactor Neutron Source`
- `Synchrotron X-ray Source`
- `Pulsed Muon Source`
- `Rotating Anode X-ray`
- `Fixed Tube X-ray`
- `UV Laser`
- `Free-Electron Laser`
- `Optical Laser`
- `Ion Source`
- `UV Plasma Source`

probe: *NX_CHAR*

type of radiation probe (pick one from the enumerated list and spell exactly)

Any of these values:

- `neutron`
- `x-ray`
- `muon`
- `electron`
- `ultraviolet`
- `visible light`
- `positron`

- `proton`

power: *NX_FLOAT* {units=*NX_POWER*}

Source power

emittance_x: *NX_FLOAT* {units=*NX_EMITTANCE*}

Source emittance (nm-rad) in X (horizontal) direction.

emittance_y: *NX_FLOAT* {units=*NX_EMITTANCE*}

Source emittance (nm-rad) in Y (horizontal) direction.

sigma_x: *NX_FLOAT* {units=*NX_LENGTH*}

particle beam size in x

sigma_y: *NX_FLOAT* {units=*NX_LENGTH*}

particle beam size in y

flux: *NX_FLOAT* {units=*NX_FLUX*}

Source intensity/area (example: s-1 cm-2)

energy: *NX_FLOAT* {units=*NX_ENERGY*}

Source energy. For storage rings, this would be the particle beam energy. For X-ray tubes, this would be the excitation voltage.

current: *NX_FLOAT* {units=*NX_CURRENT*}

Accelerator, X-ray tube, or storage ring current

voltage: *NX_FLOAT* {units=*NX_VOLTAGE*}

Accelerator voltage

frequency: *NX_FLOAT* {units=*NX_FREQUENCY*}

Frequency of pulsed source

period: *NX_FLOAT* {units=*NX_PERIOD*}

Period of pulsed source

target_material: *NX_CHAR*

Pulsed source target material

Any of these values:

- `Ta`
- `W`
- `depleted_U`
- `enriched_U`
- `Hg`
- `Pb`
- `C`

number_of_bunches: *NX_INT*

For storage rings, the number of bunches in use.

bunch_length: *NX_FLOAT* {units=*NX_TIME*}

For storage rings, temporal length of the bunch

bunch_distance: *NX_FLOAT* {units=*NX_TIME*}

For storage rings, time between bunches

pulse_width: *NX_FLOAT* {units=*NX_TIME*}

temporal width of source pulse

mode: *NX_CHAR*

source operating mode

Any of these values:

- `Single Bunch`: for storage rings
- `Multi Bunch`: for storage rings

top_up: *NX_BOOLEAN*

Is the synchrotron operating in top_up mode?

last_fill: *NX_NUMBER* {units=*NX_CURRENT*}

For storage rings, the current at the end of the most recent injection.

@time: *NX_DATE_TIME*

date and time of the most recent injection.

notes: *NXnote*

any source/facility related messages/events that occurred during the experiment

bunch_pattern: *NXdata*

For storage rings, description of the bunch pattern. This is useful to describe irregular bunch patterns.

title: *NX_CHAR*

name of the bunch pattern

pulse_shape: *NXdata*

source pulse shape

geometry: *NXgeometry*

"Engineering" location of source

distribution: *NXdata*

The wavelength or energy distribution of the source

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXsource.nxdl.xml

NXsubentry

Status:

base class, extends *NXobject*, version 1.0

Description:

Group of multiple application definitions for "multi-modal" (e.g. SAXS/WAXS) measurements.

`NXsubentry` is a base class virtually identical to *NXentry* and is used as the (overlay) location for application definitions. Use a separate `NXsubentry` for each application definition.

To use `NXsubentry` with a hypothetical application definition called `NXmyappdef`:

- Create a group with attribute `NX_class="NXsubentry"`
- Within that group, create a field called `definition="NXmyappdef"`.
- There are two optional attributes of definition: `version` and `URL`

The intended use is to define application definitions for a multi-modal (a.k.a. multi-technique) *NXentry*. Previously, an application definition replaced *NXentry* with its own definition. With the increasing popularity of instruments combining multiple techniques for data collection (such as SAXS/WAXS instruments), it was recognized the application definitions must be entered in the NeXus data file tree as children of *NXentry*.

Symbols:

No symbol table

Groups cited: *NXcharacterization*, *NXcollection*, *NXdata*, *NXinstrument*, *NXmonitor*, *NXnote*, *NXparameters*, *NXprocess*, *NXsample*, *NXuser*

Structure:

@default: *NX_CHAR*

Declares which *NXdata* group contains the data to be shown by default. It is needed to resolve ambiguity when more than one *NXdata* group exists. The value is the name of the default *NXdata* group.

It is recommended (as of NIAC2014) to use this attribute to help define the path to the default dataset to be plotted. See http://wiki.nexusformat.org/2014_How_to_find_default_data for a summary of the discussion.

@IDF_Version: *NX_CHAR*

ISIS Muon IDF_Version

title: *NX_CHAR*

Extended title for entry

experiment_identifier: *NX_CHAR*

Unique identifier for the experiment, defined by the facility, possibly linked to the proposals

experiment_description: *NX_CHAR*

Brief summary of the experiment, including key objectives.

collection_identifier: *NX_CHAR*

User or Data Acquisition defined group of NeXus files or *NXentry*

collection_description: *NX_CHAR*

Brief summary of the collection, including grouping criteria.

entry_identifier: *NX_CHAR*

unique identifier for the measurement, defined by the facility.

definition: *NX_CHAR*

Official NeXus NXDL schema to which this subentry conforms

@version: *NX_CHAR*

NXDL version number

@URL: *NX_CHAR*

URL of NXDL file

definition_local: *NX_CHAR*

Local NXDL schema extended from the subentry specified in the `definition` field. This contains any locally-defined, additional fields in the subentry.

@version: *NX_CHAR*

NXDL version number

@URL: *NX_CHAR*

URL of NXDL file

start_time: *NX_DATE_TIME*

Starting time of measurement

end_time: *NX_DATE_TIME*

Ending time of measurement

duration: *NX_INT* {units=*NX_TIME*}

Duration of measurement

collection_time: *NX_FLOAT* {units=*NX_TIME*}

Time transpired actually collecting data i.e. taking out time when collection was suspended due to e.g. temperature out of range

run_cycle: *NX_CHAR*

Such as "2007-3". Some user facilities organize their beam time into run cycles.

program_name: *NX_CHAR*

Name of program used to generate this file

@version: *NX_CHAR*

Program version number

@configuration: *NX_CHAR*

configuration of the program

revision: *NX_CHAR*

Revision id of the file due to re-calibration, reprocessing, new analysis, new instrument definition format, ...

@comment: *NX_CHAR*

pre_sample_flightpath: *NX_FLOAT* {units=*NX_LENGTH*}

This is the flightpath before the sample position. This can be determined by a chopper, by the moderator or the source itself. In other words: it the distance to the component which gives the T0 signal to the detector electronics. If another component in the NXinstrument hierarchy provides this information, this should be a link.

experiment_documentation: *NXnote*

Description of the full experiment (document in pdf, latex, ...)

notes: *NXnote*

Notes describing entry

thumbnail: *NXnote*

A small image that is representative of the entry. An example of this is a 640x480 jpeg image automatically produced by a low resolution plot of the NXdata.

@mime_type: *NX_CHAR*

The value should be an `image/*`

Obligatory value: `image/*`

(characterization): *NXcharacterization*

(user): *NXuser*

(sample): *NXsample*

(instrument): *NXinstrument*

(collection): *NXcollection*

(monitor): *NXmonitor*

(data): *NXdata*

(parameters): *NXparameters*

(process): *NXprocess*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXsubentry.nxdl.xml

NXtransformations

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

Collection of translations and rotations to describe a geometry

A sequence of transformations lists the offset and rotation steps needed to describe the position and orientation of any movable or fixed device.

This class will usually contain all axes of a sample stage or goniometer.

The entry point (`depends_on`) will be outside of this class and point to a field in here. Following the chain may also require following `depends_on` links to transformations outside, for example to a common base table.

TODO: this needs an equation (issue #484)

Symbols:

No symbol table

Groups cited: none

Structure:

TRANSFORMATION: *NX_NUMBER*

Units need to be appropriate for translation or rotation

The name of this field is not defined. The user is free to use any name that does not cause confusion. When using more than one TRANSFORMATION field, make sure that each field name is unique in the same group, as required by HDF5.

@transformation_type: *NX_CHAR*

Any of these values: `translation|rotation`

@vector: *NX_NUMBER*

Three values that define the axis for this transformation

@offset: *NX_NUMBER*

A fixed offset applied before the transformation (three vector components).

@offset_units: *NX_CHAR*

Units of the offset.

@depends_on: *NX_CHAR*

Points to the name of the next element in the geometry chain.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXtransformations.nxdl.xml

NXtranslation

Status:

base class, extends *NXobject*, version 1.0

Description:

legacy class - (used by *NXgeometry*) - general spatial location of a component.

Symbols:

No symbol table

Groups cited: *NXgeometry*

Structure:

distances[numobj, 3]: *NX_FLOAT* {units=*NX_LENGTH*}

(x,y,z) This field describes the lateral movement of a component. The pair of groups NXtranslation and NXorientation together describe the position of a component. For absolute position, the origin is the scattering center (where a perfectly aligned sample would be) with the z-axis pointing downstream and the y-axis pointing gravitationally up. For a relative position the NXtranslation is taken into account before the NXorientation. The axes are right-handed and orthonormal.

geometry: *NXgeometry*

Link to other object if we are relative, else absent

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXtranslation.nxdl.xml

NXuser

Status:

base class, extends *NXobject*, version 1.0

Description:

Contact information for a user.

The format allows more than one user with the same affiliation and contact information, but a second *NXuser* group should be used if they have different affiliations, etc.

Symbols:

No symbol table

Groups cited: none

Structure:

name: *NX_CHAR*

Name of user responsible for this entry

role: *NX_CHAR*

Role of user responsible for this entry. Suggested roles are "local_contact", "principal_investigator", and "proposer"

affiliation: *NX_CHAR*

Affiliation of user

address: *NX_CHAR*

Address of user

telephone_number: *NX_CHAR*

Telephone number of user

fax_number: *NX_CHAR*

Fax number of user

email: *NX_CHAR*

Email of user

facility_user_id: *NX_CHAR*

facility based unique identifier for this person e.g. their identification code on the facility address/contact database

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXuser.nxdl.xml

NXvelocity_selector

Status:

base class, extends *NXobject*, version 1.0

Description:

A neutron velocity selector

Symbols:

No symbol table

Groups cited: *NXgeometry*

Structure:

type: *NX_CHAR*

velocity selector type

rotation_speed: *NX_FLOAT* {units=*NX_FREQUENCY*}

velocity selector rotation speed

radius: *NX_FLOAT* {units=*NX_LENGTH*}

radius at beam centre

spwidth: *NX_FLOAT* {units=*NX_LENGTH*}

spoke width at beam centre

length: *NX_FLOAT* {units=*NX_LENGTH*}

rotor length

num: *NX_INT* {units=*NX_UNITLESS*}

number of spokes/lamella

twist: *NX_FLOAT* {units=*NX_ANGLE*}

twist angle along axis

table: *NX_FLOAT* {units=*NX_ANGLE*}

offset vertical angle

height: *NX_FLOAT* {units=*NX_LENGTH*}

input beam height

width: *NX_FLOAT* {units=*NX_LENGTH*}

input beam width

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

wavelength

wavelength_spread: *NX_FLOAT* {units=*NX_WAVELENGTH*}

deviation FWHM /Wavelength

geometry: *NXgeometry*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXvelocity_selector.nxdl.xml

NXxraylens

Status:

base class, extends *NXobject*, version 1.0

Description:

An X-ray lens, typically at a synchrotron X-ray beam line.

Based on information provided by Gerd Wellenreuther (DESY).

Symbols:

No symbol table

Groups cited: *NXnote*

Structure:

lens_geometry: *NX_CHAR*

Geometry of the lens

Any of these values:

- `paraboloid`
- `spherical`
- `elliptical`
- `hyperbolical`

symmetric: *NX_BOOLEAN*

Is the device symmetric?

cylindrical: *NX_BOOLEAN*

Is the device cylindrical?

focus_type: *NX_CHAR*

The type of focus of the lens

Any of these values: `line` | `point`

lens_thickness: *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of the lens

lens_length: *NX_FLOAT* {units=*NX_LENGTH*}

Length of the lens

curvature: *NX_FLOAT* {units=*NX_LENGTH*}

Radius of the curvature as measured in the middle of the lens

aperture: *NX_FLOAT* {units=*NX_LENGTH*}

Diameter of the lens.

number_of_lenses: *NX_INT*

Number of lenses that make up the compound lens.

lens_material: *NX_CHAR*

Material used to make the lens.

gas: *NX_CHAR*

Gas used to fill the lens

gas_pressure: *NX_FLOAT* {units=*NX_PRESSURE*}

Gas pressure in the lens

cylinder_orientation: *NXnote*

Orientation of the cylinder axis.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/base_classes/NXxraylens.nxdl.xml

3.3.2 Application Definitions

A description of each NeXus application definition is given. 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. The definition, in this case, reserves the exact term by declaring its spelling and description. Consider an application definition as a *contract* between a data provider (such as the beam line control system) and a data consumer (such as a data analysis program for a scientific technique) that describes the information is certain to be available in a data file.

NXarchive This is a definition for data to be archived by ICAT (http://www.icatproject.org/).

NXarpes This is an application definition for angular resolved photo electron spectroscopy.

NXdirecttof This is a application definition for raw data from a direct geometry TOF spectrometer

NXfluo This is an application definition for raw data from an X-ray fluorescence experiment

NXindirecttof This is a application definition for raw data from a direct geometry TOF spectrometer

NXiqproc Application definition for any $I(Q)$ data.

NXlauetof This is the application definition for a TOF laue diffractometer

NXmonopd Monochromatic Neutron and X-Ray Powder diffractometer

NXmx functional application definition for macromolecular crystallography

NXrefscan This is an application definition for a monochromatic scanning reflectometer.

NXreftof This is an application definition for raw data from a TOF reflectometer.

NXsas raw data from 2-D monochromatic SAS data with an area detector

NXsastof This is an application definition for small angle scattering using a 2D detector in TOF mode.

NXscan Application definition for a generic scan instrument.

NXspe NXSPE Inelastic Format. Application definition for NXSPE file format.

NXsqom This is the application definition for S(Q,OM) processed data.

NXstxm Application definition for a STXM instrument.

NXtas This is an application definition for a triple axis spectrometer.

NXtofnpd This is a application definition for raw data from a TOF neutron powder diffractometer

NXtofraw This is an application definition for raw data from a generic TOF instrument

NXtofsingle This is a application definition for raw data from a generic TOF instrument

NXtomo This is the application definition for x-ray or neutron tomography raw data.

NXtomophase This is the application definition for x-ray or neutron tomography raw data with phase contrast variation at each point.

NXtomoproc This is an application definition for the final result of a tomography experiment: a 3D construction of some volume of physical properties.

NXxas This is an application definition for raw data from an X-ray absorption spectroscopy experiment.

NXxasproc Processed data from XAS. This is energy versus I(incoming)/I(absorbed).

NXxbase This definition covers the common parts of all monochromatic single crystal raw data application definitions.

NXxeuler raw data from a four-circle diffractometer with an eulerian cradle, extends *NXxbase*

NXxkappa raw data from a kappa geometry (CAD4) single crystal diffractometer, extends *NXxbase*

NXxlaue raw data from a single crystal laue camera, extends *NXxrot*

NXxlaueplate raw data from a single crystal Laue camera, extends *NXxlaue*

NXxnb raw data from a single crystal diffractometer, extends *NXxbase*

NXxrot raw data from a rotation camera, extends *NXxbase*

NXarchive

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is a definition for data to be archived by ICAT (http://www.icatproject.org/).

Symbols:

No symbol table

Groups cited: *NXentry*, *NXinstrument*, *NXsample*, *NXsource*, *NXuser*

Structure:

entry: *NXentry*

@index: *NX_CHAR*

title: *NX_CHAR*

experiment_identifier: *NX_CHAR*

unique identifier for the experiment

experiment_description: *NX_CHAR*

Brief description of the experiment and its objectives

collection_identifier: *NX_CHAR*

ID of user or DAQ define group of data files

collection_description: *NX_CHAR*

Brief summary of the collection, including grouping criteria

entry_identifier: *NX_CHAR*

unique identifier for this measurement as provided by the facility

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

duration: *NX_FLOAT* {units=*NX_TIME*}

TODO: needs documentation

collection_time: *NX_FLOAT* {units=*NX_TIME*}

TODO: needs documentation

run_cycle: *NX_CHAR*

TODO: needs documentation

revision: *NX_CHAR*

revision ID of this file, may be after recalibration, reprocessing etc.

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXarchive`

program: *NX_CHAR*

The program and version used for generating this file

@version: *NX_CHAR*

release_date: *NX_CHAR* {units=*NX_TIME*}

when this file is to be released into PD

user: *NXuser*

name: *NX_CHAR*

role: *NX_CHAR*

role of the user

facility_user_id: *NX_CHAR*

ID of the user in the facility burocracy database

instrument: *NXinstrument*

name: *NX_CHAR*

description: *NX_CHAR*

Brief description of the instrument

(source): *NXsource*

type: *NX_CHAR*

Any of these values:

- `Spallation Neutron Source`
- `Pulsed Reactor Neutron Source`
- `Reactor Neutron Source`
- `Synchrotron X-Ray Source`
- `Pulsed Muon Source`
- `Rotating Anode X-Ray`
- `Fixed Tube X-Ray`

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray` | `electron`

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

sample_id: *NX_CHAR*

Unique database id of the sample

description: *NX_CHAR*

type: *NX_CHAR*

Any of these values:

- `sample`
- `sample+can`
- `calibration sample`
- `normalisation sample`
- `simulated data`
- `none`
- `sample_environment`

chemical_formula: *NX_CHAR*

Chemical formula formatted according to CIF conventions

preparation_date: *NX_CHAR* {units=*NX_TIME*}

situation: *NX_CHAR*

Description of the environment the sample is in: air, vacuum, oxidizing atmosphere, dehydrated, etc.

temperature: *NX_FLOAT* {units=*NX_TEMPERATURE*}

magnetic_field: *NX_FLOAT* {units=*NX_CURRENT*}

electric_field: *NX_FLOAT* {units=*NX_VOLTAGE*}

stress_field: *NX_FLOAT* {units=*NX_UNITLESS*}

pressure: *NX_FLOAT* {units=*NX_PRESSURE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXarchive.nxdl.xml

NXarpes

Status:

application definition, extends *NXobject*, version 1.0

Description:

This is an application definition for angular resolved photo electron spectroscopy.

It has been drawn up with hemispherical electron analysers in mind.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonochromator*, *NXsample*, *NXsource*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

NeXus convention is to use “entry1”, “entry2”, ... for analysis software to locate each entry.

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms.

Obligatory value: `NXarpes`

(instrument): *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Obligatory value: `x-ray`

monochromator: *NXmonochromator*

energy: *NX_NUMBER* {units=*NX_ENERGY*}

analyser: *NXdetector*

data: *NX_NUMBER*

lens_mode: *NX_CHAR*

setting for the electron analyser lens

acquisition_mode: *NX_CHAR*

Any of these values: `swept` | `fixed`

entrance_slit_shape: *NX_CHAR*

Any of these values: `curved` | `straight`

entrance_slit_setting: *NX_NUMBER* {units=*NX_ANY*}

dial setting of the entrance slit

entrance_slit_size: *NX_CHAR* {units=*NX_LENGTH*}

size of the entrance slit

pass_energy: *NX_CHAR* {units=*NX_ENERGY*}

energy of the electrons on the mean path of the analyser

time_per_channel: *NX_CHAR* {units=*NX_TIME*}

todo: define more clearly

angles: *NX_NUMBER* {units=*NX_ANGLE*}

Angular axis of the analyser data which dimension the axis applies to is defined using the normal NXdata methods.

energies: *NX_NUMBER* {units=*NX_ENERGY*}

Energy axis of the analyser data which dimension the axis applies to is defined using the normal NXdata methods.

sensor_size[]: *NX_INT*

number of raw active elements in fast and slow pixel dimension direction

region_origin[]: *NX_INT*

origin of rectangular region selected for readout

region_size[]: *NX_INT*

size of rectangular region selected for readout

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

temperature: *NX_NUMBER* {units=*NX_TEMPERATURE*}

(data): *NXdata*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXarpes.nxdl.xml

NXdirecttof

Status:

application definition, extends *NXtofraw*, version 1.0b

Description:

This is a application definition for raw data from a direct geometry TOF spectrometer

Symbols:

No symbol table

Groups cited: *NXentry*, *NXfermi_chopper*, *NXinstrument*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXdirecttof`

(instrument): *NXinstrument*

fermi_chopper: *NXfermi_chopper*

rotation_speed: *NX_FLOAT* {units=*NX_FREQUENCY*}

chopper rotation speed

energy: *NX_FLOAT* {units=*NX_ENERGY*}

energy selected

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXdirecttof.nxdl.xml

NXfluo

Status:

application definition, extends *NXobject*, version 1.0

Description:

This is an application definition for raw data from an X-ray fluorescence experiment

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXmonochromator*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms.

Obligatory value: `NXfluo`

(instrument): *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Obligatory value: `x-ray`

monochromator: *NXmonochromator*

wavelength: *NX_FLOAT*

fluorescence: *NXdetector*

data[nenergy]: *NX_INT*

energy[nenergy]: *NX_FLOAT*

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

data: *NX_INT*

data: *NXdata*

energy –> /entry/instrument/fluorescence/energy

data –> /entry/instrument/fluorescence/data

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXfluo.nxdl.xml

NXindirecttof

Status:

application definition, extends *NXtofraw*, version 1.0b

Description:

This is a application definition for raw data from a direct geometry TOF spectrometer

Symbols:

No symbol table

Groups cited: *NXentry*, *NXinstrument*, *NXmonochromator*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXindirecttof`

(instrument): *NXinstrument*

analyser: *NXmonochromator*

energy[nDet]: *NX_FLOAT* {units=*NX_ENERGY*}

analyzed energy

polar_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

polar angle towards sample

distance[ndet]: *NX_FLOAT* {units=*NX_LENGTH*}

distance from sample

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXindirecttof.nxdl.xml

NXiqproc

Status:

application definition, extends *NXobject*, version 1.0b

Description:

Application definition for any $I(Q)$ data.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXentry*, *NXinstrument*, *NXparameters*, *NXprocess*, *NXsample*, *NXsource*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

title: *NX_CHAR*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXiqproc`

instrument: *NXinstrument*

name: *NX_CHAR*

Name of the instrument from which this data was reduced.

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray` | `electron`

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

reduction: *NXprocess*

program: *NX_CHAR*

version: *NX_CHAR*

input: *NXparameters*

Input parameters for the reduction program used

filenames: *NX_CHAR*

Raw data files used to generate this I(Q)

output: *NXparameters*

Eventual output parameters from the data reduction program used

(data): *NXdata*

data[NE, NQX, NQY]: *NX_INT*

This is I(Q). The client has to analyse the dimensions of I(Q). Often, multiple I(Q) for various environment conditions are measured; that would be the first dimension. Q can be multidimensional, this accounts for the further dimensions in the data

variable[NE]: *NX_CHAR*

@varied_variable: *NX_CHAR*

The real name of the varied variable in the first dim of data, temperature, P, MF etc...

qx[NQX]: *NX_CHAR*

Values for the first dimension of Q

qy[NQY]: *NX_CHAR*

Values for the second dimension of Q

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXiqproc.nxdl.xml

NXlauetof

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is the application definition for a TOF laue diffractometer

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXlauetof`

instrument: *NXinstrument*

detector: *NXdetector*

This assumes a planar 2D detector. All angles and distances refer to the center of the detector.

polar_angle: *NX_FLOAT* {units=*NX_ANGLE*}

The polar_angle (two theta) where the detector is placed.

azimuthal_angle: *NX_FLOAT* {units=*NX_ANGLE*}

The azimuthal angle where the detector is placed.

data[number of x pixels, number of y pixels, nTOF]: *NX_INT*

@signal: *NX_POSINT*

Obligatory value: `1`

x_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

distance: *NX_FLOAT* {units=*NX_LENGTH*}

time_of_flight[nTOF]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

orientation_matrix[3, 3]: *NX_FLOAT*

The orientation matrix according to Busing and Levy conventions. This is not strictly necessary as the UB can always be derived from the data. But let us bow to common usage which includes thie UB nearly always.

unit_cell[6]: *NX_FLOAT*

The unit cell, a, b, c, alpha, beta, gamma. Again, not strictly necessary, but normally written.

control: *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

data[nTOF]: *NX_INT*

use these attributes `primary=1 signal=1`

time_of_flight[nTOF]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

name: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

time_of_flight –> /NXentry/NXinstrument/NXdetector/time_of_flight

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXlauetof.nxdl.xml

NXmonopd

Status:

application definition, extends *NXobject*, version 1.0b

Description:

Monochromatic Neutron and X-Ray Powder diffractometer

Instrument definition for a powder diffractometer at a monochromatic neutron or X-ray beam. This is both suited for a powder diffractometer with a single detector or a powder diffractometer with a position sensitive detector.

Symbols:

No symbol table

Groups cited: *NXcrystal*, *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXmonopd`

(instrument): *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray` | `electron`

(crystal): *NXcrystal*

wavelength[i]: *NX_FLOAT* {units=*NX_WAVELENGTH*}

Optimum diffracted wavelength

(detector): *NXdetector*

polar_angle[ndet]: *NX_FLOAT*

where ndet = number of detectors

data[ndet]: *NX_INT*

detector signal (usually counts) are already corrected for detector efficiency

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

rotation_angle: *NX_FLOAT* {units=*NX_ANGLE*}

Optional rotation angle for the case when the powder diagram has been obtained through an omega-2theta scan like from a traditional single detector powder diffractometer

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor` | `timer`

preset: *NX_FLOAT*

preset value for time or monitor

integral: *NX_FLOAT* {units=*NX_ANY*}

Total integral monitor counts

(data): *NXdata*

polar_angle –> /NXentry/NXinstrument/NXdetector/polar_angle

Link to polar angle in /NXentry/NXinstrument/NXdetector

data –> /NXentry/NXinstrument/NXdetector/data

Link to data in /NXentry/NXinstrument/NXdetector

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXmonopd.nxdl.xml

NXmx

Status:

application definition, extends *NXobject*, version 1.4

Description:

functional application definition for macromolecular crystallography

Symbols:

These symbols will be used below to coordinate datasets with the same shape.

np: number of scan points

i: number of detector pixels in the slow direction

j: number of detector pixels in the fast direction

Groups cited: *NXattenuator*, *NXbeam*, *NXcollection*, *NXdata*, *NXdetector_module*, *NXdetector*, *NXentry*, *NXinstrument*, *NXsample*, *NXtransformations*

Structure:

(entry): *NXentry*

title: (optional) *NX_CHAR*

start_time: (optional) *NX_DATE_TIME*

end_time: (optional) *NX_DATE_TIME*

definition: *NX_CHAR*

NeXus NXDL schema to which this file conforms

Obligatory value: `NXmx`

(instrument): *NXinstrument*

(attenuator): (optional) *NXattenuator*

attenuator_transmission: (optional) *NX_NUMBER* {units=*NX_UNITLESS*}

(detector): *NXdetector*

depends_on: *NX_CHAR*

data[np, i, j]: *NX_NUMBER*

description: (optional) *NX_CHAR*

name/manufacturer/model/etc. information

time_per_channel: (optional) *NX_CHAR* {units=*NX_TIME*}

todo: define more clearly

distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

Distance from the sample to the beam center. This value is a guidance only, the proper geometry can be found following the depends_on axis chain.

dead_time: (optional) *NX_FLOAT* {units=*NX_TIME*}

Detector dead time

count_time: (optional) *NX_NUMBER* {units=*NX_TIME*}

Elapsed actual counting time

beam_center_x: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

This is the x position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

beam_center_y: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

This is the y position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

angular_calibration_applied: (optional) *NX_BOOLEAN*

True when the angular calibration has been applied in the electronics, false otherwise.

angular_calibration[i, j]: (optional) *NX_FLOAT*

Angular calibration data.

flatfield_applied: (optional) *NX_BOOLEAN*

True when the flat field correction has been applied in the electronics, false otherwise.

flatfield[i, j]: (optional) *NX_FLOAT*

Flat field correction data.

flatfield_error[i, j]: (optional) *NX_FLOAT*

Errors of the flat field correction data.

pixel_mask_applied: (optional) *NX_BOOLEAN*

True when the pixel mask correction has been applied in the electronics, false otherwise.

pixel_mask[i, j]: (optional) *NX_INT*

The 32-bit pixel mask for the detector. Contains a bit field for each pixel to signal dead, blind or high or otherwise unwanted or undesirable pixels. They have the following meaning:

- bit 0: gap (pixel with no sensor)
- bit 1: dead
- bit 2: under responding
- bit 3: over responding
- bit 4: noisy
- bit 5: -undefined-
- bit 6: pixel is part of a cluster of problematic pixels (bit set in addition to others)
- bit 7: -undefined-
- bit 8: user defined mask (e.g. around beamstop)
- bits 9-30: -undefined-
- bit 31: virtual pixel (corner pixel with interpolated value)

Normal data analysis software would not take pixels into account when a bit in (mask & 0x0000FFFF) is set. Tag bit in the upper two bytes would indicate special pixel properties that normally would not be a sole reason to reject the intensity value (unless lower bits are set.

If the full bit depths is not required, providing a mask with fewer bits is permissible.

countrate_correction_applied: (optional) *NX_BOOLEAN*

True when a count-rate correction has already been applied in the data recorded here, false otherwise.

bit_depth_readout: (optional) *NX_INT*

How many bits the electronics record per pixel.

detector_readout_time: (optional) *NX_FLOAT* {units=*NX_TIME*}

Time it takes to read the detector (typically milliseconds). This is important to know for time resolved experiments.

frame_time: (optional) *NX_FLOAT* {units=*NX_TIME*}

This is time for each frame. This is exposure_time + readout time.

gain_setting: (optional) *NX_CHAR*

The gain setting of the detector. This influences background.

saturation_value: (optional) *NX_INT*

The value at which the detector goes into saturation. Data above this value is known to be invalid.

sensor_material: (optional) *NX_CHAR*

At times, radiation is not directly sensed by the detector. Rather, the detector might sense the output from some converter like a scintillator. This is the name of this converter material.

sensor_thickness: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

At times, radiation is not directly sensed by the detector. Rather, the detector might sense the output from some converter like a scintillator. This is the thickness of this converter material.

threshold_energy: (optional) *NX_FLOAT* {units=*NX_ENERGY*}

Single photon counter detectors can be adjusted for a certain energy range in which they work optimally. This is the energy setting for this.

type: (optional) *NX_CHAR*

Description of type such as scintillator, ccd, pixel, image plate, CMOS, ...

(transformations): (optional) *NXtransformations*

Suggested location for axes (transformations) to do with the detector

(collection): (optional) *NXcollection*

Suggested container for detailed non-standard detector information like corrections applied automatically or performance settings.

(detector_module): *NXdetector_module*

Many detectors consist of multiple smaller modules that are operated in sync and store their data in a common dataset. To allow consistent parsing of the experimental geometry, this application definiton requires all detectors to define a detector module, even if there is only one.

data_origin: *NX_INT*

A two value field which gives the index of the start of the modules data in the main area detector image in the underlying NXdetector module.

data_size: *NX_INT*

Two values for the size of the module in pixels in each direction.

module_offset: *NX_NUMBER* {units=*NX_LENGTH*}

Offset of the module in regards to the origin of the detector in an arbitrary direction.

@transformation_type: *NX_CHAR*

Obligatory value: `translation`

@vector: *NX_CHAR*

@offset: *NX_CHAR*

@depends_on: *NX_CHAR*

fast_pixel_direction: *NX_NUMBER* {units=*NX_LENGTH*}

Values along the direction of fastest varying pixel direction.The direction itself is given through the vector attribute

@transformation_type: *NX_CHAR*

Obligatory value: `translation`

@vector: *NX_CHAR*

@offset: *NX_CHAR*

@depends_on: *NX_CHAR*

slow_pixel_direction: *NX_NUMBER* {units=*NX_LENGTH*}

Values along the direction of slow varying pixel direction. The direction itself is given through the vector attribute

@transformation_type: *NX_CHAR*

Obligatory value: `translation`

@vector: *NX_CHAR*

@offset: *NX_CHAR*

@depends_on: *NX_CHAR*

(sample): *NXsample*

name: (optional) *NX_CHAR*

Descriptive name of sample

depends_on: (optional) *NX_CHAR*

This should be an absolute requirement to have for any scan experiment. The reason it is optional is mainly to accommodate XFEL single shot exposures.

temperature: (optional) *NX_CHAR* {units=*NX_TEMPERATURE*}

(beam): *NXbeam*

incident_wavelength: (optional) *NX_NUMBER* {units=*NX_WAVELENGTH*}

flux: (optional) *NX_FLOAT* {units=*NX_FLUX*}

flux incident on beam plane area

incident_polarisation_stokes[np, 4]: (optional) *NX_CHAR*

incident_wavelength_spectrum: (optional) *NXdata*

(transformations): (optional) *NXtransformations*

Suggested location for sample goniometer or other axes (transformations)

(data): *NXdata*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXmx.nxdl.xml

NXrefscan

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is an application definition for a monochromatic scanning reflectometer.

It does not have the information to calculate the resolution since it does not have any apertures.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXmonochromator*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXrefscan`

instrument: *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray` | `electron`

monochromator: *NXmonochromator*

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

(detector): *NXdetector*

data[NP]: *NX_INT*

polar_angle[NP]: *NX_FLOAT* {units=*NX_ANGLE*}

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

rotation_angle[NP]: *NX_FLOAT* {units=*NX_ANGLE*}

control: *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor` | `timer`

preset: *NX_FLOAT*

preset value for time or monitor

data[NP]: *NX_FLOAT* {units=*NX_ANY*}

Monitor counts for each step

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

rotation_angle –> /NXentry/NXsample/rotation_angle

polar_angle –> /NXentry/NXinstrument/NXdetector/polar_angle

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXrefscan.nxdl.xml

NXreftof

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is an application definition for raw data from a TOF reflectometer.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXdisk_chopper*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXreftof`

instrument: *NXinstrument*

name: *NX_CHAR*

chopper: *NXdisk_chopper*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Distance between chopper and sample

detector: *NXdetector*

data[xsize, ysize, nTOF]: *NX_INT*

time_of_flight[nTOF]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

Array of time values for each bin in a time-of-flight measurement

distance: *NX_FLOAT* {units=*NX_LENGTH*}

polar_angle: *NX_FLOAT* {units=*NX_ANGLE*}

x_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

rotation_angle: *NX_FLOAT* {units=*NX_ANGLE*}

control: *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT* {units=*NX_ANY*}

preset value for time or monitor

integral: *NX_INT*

Total integral monitor counts

time_of_flight: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

Time channels

data: *NX_INT*

Monitor counts in each time channel

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

time_binning –> /NXentry/NXinstrument/NXdetector/time_binning

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXreftof.nxdl.xml

NXsas

Status:

application definition, extends *NXobject*, version 1.0b

Description:

raw data from 2-D monochromatic SAS data with an area detector

This is an application definition for raw data (not processed or reduced data) from a 2-D small angle scattering instrument collected with a monochromatic beam and an area detector. It is meant to be suitable both for neutron SANS and X-ray SAXS data.

It covers all raw data from all SAS techniques: SAS, WSAS, grazing incidence, GISAS

Symbols:

No symbol table

Groups cited: *NXcollimator*, *NXdata*, *NXdetector*, *NXentry*, *NXgeometry*, *NXinstrument*, *NXmonitor*, *NXmonochromator*, *NXsample*, *NXshape*, *NXsource*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

NeXus convention is to use `entry1`, `entry2`, ... for analysis software to locate each entry

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXsas`

instrument: *NXinstrument*

name: *NX_CHAR*

Name of the instrument actually used to perform the experiment

source: *NXsource*

type: *NX_CHAR*

type of radiation source

name: *NX_CHAR*

Name of the radiation source

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray`

monochromator: *NXmonochromator*

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

The wavelength of the radiation

wavelength_spread: *NX_FLOAT*

delta_lambda/lambda ($\Delta\lambda/\lambda$): Important for resolution calculations

collimator: *NXcollimator*

geometry: *NXgeometry*

shape: *NXshape*

shape: *NX_CHAR*

Any of these values: `nxcylinder` | `nxbox`

size: *NX_FLOAT* {units=*NX_LENGTH*}

The collimation length

detector: *NXdetector*

data[nXPixel, nYPixel]: *NX_NUMBER*

This is area detector data, of number of x-pixel versus number of y-pixels. Since the beam center is to be determined as a step of data reduction, it is not necessary to document or assume the position of the beam center in acquired data.

distance: *NX_FLOAT* {units=*NX_LENGTH*}

The distance between detector and sample

x_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

Physical size of a pixel in x-direction

y_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

Size of a pixel in y direction

polar_angle: *NX_FLOAT* {units=*NX_ANGLE*}

azimuthal_angle: *NX_FLOAT* {units=*NX_ANGLE*}

rotation_angle: *NX_FLOAT* {units=*NX_ANGLE*}

aequatorial_angle: *NX_FLOAT* {units=*NX_ANGLE*}

beam_center_x: *NX_FLOAT* {units=*NX_LENGTH*}

This is the x position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

beam_center_y: *NX_FLOAT* {units=*NX_LENGTH*}

This is the y position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

aequatorial_angle: *NX_FLOAT* {units=*NX_ANGLE*}

control: *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor` | `timer`

preset: *NX_FLOAT*

preset value for time or monitor

integral: *NX_FLOAT* {units=*NX_ANY*}

Total integral monitor counts

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXsas.nxdl.xml

NXsastof

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is an application definition for small angle scattering using a 2D detector in TOF mode.

It strives to cover all the SAS techniques in the file again

Symbols:

No symbol table

Groups cited: *NXcollimator*, *NXdata*, *NXdetector*, *NXentry*, *NXgeometry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXshape*, *NXsource*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

NeXus convention is to use "entry1", "entry2", ... for analysis software to locate each entry

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXsastof`

instrument: *NXinstrument*

name: *NX_CHAR*

Name of the instrument actually used to perform the experiment

source: *NXsource*

type: *NX_CHAR*

type of radiation source

name: *NX_CHAR*

Name of the radiation source

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray`

collimator: *NXcollimator*

geometry: *NXgeometry*

shape: *NXshape*

shape: *NX_CHAR*

Any of these values: `nxcylinder` | `nxbox`

size: *NX_FLOAT* {units=*NX_LENGTH*}

The collimation length

detector: *NXdetector*

data[nXPixel, nYPixel, nTOF]: *NX_NUMBER*

This is area detector data, of number of x-pixel versus number of y-pixels. Since the beam center is to be determined as a step of data reduction, it is not necessary to document or assume the position of the beam center in acquired data.

time_of_flight[nTOF]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

distance: *NX_FLOAT* {units=*NX_LENGTH*}

The distance between detector and sample

x_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

Physical size of a pixel in x-direction

y_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

Size of a pixel in y direction

polar_angle: *NX_FLOAT* {units=*NX_ANGLE*}

azimuthal_angle: *NX_FLOAT* {units=*NX_ANGLE*}

rotation_angle: *NX_FLOAT* {units=*NX_ANGLE*}

aequatorial_angle: *NX_FLOAT* {units=*NX_ANGLE*}

beam_center_x: *NX_FLOAT* {units=*NX_LENGTH*}

This is the x position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

beam_center_y: *NX_FLOAT* {units=*NX_LENGTH*}

This is the y position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

aequatorial_angle: *NX_FLOAT* {units=*NX_ANGLE*}

control: *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

data[nTOF]: *NX_INT*

time_of_flight[nTOF]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

time_of_flight –> /NXentry/NXinstrument/NXdetector/time_of_flight

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXsastof.nxdl.xml

NXscan

Status:

application definition, extends *NXobject*, version 1.0b

Description:

Application definition for a generic scan instrument.

This definition is more an example then a stringent definition as the content of a given NeXus scan file needs to differ for different types of scans. This example definition shows a scan like done on a rotation camera: the sample is rotated and a detector image, the rotation angle and a monitor value is stored at each step in the scan. In the following, the symbol `NP` is used to represent the number of scan points. These are the rules for storing scan data in NeXus files which are implemented in this example:

- Each value varied throughout a scan is stored as an array of length `NP` at its respective location within the NeXus hierarchy.
- For area detectors, `NP` is the first dimension, example for a detector of 256x256: `data[NP,256,256]`
- The NXdata group contains links to all variables varied in the scan and the data. This to give an equivalent to the more familiar classical tabular representation of scans.

These rules exist for a reason: HDF allows the first dimension of a data set to be unlimited. This means the data can be appended too. Thus a NeXus file built according to the rules given above can be used in the following way:

- At the start of a scan, write all the static information.
- At each scan point, append new data from varied variables and the detector to the file.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*

Structure:

(entry): *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXscan`

(instrument): *NXinstrument*

(detector): *NXdetector*

data[NP, xdim, ydim]: *NX_INT*

(sample): *NXsample*

rotation_angle[NP]: *NX_FLOAT*

(monitor): *NXmonitor*

data[NP]: *NX_INT*

(data): *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

rotation_angle –> /NXentry/NXsample/rotation_angle

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXscan.nxdl.xml

NXspe

Status:

application definition, extends *NXobject*, version 1.0

Description:

NXSPE Inelastic Format. Application definition for NXSPE file format.

Symbols:

No symbol table

Groups cited: *NXcollection*, *NXdata*, *NXentry*, *NXfermi_chopper*, *NXinstrument*, *NXsample*

Structure:

(entry): *NXentry*

program_name: *NX_CHAR*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms.

Any of these values: `NXSPE` | `NXspe`

@version: *NX_CHAR*

NXSPE_info: *NXcollection*

fixed_energy: *NX_FLOAT* {units=*NX_ENERGY*}

The fixed energy used for this file.

ki_over_kf_scaling: *NX_BOOLEAN*

Indicates whether ki/kf scaling has been applied or not.

psi: *NX_FLOAT* {units=*NX_ANGLE*}

Orientation angle as expected in DCS-MSlice

data: *NXdata*

azimuthal: *NX_FLOAT* {units=*NX_ANGLE*}

azimuthal_width: *NX_FLOAT* {units=*NX_ANGLE*}

polar: *NX_FLOAT* {units=*NX_ANGLE*}

polar_width: *NX_FLOAT* {units=*NX_ANGLE*}

distance: *NX_FLOAT* {units=*NX_LENGTH*}

data: *NX_NUMBER*

error: *NX_NUMBER*

energy: *NX_FLOAT* {units=*NX_ENERGY*}

(instrument): *NXinstrument*

name: *NX_CHAR*

(fermi_chopper): *NXfermi_chopper*

energy: *NX_NUMBER* {units=*NX_ENERGY*}

(sample): *NXsample*

rotation_angle: *NX_NUMBER* {units=*NX_ANGLE*}

seblock: *NX_CHAR*

temperature: *NX_NUMBER* {units=*NX_TEMPERATURE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXspe.nxdl.xml

NXsqom

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is the application definition for S(Q,OM) processed data.

As this kind of data is in general not on a rectangular grid after data reduction, it is stored as Q,E positions plus their intensity, table like. It is the task of a possible visualisation program to regrid this data in a sensible way.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXentry*, *NXinstrument*, *NXparameters*, *NXprocess*, *NXsample*, *NXsource*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

title: *NX_CHAR*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXsqom`

instrument: *NXinstrument*

name: *NX_CHAR*

Name of the instrument from which this data was reduced.

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray` | `electron`

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

reduction: *NXprocess*

program: *NX_CHAR*

version: *NX_CHAR*

input: *NXparameters*

Input parameters for the reduction program used

filenames: *NX_CHAR*

Raw data files used to generate this I(Q)

output: *NXparameters*

Eventual output parameters from the data reduction program used

(data): *NXdata*

data[NP]: *NX_INT*

This is the intensity for each point in QE

qx[NP]: *NX_CHAR* {units=*NX_WAVENUMBER*}

Positions for the first dimension of Q

qy[NP]: *NX_CHAR* {units=*NX_WAVENUMBER*}

Positions for the the second dimension of Q

qz[NP]: *NX_CHAR* {units=*NX_WAVENUMBER*}

Positions for the the third dimension of Q

en[NP]: *NX_FLOAT* {units=*NX_ENERGY*}

Values for the energy transfer for each point

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXsqom.nxdl.xml

NXstxm

Status:

application definition, extends *NXobject*, version 1.1

Description:

Application definition for a STXM instrument.

The interferometer position measurements, monochromator photon energy values and detector measurements are all treated as NXdetectors and stored within the NXinstrument group as lists of values stored in chronological order. The NXdata group then holds another version of the data in a regular 3D array (NumE by NumY by NumX, for a total of NumP points in a sample image stack type scan). The former data values should be stored with a minimum loss of precision, while the latter values can be simplified and/or approximated in order to fit the constraints of a regular 3D array. 'Line scans' and 'point spectra' are just sample_image scan types with reduced dimensions in the same way as single images have reduced E dimensions compared to image 'stacks'.

Symbols:

These symbols will be used below to coordinate the shapes of the datasets.

numP: total number of scan points

numE: number of photon energies scanned

numY: number of pixels in Y direction

numX: number of pixels in X direction

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXmonochromator*, *NXsample*, *NXsource*

Structure:

(entry): *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXstxm`

(instrument): *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

monochromator: *NXmonochromator*

energy[NumP]: *NX_CHAR*

(detector): *NXdetector*

data[NumP]: *NX_NUMBER*

sample_x: (optional) *NXdetector*

Measurements of the sample position from the x-axis interferometer.

data[NumP]: *NX_FLOAT*

sample_y: (optional) *NXdetector*

Measurements of the sample position from the y-axis interferometer.

data[NumP]: *NX_FLOAT*

sample_z: (optional) *NXdetector*

Measurements of the sample position from the z-axis interferometer.

data[NumP]: *NX_FLOAT*

(sample): *NXsample*

rotation_angle: *NX_FLOAT*

(data): *NXdata*

stxm_scan_type: *NX_CHAR*

Label for typical scan types as a convenience for humans. Each label corresponds to a specific set of axes being scanned to produce a data array of shape:

- sample point spectrum: (photon_energy,)

- sample line spectrum: (photon_energy, sample_y/sample_x)
- sample image: (sample_y, sample_x)
- sample image stack: (photon_energy, sample_y, sample_x)
- sample focus: (zoneplate_z, sample_y/sample_x)
- osa image: (osa_y, osa_x)
- osa focus: (zoneplate_z, osa_y/osa_x)
- detector image: (detector_y, detector_x)

The "generic scan" string is to be used when none of the other choices are appropriate.

Any of these values:

- `sample point spectrum`
- `sample line spectrum`
- `sample image`
- `sample image stack`
- `sample focus`
- `osa image`
- `osa focus`
- `detector image`
- `generic scan`

data: *NX_NUMBER*

Detectors that provide more than one value per scan point should be summarised to a single value per scan point for this array in order to simplify plotting.

Note that 'Line scans' and focus type scans measure along one spatial dimension but are not restricted to being parallel to the X or Y axes. Such scans should therefore use a single dimension for the positions along the spatial line. The 'sample_x' and 'sample_y' fields should then contain lists of the x- and y-positions and should both have the 'axis' attribute pointing to the same dimension.

energy[NumE]: *NX_FLOAT*

List of photon energies of the X-ray beam. If scanned through multiple values, then an 'axis' attribute will be required to link the field to the appropriate data array dimension.

sample_y[NumY]: *NX_FLOAT*

List of Y positions on the sample. If scanned through multiple values, then an 'axis' attribute will be required to link the field to the appropriate data array dimension.

sample_x[NumX]: *NX_FLOAT*

List of X positions on the sample. If scanned through multiple values, then an 'axis' attribute will be required to link the field to the appropriate data array dimension.

control: (optional) *NXmonitor*

data: *NX_FLOAT*

Values to use to normalise for time-variations in photon flux. Typically, the synchrotron storage ring electron beam current is used as a proxy for the X-ray beam intensity.
Array must have same shape as the NXdata groups.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXstxm.nxdl.xml

NXtas

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is an application definition for a triple axis spectrometer.

It is for the trademark scan of the TAS, the Q-E scan. For your alignment scans use the rules in *NXscan*.

Symbols:

No symbol table

Groups cited: *NXcrystal*, *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtas`

(instrument): *NXinstrument*

(source): *NXsource*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron` | `x-ray`

monochromator: *NXcrystal*

ei[np]: *NX_FLOAT* {units=*NX_ENERGY*}

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

analyser: *NXcrystal*

ef[np]: *NX_FLOAT* {units=*NX_ENERGY*}

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

polar_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

(detector): *NXdetector*

data[np]: *NX_INT*

polar_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

qh[np]: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

qk[np]: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

ql[np]: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

en[np]: *NX_FLOAT* {units=*NX_ENERGY*}

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

polar_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

sgu[np]: *NX_FLOAT* {units=*NX_ANGLE*}

sgl[np]: *NX_FLOAT* {units=*NX_ANGLE*}

unit_cell[6]: *NX_FLOAT* {units=*NX_LENGTH*}

orientation_matrix[9]: *NX_FLOAT* {units=*NX_DIMENSIONLESS*}

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor` | `timer`

preset: *NX_FLOAT*

preset value for time or monitor

data[np]: *NX_FLOAT* {units=*NX_ANY*}

Total integral monitor counts

(data): *NXdata*

One of the ei,ef,qh,qk,ql,en should get a primary=1 attribute to denote the main scan axis

ei –> /NXentry/NXinstrument/monochromator:NXcrystal/ei

ef –> /NXentry/NXinstrument/analyzer:NXcrystal/ef

en –> /NXentry/NXsample/en

qh –> /NXentry/NXsample/qh

qk –> /NXentry/NXsample/qk

ql –> /NXentry/NXsample/ql

data –> /NXentry/NXinstrument/NXdetector/data

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtas.nxdl.xml

NXtofnpd

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is a application definition for raw data from a TOF neutron powder diffractometer

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXuser*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtofnpd`

pre_sample_flightpath: *NX_FLOAT* {units=*NX_LENGTH*}

This is the flight path before the sample position. This can be determined by a chopper, by the moderator or the source itself. In other words: it the distance to the component which gives the T0 signal to the detector electronics. If another component in the NXinstrument hierarchy provides this information, this should be a link.

user: *NXuser*

name: *NX_CHAR*

(instrument): *NXinstrument*

detector: *NXdetector*

data[ndet, ntimechan]: *NX_INT*

detector_number[ndet]: *NX_INT*

distance[ndet]: *NX_FLOAT* {units=*NX_LENGTH*}

distance to sample for each detector

time_of_flight[ntimechan]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

polar_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

polar angle for each detector element

azimuthal_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

azimuthal angle for each detector element

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

distance: *NX_FLOAT* {units=*NX_LENGTH*}

data[ntimechan]: *NX_INT*

time_of_flight[ntimechan]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

detector_number –> /NXentry/NXinstrument/NXdetector/detector_number

time_of_flight –> /NXentry/NXinstrument/NXdetector/time_of_flight

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtofnpd.nxdl.xml

NXtofraw

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is an application definition for raw data from a generic TOF instrument

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXuser*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtofraw`

duration: *NX_FLOAT*

run_number: *NX_INT*

pre_sample_flightpath: *NX_FLOAT* {units=*NX_LENGTH*}

This is the flight path before the sample position. This can be determined by a chopper, by the moderator, or the source itself. In other words: it is the distance to the component which gives the T0 signal to the detector electronics. If another component in the NXinstrument hierarchy provides this information, this should be a link.

user: *NXuser*

name: *NX_CHAR*

instrument: *NXinstrument*

detector: *NXdetector*

data[ndet, ntimechan]: *NX_INT*

detector_number[ndet]: *NX_INT*

distance[ndet]: *NX_FLOAT* {units=*NX_LENGTH*}

distance to sample for each detector

time_of_flight[ntimechan]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

polar_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

polar angle for each detector element

azimuthal_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

polar angle for each detector element

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

nature: *NX_CHAR*

Any of these values: `powder|liquid|single crystal`

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

distance: *NX_FLOAT* {units=*NX_LENGTH*}

data[ntimechan]: *NX_INT*

time_of_flight[ntimechan]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

integral_counts: *NX_INT* {units=*NX_UNITLESS*}

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

detector_number –> /NXentry/NXinstrument/NXdetector/detector_number

time_of_flight –> /NXentry/NXinstrument/NXdetector/time_of_flight

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtofraw.nxdl.xml

NXtofsingle

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is a application definition for raw data from a generic TOF instrument

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXuser*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtofsingle`

duration: *NX_FLOAT*

pre_sample_flightpath: *NX_FLOAT* {units=*NX_LENGTH*}

This is the flight path before the sample position. This can be determined by a chopper, by the moderator or the source itself. In other words: it the distance to the component which gives the T0 signal to the detector electronics. If another component in the NXinstrument hierarchy provides this information, this should be a link.

user: *NXuser*

name: *NX_CHAR*

(instrument): *NXinstrument*

detector: *NXdetector*

data[xsize, ysize, ntimechan]: *NX_INT*

distance[1]: *NX_FLOAT* {units=*NX_LENGTH*}

Distance to sample for the center of the detector

time_of_flight[ntimechan]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

polar_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

polar angle for each detector element

azimuthal_angle[ndet]: *NX_FLOAT* {units=*NX_ANGLE*}

azimuthal angle for each detector element

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

nature: *NX_CHAR*

Any of these values: `powder|liquid|single crystal`

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

distance: *NX_FLOAT* {units=*NX_LENGTH*}

data[ntimechan]: *NX_INT*

time_of_flight[ntimechan]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

data: *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

detector_number –> /NXentry/NXinstrument/NXdetector/detector_number

time_of_flight –> /NXentry/NXinstrument/NXdetector/time_of_flight

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtofsingle.nxdl.xml

NXtomo

Status:

application definition, extends *NXobject*, version 2.0

Description:

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.

Symbols:

These symbols will be used below to coordinate datasets with the same shape.

nFrames: number of frames

xsize: number of pixels in X direction

ysize: number of pixels in Y direction

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: (optional) *NX_CHAR*

start_time: (optional) *NX_DATE_TIME*

end_time: (optional) *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtomo`

instrument: *NXinstrument*

(source): (optional) *NXsource*

type: (optional) *NX_CHAR*

name: (optional) *NX_CHAR*

probe: (optional) *NX_CHAR*

Any of these values: `neutron` | `x-ray` | `electron`

detector: *NXdetector*

data[nFrames, xsize, ysize]: *NX_INT*

image_key[nFrames]: *NX_INT*

In order to distinguish between sample projectrions, dark and flat images, a magic number is recorded per frame. The key is as follows:

- projection = 0
- flat field = 1
- dark field = 2
- invalid = 3

x_pixel_size: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_size: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

Distance between detector and sample

x_rotation_axis_pixel_position: (optional) *NX_FLOAT*

y_rotation_axis_pixel_position: (optional) *NX_FLOAT*

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

rotation_angle[nFrames]: *NX_FLOAT* {units=*NX_ANGLE*}

In practice this axis is always aligned along one pixel direction on the detector and usually vertical. There are experiments with horizontal rotation axes, so this would need to be indicated somehow. For now the best way for that is an open question.

x_translation[nFrames]: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

y_translation[nFrames]: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

z_translation[nFrames]: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

control: (optional) *NXmonitor*

data[nFrames]: *NX_FLOAT* {units=*NX_ANY*}

Total integral monitor counts for each measured frame. Allows a to correction for fluctuations in the beam between frames.

data: *NXdata*

data –> /NXentry/NXinstrument/detector:NXdetector/data

rotation_angle –> /NXentry/NXsample/rotation_angle

image_key –> /NXentry/NXinstrument/detector:NXdetector/image_key

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtomo.nxdl.xml

NXtomophase

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is the application definition for x-ray or neutron tomography raw data with phase contrast variation at each point.

In tomography first some dark field images are measured, some bright field images and, of course the sample. In order to properly sort the order of the images taken, a sequence number is stored with each image.

Symbols:

These symbols will be used below to coordinate datasets with the same shape.

nBrightFrames: number of bright frames

nDarkFrames: number of dark frames

nSampleFrames: number of image (sample) frames

nPhase: number of phase settings

xsize: number of pixels in X direction

ysize: number of pixels in Y direction

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

end_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtomophase`

instrument: *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron|x-ray|electron`

bright_field: *NXdetector*

data[nBrightFrames, xsize, ysize]: *NX_INT*

sequence_number[nBrightFrames]: *NX_INT*

dark_field: *NXdetector*

data[nDarkFrames, xsize, ysize]: *NX_INT*

sequence_number[nDarkFrames]: *NX_INT*

sample: *NXdetector*

data[nSampleFrames, nPhase, xsize, ysize]: *NX_INT*

sequence_number[nSampleFrames, nPhase]: *NX_INT*

x_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Distance between detector and sample

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

rotation_angle[nSampleFrames]: *NX_FLOAT* {units=*NX_ANGLE*}

x_translation[nSampleFrames]: *NX_FLOAT* {units=*NX_LENGTH*}

y_translation[nSampleFrames]: *NX_FLOAT* {units=*NX_LENGTH*}

z_translation[nSampleFrames]: *NX_FLOAT* {units=*NX_LENGTH*}

control: *NXmonitor*

integral[nDarkFrames + nBrightFrames + nSampleFrame]: *NX_FLOAT* {units=*NX_ANY*}

Total integral monitor counts for each measured frame. Allows a correction for fluctuations in the beam between frames.

data: *NXdata*

data –> /NXentry/NXinstrument/sample:NXdetector/data

rotation_angle –> /NXentry/NXsample/rotation_angle

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtomophase.nxdl.xml

NXtomoproc

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This is an application definition for the final result of a tomography experiment: a 3D construction of some volume of physical properties.

Symbols:

These symbols will be used below to coordinate datasets with the same shape.

nx: number of voxels in X direction

ny: number of voxels in Y direction

nz: number of voxels in Z direction

Groups cited: *NXdata*, *NXentry*, *NXinstrument*, *NXparameters*, *NXprocess*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXtomoproc`

(instrument): *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron|x-ray|electron`

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

reconstruction: *NXprocess*

program: *NX_CHAR*

Name of the program used for reconstruction

version: *NX_CHAR*

Version of the program used

date: *NX_DATE_TIME*

Date and time of reconstruction processing.

parameters: *NXparameters*

raw_file: *NX_CHAR*

Original raw data file this data was derived from

data: *NXdata*

data[nx, nx, nz]: *NX_INT*

This is the reconstructed volume. This can be different things. Please indicate in the unit attribute what physical quantity this really is.

@transform: *NX_CHAR*

@offset: *NX_CHAR*

@scaling: *NX_CHAR*

x[nx]: *NX_FLOAT* {units=*NX_ANY*}

This is an array holding the values to use for the x-axis of data. The units must be appropriate for the measurement.

y[ny]: *NX_FLOAT* {units=*NX_ANY*}

This is an array holding the values to use for the y-axis of data. The units must be appropriate for the measurement.

z[nz]: *NX_FLOAT* {units=*NX_ANY*}

This is an array holding the values to use for the z-axis of data. The units must be appropriate for the measurement.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXtomoproc.nxdl.xml

NXxas

Status:

application definition, extends *NXobject*, version 1.0

Description:

This is an application definition for raw data from an X-ray absorption spectroscopy experiment.

This is essentially a scan on energy versus incoming/ absorbed beam.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXmonochromator*, *NXsample*, *NXsource*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

NeXus convention is to use "entry1", "entry2", ... for analysis software to locate each entry.

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxas`

(instrument): *NXinstrument*

(source): *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Obligatory value: `x-ray`

monochromator: *NXmonochromator*

energy[np]: *NX_FLOAT*

incoming_beam: *NXdetector*

data[np]: *NX_INT*

absorbed_beam: *NXdetector*

data[np]: *NX_INT*

mark this field with attribute `signal=1`

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

(monitor): *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

data[np]: *NX_INT*

(data): *NXdata*

energy –> /entry/instrument/monochromator/energy

absorbed_beam –> /entry/instrument/absorbed_beam/data

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxas.nxdl.xml

NXxasproc

Status:

application definition, extends *NXobject*, version 1.0

Description:

Processed data from XAS. This is energy versus I(incoming)/I(absorbed).

Symbols:

No symbol table

Groups cited: *NXdata*, *NXentry*, *NXparameters*, *NXprocess*, *NXsample*

Structure:

(entry): *NXentry*

@entry: *NX_CHAR*

NeXus convention is to use “entry1”, “entry2”, ... for analysis software to locate each entry.

title: *NX_CHAR*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxasproc`

(sample): *NXsample*

name: *NX_CHAR*

Descriptive name of sample

XAS_data_reduction: *NXprocess*

program: *NX_CHAR*

Name of the program used for reconstruction

version: *NX_CHAR*

Version of the program used

date: *NX_DATE_TIME*

Date and time of reconstruction processing.

parameters: *NXparameters*

raw_file: *NX_CHAR*

Original raw data file this data was derived from

(data): *NXdata*

energy[np]: *NX_CHAR*

data[np]: *NX_FLOAT*

This is corrected and calibrated I(incoming)/I(absorbed). So it is the absorption. Expect attribute `signal=1`

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxasproc.nxdl.xml

NXxbase

Status:

application definition, extends *NXobject*, version 1.0b

Description:

This definition covers the common parts of all monochromatic single crystal raw data application definitions.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXmonochromator*, *NXsample*, *NXsource*

Structure:

entry: *NXentry*

title: *NX_CHAR*

start_time: *NX_DATE_TIME*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxbase`

instrument: *NXinstrument*

source: *NXsource*

type: *NX_CHAR*

name: *NX_CHAR*

probe: *NX_CHAR*

Any of these values: `neutron|x-ray|electron`

monochromator: *NXmonochromator*

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

detector: *NXdetector*

The name of the group is detector if there is only one detector, if there are several, names have to be detector1, detector2, ...detectorn.

data[np, number of x pixels, number of y pixels]: *NX_INT*

The area detector data, the first dimension is always the number of scan points, the second and third are the number of pixels in x and y. The origin is always assumed to be in the center of the detector. maxOccurs is limited to the the number of detectors on your instrument.

@signal: *NX_POSINT*

Obligatory value: `1`

x_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_size: *NX_FLOAT* {units=*NX_LENGTH*}

distance: *NX_FLOAT* {units=*NX_LENGTH*}

frame_start_number: *NX_INT*

This is the start number of the first frame of a scan. In PX one often scans a couple of frames on a give sample, then does something else, then returns to the same sample and scans some more frames. Each time with a new data file. This number helps concatenating such measurements.

sample: *NXsample*

name: *NX_CHAR*

Descriptive name of sample

orientation_matrix[3, 3]: *NX_FLOAT*

The orientation matrix according to Busing and Levy conventions. This is not strictly necessary as the UB can always be derived from the data. But let us bow to common usage which includes the UB nearly always.

unit_cell[6]: *NX_FLOAT*

The unit cell, a, b, c, alpha, beta, gamma. Again, not strictly necessary, but normally written.

temperature[NP]: *NX_FLOAT*

The sample temperature or whatever sensor represents this value best

x_translation: *NX_FLOAT* {units=*NX_LENGTH*}

Translation of the sample along the X-direction of the laboratory coordinate system

y_translation: *NX_FLOAT* {units=*NX_LENGTH*}

Translation of the sample along the Y-direction of the laboratory coordinate system

distance: *NX_FLOAT* {units=*NX_LENGTH*}

Translation of the sample along the Z-direction of the laboratory coordinate system

control: *NXmonitor*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_FLOAT*

preset value for time or monitor

integral: *NX_FLOAT* {units=*NX_ANY*}

Total integral monitor counts

(data): *NXdata*

The name of this group id data if there is only one detector; if there are several the names will be data1, data2, data3 and will point to the corresponding detector groups in the instrument hierarchy.

data –> /NXentry/NXinstrument/NXdetector/data

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxbase.nxdl.xml

NXxeuler

Status:

application definition, extends *NXxbase*, version 1.0b

Description:

raw data from a four-circle diffractometer with an eulerian cradle, extends *NXxbase*

It extends *NXxbase*, so the full definition is the content of *NXxbase* plus the data defined here. All four angles are logged in order to support arbitrary scans in reciprocal space.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXsample*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxeuler`

instrument: *NXinstrument*

detector: *NXdetector*

polar_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

The polar_angle (two theta) where the detector is placed at each scan point.

sample: *NXsample*

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the sample rotation angle at each scan point

chi[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the chi angle of the eulerian cradle at each scan point

phi[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the phi rotation of the eulerian cradle at each scan point

name: *NXdata*

polar_angle –> /NXentry/NXinstrument/NXdetector/polar_angle

rotation_angle –> /NXentry/NXsample/rotation_angle

chi –> /NXentry/NXsample/chi

phi –> /NXentry/NXsample/phi

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxeuler.nxdl.xml

NXxkappa

Status:

application definition, extends *NXxbase*, version 1.0b

Description:

raw data from a kappa geometry (CAD4) single crystal diffractometer, extends *NXxbase*

This is the application definition for raw data from a kappa geometry (CAD4) single crystal diffractometer. It extends *NXxbase*, so the full definition is the content of *NXxbase* plus the data defined here.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXsample*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxkappa`

instrument: *NXinstrument*

detector: *NXdetector*

polar_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

The polar_angle (two theta) at each scan point

sample: *NXsample*

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the sample rotation angle at each scan point

kappa[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the kappa angle at each scan point

phi[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the phi angle at each scan point

alpha: *NX_FLOAT* {units=*NX_ANGLE*}

This holds the inclination angle of the kappa arm.

name: *NXdata*

polar_angle –> /NXentry/NXinstrument/NXdetector/polar_angle

rotation_angle –> /NXentry/NXsample/rotation_angle

kappa –> /NXentry/NXsample/kappa

phi –> /NXentry/NXsample/phi

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxkappa.nxdl.xml

NXxlaue

Status:

application definition, extends *NXxrot*, version 1.0b

Description:

raw data from a single crystal laue camera, extends *NXxrot*

This is the application definition for raw data from a single crystal laue camera. It extends *NXxrot*.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXentry*, *NXinstrument*, *NXsource*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxlaue`

instrument: *NXinstrument*

source: *NXsource*

distribution: *NXdata*

This is the wavelength distribution of the beam

data[ne]: *NX_CHAR*

expect `signal=1 axes="energy"`

wavelength[ne]: *NX_CHAR* {units=*NX_WAVELENGTH*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxlaue.nxdl.xml

NXxlaueplate

Status:

application definition, extends *NXxlaue*, version 1.0b

Description:

raw data from a single crystal Laue camera, extends *NXxlaue*

This is the application definition for raw data from a single crystal Laue camera with an image plate as a detector. It extends *NXxlaue*.

Symbols:

No symbol table

Groups cited: *NXdetector*, *NXentry*, *NXinstrument*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxlaueplate`

instrument: *NXinstrument*

detector: *NXdetector*

diameter: *NX_FLOAT* {units=*NX_LENGTH*}

The diameter of a cylindrical detector

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxlaueplate.nxdl.xml

NXxnb

Status:

application definition, extends *NXxbase*, version 1.0b

Description:

raw data from a single crystal diffractometer, extends *NXxbase*

This is the application definition for raw data from a single crystal diffractometer measuring in normal beam mode. It extends *NXxbase*, so the full definition is the content of *NXxbase* plus the data defined here. All angles are logged in order to support arbitrary scans in reciprocal space.

Symbols:

No symbol table

Groups cited: *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXsample*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms

Obligatory value: `NXxnb`

instrument: *NXinstrument*

detector: *NXdetector*

polar_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

The polar_angle (gamma) of the detector for each scan point.

tilt_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

The angle by which the detector has been tilted out of the scattering plane.

sample: *NXsample*

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the sample rotation angle at each scan point

name: *NXdata*

polar_angle –> /NXentry/NXinstrument/NXdetector/polar_angle

tilt –> /NXentry/NXinstrument/NXdetector/tilt

rotation_angle –> /NXentry/NXsample/rotation_angle

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxnb.nxdl.xml

NXxrot

Status:

application definition, extends *NXxbase*, version 1.0b

Description:

raw data from a rotation camera, extends *NXxbase*

This is the application definition for raw data from a rotation camera. It extends *NXxbase*, so the full definition is the content of *NXxbase* plus the data defined here.

Symbols:

No symbol table

Groups cited: *NXattenuator*, *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXsample*

Structure:

entry: *NXentry*

definition: *NX_CHAR*

Official NeXus NXDL schema to which this file conforms.

Obligatory value: `NXxrot`

instrument: *NXinstrument*

detector: *NXdetector*

polar_angle: *NX_FLOAT* {units=*NX_ANGLE*}

The polar_angle (two theta) where the detector is placed.

beam_center_x: *NX_FLOAT* {units=*NX_LENGTH*}

This is the x position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

beam_center_y: *NX_FLOAT* {units=*NX_LENGTH*}

This is the y position where the direct beam would hit the detector. This is a length, not a pixel position, and can be outside of the actual detector.

attenuator: *NXattenuator*

attenuator_transmission: *NX_FLOAT* {units=*NX_ANY*}

sample: *NXsample*

rotation_angle[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the sample rotation start angle at each scan point

rotation_angle_step[np]: *NX_FLOAT* {units=*NX_ANGLE*}

This is an array holding the step made for sample rotation angle at each scan point

name: *NXdata*

rotation_angle –> /NXentry/NXsample/rotation_angle

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXxrot.nxdl.xml

3.3.3 Contributed Definitions

A description of each NeXus contributed definition is given. 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. The *NIAC: The NeXus International Advisory Committee* is charged to review any new contributed definitions and provide feedback to the authors before ratification and acceptance as either a base class or application definition.

NXcanSAS Implementation of the canSAS standard to store reduced small-angle scattering data of any dimension.

NXcontainer State of a container holding the sample under investigation.

NXelectrostatic_kicker definition for a electrostatic kicker.

NXmagnetic_kicker definition for a magnetic kicker.

NXquadrupole_magnet definition for a quadrupole magnet.

NXreflections This is a definition for reflection data from diffraction experiments

NXseparator definition for an electrostatic separator.

NXsnsevent This is a definition for event data from Spallation Neutron Source (SNS) at ORNL.

NXsnshisto This is a definition for histogram data from Spallation Neutron Source (SNS) at ORNL.

NXsolenoid_magnet definition for a solenoid magnet.

NXspecdata Data collected by SPEC control and data acquisition software

NXspin_rotator definition for a spin rotator.

NXcanSAS

Status:

application definition, extends *NXobject*, version 1.0

Description:

Implementation of the canSAS standard to store reduced small-angle scattering data of any dimension.

For more details, see:

- http://www.cansas.org/
- http://www.cansas.org/formats/canSAS1d/1.1/doc/
- https://github.com/canSAS-org/NXcanSAS_examples

The minimum requirements for *reduced* small-angle scattering data as described by canSAS are summarized in the following figure:

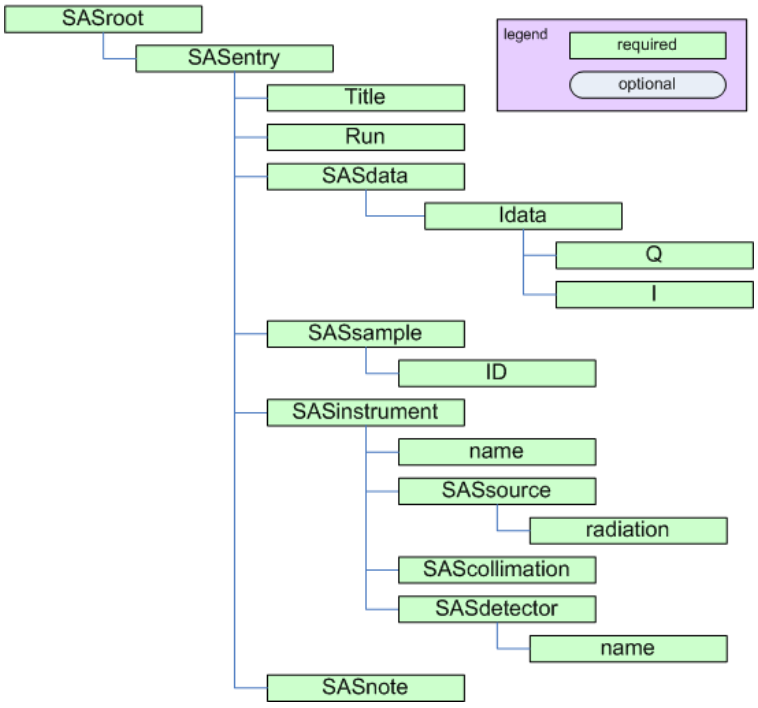


Fig. 3.10: The minimum requirements for *reduced* small-angle scattering data. (`full image`)

Implementation of canSAS standard in NeXus

This application definition is an implementation of the canSAS standard for storing both one-dimensional and multi-dimensional data.

The canSAS data format has a structure similar to NeXus, not identical. To allow canSAS data to be expressed in NeXus, yet identifiable by the canSAS standard, an additional group attribute `canSAS_class` was introduced. Here is the mapping of some common groups.

group (*)	NX_class	canSAS_class
sasentry	NXentry	SASentry
sasdata	NXdata	SASdata
sasdetector	NXdetector	SASdetector
sasinstrument	NXinstrument	SASinstrument
sasnote	NXnote	SASnote
sasprocess	NXprocess	SASprocess
sasprocessnote	NXcollection	SASprocessnote
sastransmission	NXdata	SAStransmission_spectrum
sassample	NXsample	SASsample
sassource	NXsource	SASsource

(*) The name of each group is a suggestion, not a fixed requirement and is chosen as fits each data file. See the section on defining *NXDL group and field names*.

In canSAS1d, *orientation* (rotations of sample or detector) are described in terms of *roll*, *pitch*, and *yaw*. NeXus uses different terms as shown in the next table:

canSAS1d	NeXus
roll	`polar_angle`
pitch	`x_axis_rotation` (not expected or defined here)
yaw	`azimuthal_angle`

Symbols:

No symbol table

Groups cited: *NXaperture*, *NXcollection*, *NXcollimator*, *NXdata*, *NXdetector*, *NXentry*, *NXinstrument*, *NXnote*, *NXprocess*, *NXsample*, *NXsource*

Structure:

(entry): *NXentry*

Place the canSAS `SASentry` group as a child of a NeXus `NXentry` group (when data from multiple techniques are being stored) or as a replacement for the `NXentry` group.

Note: It is required for all numerical objects to provide a *units* attribute that describes the engineering units. Use the Unidata UDunits [1] specification as this is compatible with various community standards.

@default: *NX_CHAR*

Declares which *NXdata* group contains the data to be shown by default. It is needed to resolve ambiguity when more than one *NXdata* group exists. The value is the name of the default *NXdata* group.

@canSAS_class: *NX_CHAR*

Official canSAS group: **SASentry**

[1] The UDunits specification also includes instructions for derived units.

Obligatory value: `SASentry`

@version: *NX_CHAR*

Describes the version of the canSAS standard used to write this data. This must be a text (not numerical) representation. Such as:

```
@version="1.0"
```

Obligatory value: `1.0`

definition: *NX_CHAR*

Official NeXus NXDL schema to which this subentry conforms.

Obligatory value: `NXcanSAS`

title: *NX_CHAR*

Title of this *SASentry*.

run: *NX_CHAR*

Run identification for this *SASentry*. For many facilities, this is an integer. Use multiple instances of `run` as needed, keeping in mind that HDF5 requires unique names for all entities in a group.

@name: *NX_CHAR*

Optional string attribute to identify this particular *run*. Could use this to associate (correlate) multiple *SASdata* elements with *run* elements.

(data): *NXdata*

A *SASData* group contains reduced a single small-angle scattering data set that can be represented as $I(\vec{Q})$ or $I(|\vec{Q}|)$.

Q can be either a vector ($\vec{Q}$) or a vector magnitude ($|\vec{Q}|$)

The name of each *SASdata* must be unique within a SASentry group. Such as `sasdata01`.

A *SASdata* group has several attributes:

- I_axes
- Q_indices
- Mask_indices

To indicate the dependency relationships of other varied parameters, use attributes similar to `@Mask_indices` (such as `@Temperature_indices` or `@Pressure_indices`).

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASdata

Obligatory value: `SASdata`

@signal: *NX_CHAR*

Name of the default data field.

Obligatory value:

- `I`: For canSAS **SASdata**, this is always “I”.

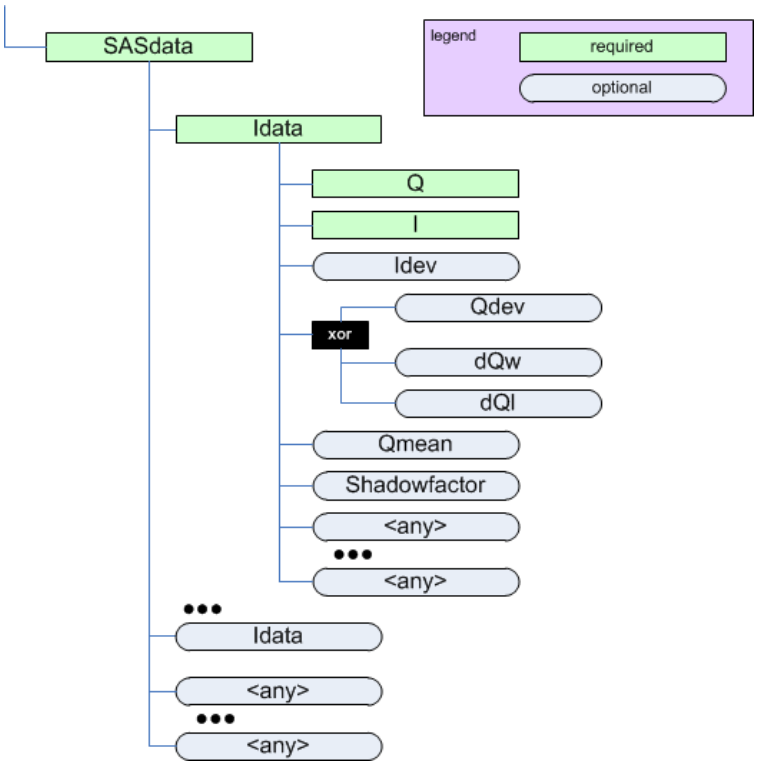


Fig. 3.11: The *SASdata* element (`full image`)

@I_axes: *NX_CHAR*

String array that defines the independent data fields used in the default plot for all of the dimensions of the *signal* field (the *signal* field is the field in this group that is named by the `signal` attribute of this group). One entry is provided for every dimension of the `I` data object. Such as:

```
@I_axes="Temperature", "Time", "Pressure", "Q", "Q"
```

Since there are five items in the list, the intensity field of this example `I` must be a five-dimensional array (rank=5).

@I_uncertainties: *NX_CHAR*

Generally, this is the estimate of the uncertainty of each I. Typically the estimated standard deviation. For Poisson statistics, use $1/\sqrt{I}$.

(optional for numerical arrays) Name of the data object (in this SASdata group) that provides the uncertainty to be used for data analysis.

Idev is the canonical name from the 1D standard. The multi-D standard allows for this name to be described in this attribute. Such as:

```
@I_uncertainties="Idev"
```

@Q_indices: *NX_INT*

Integer or integer array that describes which indices (of the I data object) are used to reference the `Q` data object. The items in this array use zero-based indexing. Such as:

```
@Q_indices=1,3,4
```

which indicates that `Q` requires three indices from the I data object: one for time and two for Q position. Thus, in this example, the `Q` data is time-dependent: $\vec{Q}(t)$.

@Q_uncertainties: *NX_CHAR*

(optional for numerical arrays) Generally, this is the estimate of the uncertainty of each Q. Typically the estimated standard deviation. Names the data object (in this SASdata group) that provides the uncertainty to be used for data analysis. Such as:

```
@Q_uncertainties="Qdev"
```

Can use this to describe the slit-length at each datum. Use a subgroup to describe any supplementary uncertainty data.

To specify two-dimensional uncertainty, such as (*dQw*, *dQl*), use a string array, such as:

```
@Q_uncertainties="dQw", "dQl"
```

@Mask_indices: *NX_CHAR*

Integer or integer array that describes which indices (of the I data object) are used to reference the `Mask` data object. The items in this array use zero-based indexing. Such as:

```
@Mask_indices=3,4
```

which indicates that Q requires two indices from the I data object for Q position.

Q: *NX_NUMBER* {units=*NX_PER_LENGTH*}

Array of Q data to accompany I.

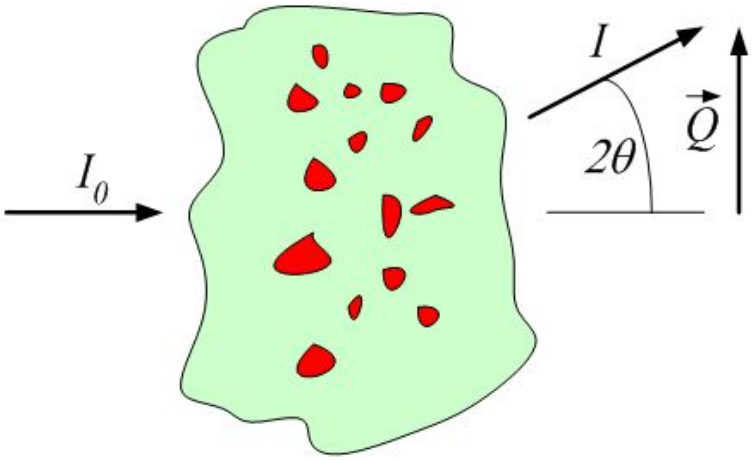


Fig. 3.12: The $\vec{Q}$ geometry. (`full image`)

Q may be represented either as the three-dimensional scattering vector $\vec{Q}$ or

by the magnitude of the scattering vector, $|\vec{Q}|$.

$$|\vec{Q}| = (4\pi/\lambda)sin(\theta) \tag{3.1}$$

When we write Q, we may refer to either or both of $|\vec{Q}|$ or $\vec{Q}$, depending on the context.

I: *NX_NUMBER*

Array of intensity (I) data.

The intensity may be represented in one of these forms:

absolute units: $d\Sigma/d\Omega(Q)$ differential cross-section per unit volume per unit solid angle (typical units: 1/cm/sr)

absolute units: $d\sigma/d\Omega(Q)$ differential cross-section per unit atom per unit solid angle (typical units: cm^2)

arbitrary units: $I(Q)$ usually a ratio of two detectors but units are meaningless (typical units: a.u.)

This presents a few problems for analysis software to sort out when reading the data. Fortunately, it is possible to analyze the *units* to determine which type of intensity is being reported and make choices at the time the file is read. But this is an area for consideration and possible improvement.

One problem arises with software that automatically converts data into some canonical units used by that software. The software should not convert units between these different types of intensity indiscriminately.

A second problem is that when arbitrary units are used, then the set of possible analytical results is restricted. With such units, no meaningful volume fraction or number density can be determined directly from $I(Q)$.

In some cases, it is possible to apply a factor to convert the arbitrary units to an absolute scale. This should be considered as a possibility of the analysis process.

Idev: (optional) *NX_NUMBER* {units=*NX_PER_LENGTH*}

Estimated uncertainty (usually standard deviation) in I. Must have the same units as I.

When present, the name of this field is also recorded in the *uncertainties* attribute of *I*, as in:

```
I/@uncertainties="Idev"
```

Qdev: (optional) *NX_NUMBER* {units=*NX_PER_LENGTH*}

Estimated uncertainty (usually standard deviation) in Q. Must have the same units as Q.

When present, the name of this field is also recorded in the *uncertainties* attribute of *Q*, as in:

```
Q/@uncertainties="Qdev"

Q/@uncertainties="dQw", "dQl"
```

dQw: (optional) *NX_NUMBER* {units=*NX_PER_LENGTH*}

Q resolution along the axis of scanning (the high-resolution *slit width* direction). Useful for defining resolution data from slit-smearing instruments such as Bonse-Hart geometry. Must have the same units as Q.

When present, the name of this field is also recorded in the *uncertainties* attribute of Q, as in:

```
Q/@uncertainties="dQw", "dQl"
```

dQl: (optional) *NX_NUMBER* {units=*NX_PER_LENGTH*}

Q resolution perpendicular to the axis of scanning (the low-resolution *slit length* direction). Useful for defining resolution data from slit-smearing instruments such as Bonse-Hart geometry. Must have the same units as Q.

When present, the name of this field is also recorded in the *uncertainties* attribute of Q, as in:

```
Q/@uncertainties="dQw", "dQl"
```

Qmean: (optional) *NX_NUMBER* {units=*NX_PER_LENGTH*}

Mean value of Q for this data point. Useful when describing data that has been binned from higher-resolution data. It is unexpected for `Q` and `Qmean` to have different units.

ShadowFactor: (optional) *NX_CHAR* {units=*NX_DIMENSIONLESS*}

A numerical factor applied to pixels affected by the beam stop penumbra. Used in data files from NIST/NCNR instruments.

See: J.G. Barker and J.S. Pedersen (1995) *J. Appl. Cryst.* **28**, 105-114.

(instrument): *NXinstrument*

This the SAS instrument.

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASinstrument

Obligatory value: `SASinstrument`

(collimator): (optional) *NXcollimator*

Description of a collimating element in the instrument.

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SAScollimation

Obligatory value: `SAScollimation`

length: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

Amount/length of collimation inserted (as on a SANS instrument)

distance: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

Distance from this collimation element to the sample

aperture: (optional) *NXaperture*

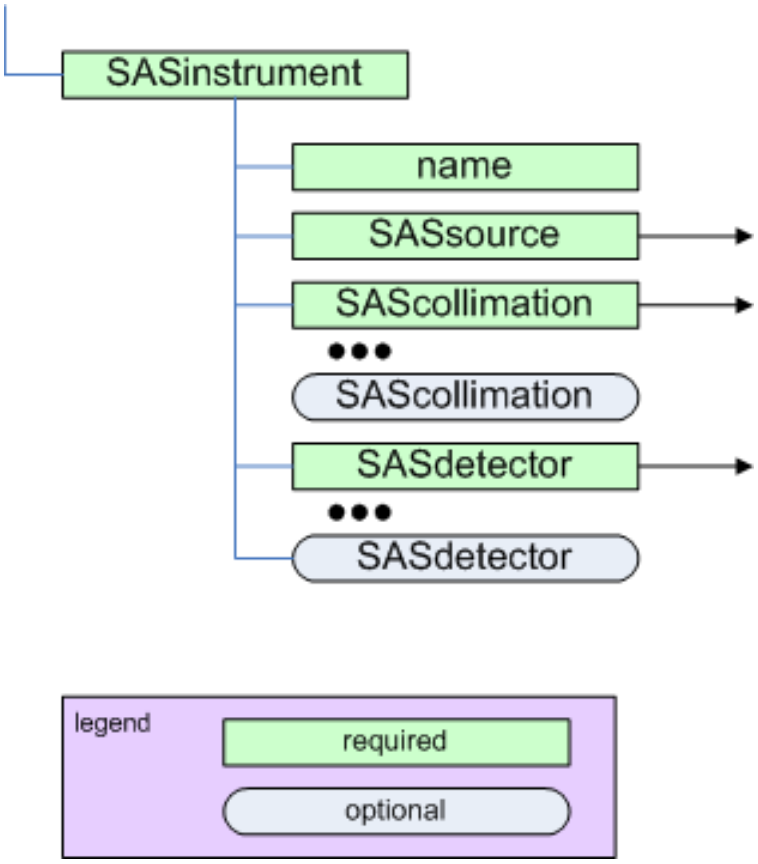


Fig. 3.13: The *SASinstrument* element (`full image`)

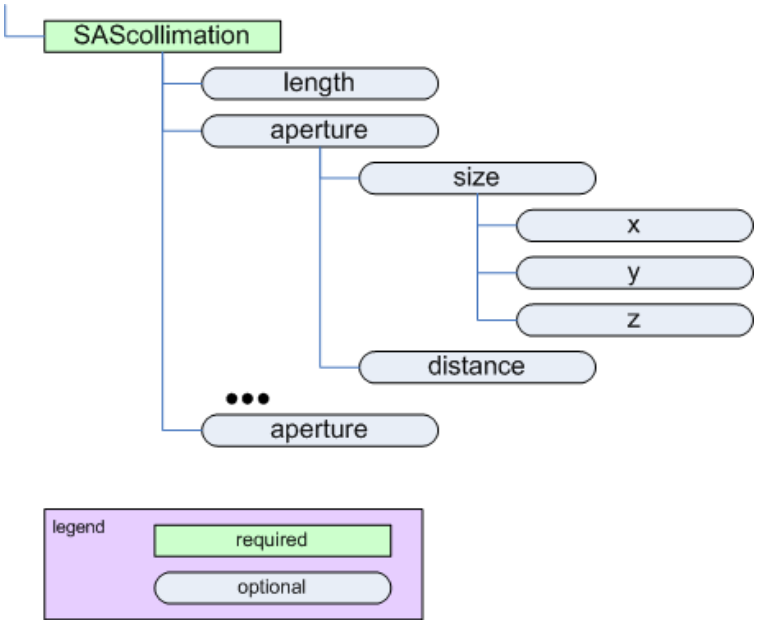


Fig. 3.14: The *SAScollimation* element (`full image`)

Name of "aperture" is only a suggestion. Base class could be either **NXpinhole** or **NXslit**. But **NXaperture** is generic and limits the variation in data files.

shape: *NX_CHAR*

describe the type of aperture (pinhole, 4-blade slit, Soller slit, ...)

x_gap: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

opening along the x axis

y_gap: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

opening along the y axis

(detector): (optional) *NXdetector*

Description of a detector in the instrument.

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASdetector

Obligatory value: `SASdetector`

name: *NX_CHAR*

Identifies the name of this detector

SDD: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

Distance between sample and detector.

Note: In NXdetector, the `distance` field records the distance to the previous component ... most often the sample. This use is the same as `SDD` for most SAS instruments but not all. For example, Bonse-Hart cameras have one or more crystals between the sample and detector.

We define here the field `SDD` to document without ambiguity the distance between sample and detector.

slit_length: (optional) *NX_NUMBER* {units=*NX_PER_LENGTH*}

Slit length of the instrument for this detector, expressed in the same units as Q.

x_position: (optional) *NX_CHAR*

Location of the detector in x

y_position: (optional) *NX_CHAR*

Location of the detector in y

polar_angle: (optional) *NX_CHAR*

Rotation of the detector about the z axis (roll)

azimuthal_angle: (optional) *NX_CHAR*

Rotation of the detector about the y axis (yaw)

beam_center_x: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

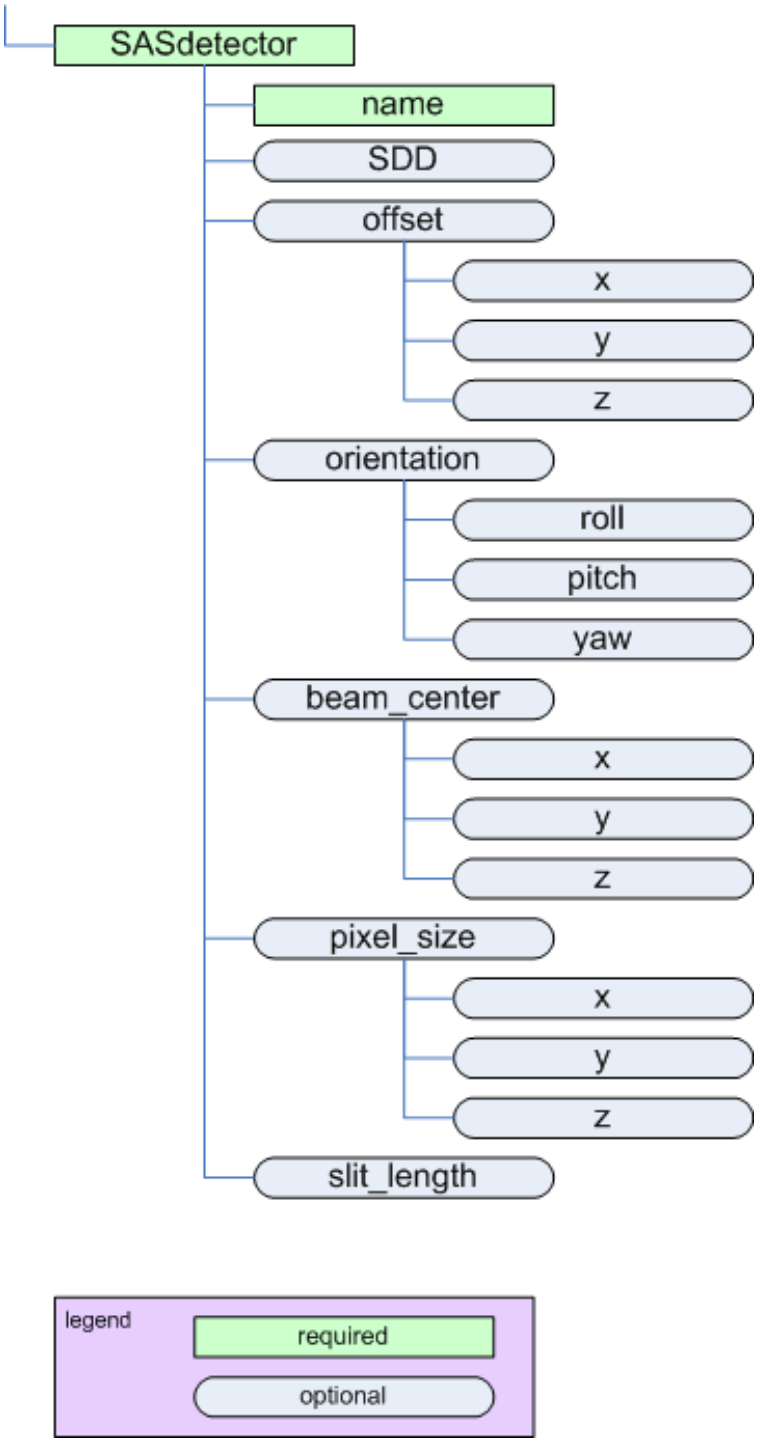


Fig. 3.15: The *SASdetector* element (`full image`)

Position of the beam center on the detector.

This is the x position where the direct beam would hit the detector plane. This is a length, not a pixel position, and can be outside of the actual detector.

beam_center_y: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

Position of the beam center on the detector.

This is the y position where the direct beam would hit the detector plane. This is a length, not a pixel position, and can be outside of the actual detector.

x_pixel_size: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

Size of each detector pixel. If it is scalar all pixels are the same size

y_pixel_size: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

Size of each detector pixel. If it is scalar all pixels are the same size

(source): (optional) *NXsource*

Description of the radiation source.

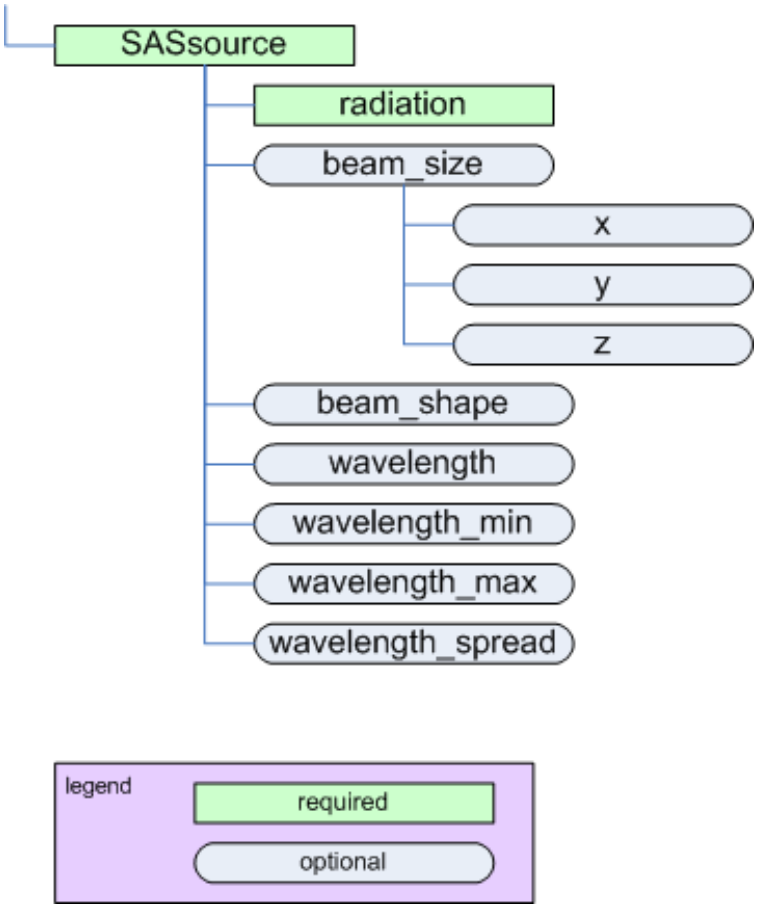


Fig. 3.16: The *SASsource* element (`full image`)

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASsource

Obligatory value: `SASsource`

radiation: *NX_CHAR*

Name of the radiation used. Note that this is **not** the name of the facility!

Any of these values:

- `Spallation Neutron Source`
- `Pulsed Reactor Neutron Source`
- `Reactor Neutron Source`
- `Synchrotron X-ray Source`
- `Pulsed Muon Source`
- `Rotating Anode X-ray`
- `Fixed Tube X-ray`
- `UV Laser`
- `Free-Electron Laser`
- `Optical Laser`
- `Ion Source`
- `UV Plasma Source`
- `neutron`
- `x-ray`
- `muon`
- `electron`
- `ultraviolet`
- `visible light`
- `positron`
- `proton`

beam_shape: (optional) *NX_CHAR*

Text description of the shape of the beam (incident on the sample).

incident_wavelength: (optional) *NX_NUMBER* {units=*NX_WAVELENGTH*}

wavelength (λ) of radiation incident on the sample

wavelength_min: (optional) *NX_NUMBER* {units=*NX_WAVELENGTH*}

Some facilities specify wavelength using a range. This is the lowest wavelength in such a range.

wavelength_max: (optional) *NX_NUMBER* {units=*NX_WAVELENGTH*}

Some facilities specify wavelength using a range. This is the highest wavelength in such a range.

incident_wavelength_spread: (optional) *NX_NUMBER* {units=*NX_WAVELENGTH*}

Some facilities specify wavelength using a range. This is the width (FWHM) of such a range.

beam_size_x: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

Size of the incident beam along the x axis.

beam_size_y: (optional) *NX_NUMBER* {units=*NX_LENGTH*}

Size of the incident beam along the y axis.

(sample): *NXsample*

Description of the sample.

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASsample

Obligatory value: `SASsample`

name: *NX_CHAR*

ID: Text string that identifies this sample.

thickness: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

Thickness of this sample

transmission: (optional) *NX_NUMBER* {units=*NX_DIMENSIONLESS*}

Transmission (I/I_0) of this sample. Note that there is no *units* attribute as this number is dimensionless.

temperature: (optional) *NX_NUMBER* {units=*NX_TEMPERATURE*}

Temperature of this sample.

details: (optional) *NX_CHAR*

Any additional sample details.

x_position: (optional) *NX_CHAR*

Location of the sample in x

y_position: (optional) *NX_CHAR*

Location of the sample in y

polar_angle: (optional) *NX_CHAR*

Rotation of the sample about the z axis (roll)

azimuthal_angle: (optional) *NX_CHAR*

Rotation of the sample about the y axis (yaw)

(process): (optional) *NXprocess*

Description of a processing or analysis step.

Add additional fields as needed to describe value(s) of any variable, parameter, or term related to the *SASprocess* step. Be sure to include *units* attributes for all numerical fields.

@canSAS_class: *NX_CHAR*

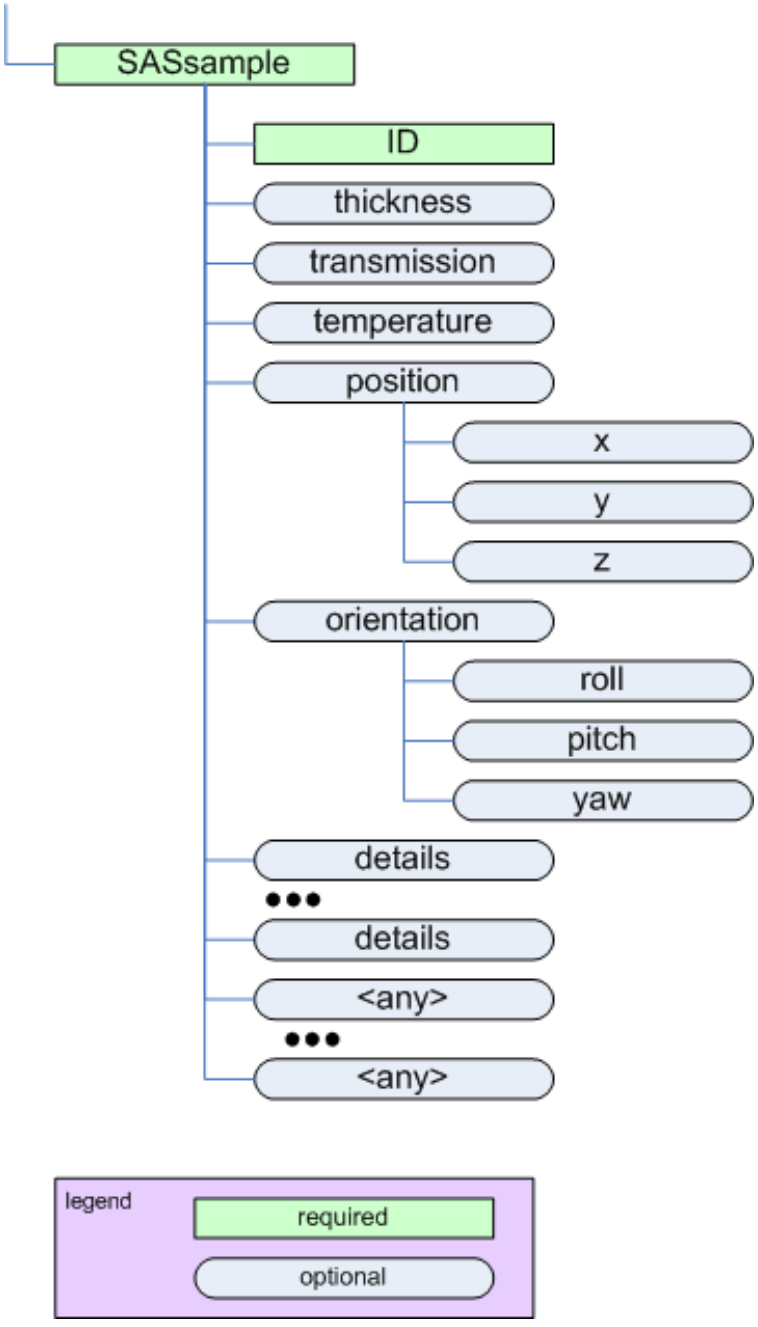


Fig. 3.17: The *SASsample* element (`full image`)

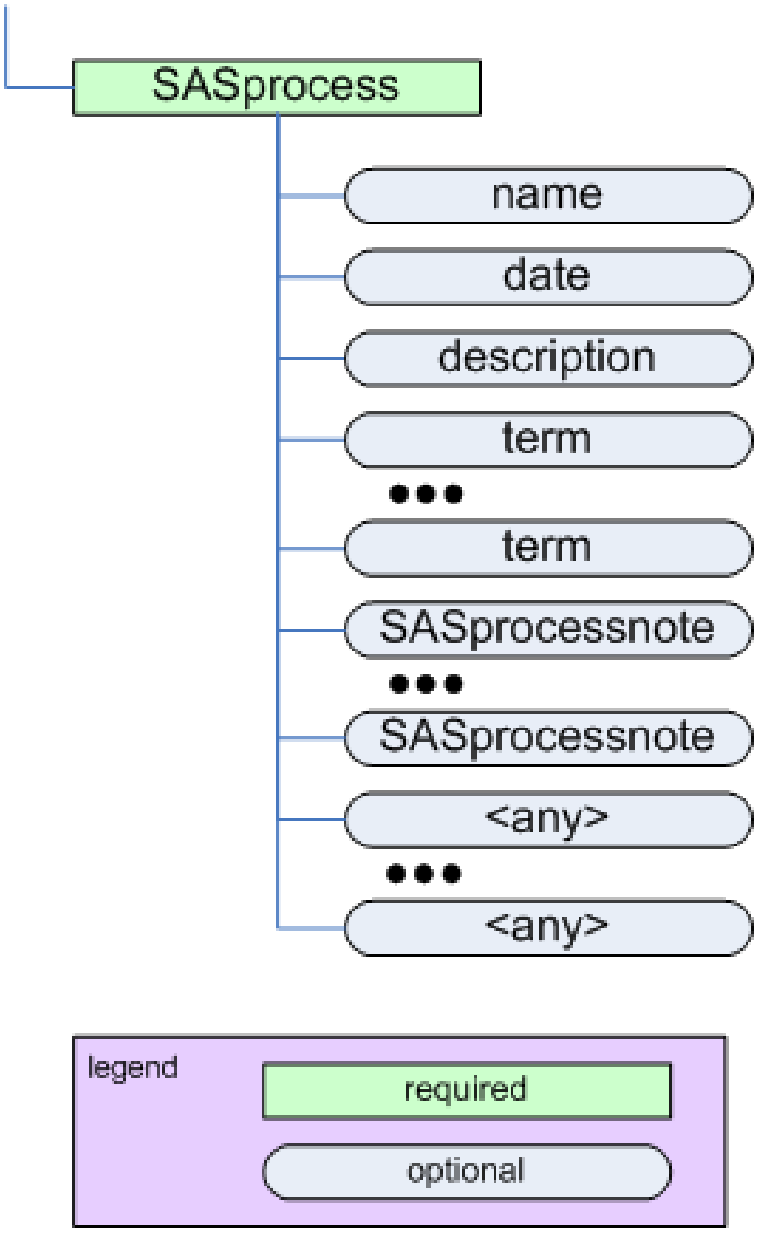


Fig. 3.18: The *SASprocess* element (`full image`)

Official canSAS group: NXcanSAS (contributed definition); SASprocess

Obligatory value: `SASprocess`

name: (optional) *NX_CHAR*

Optional name for this data processing or analysis step

date: (optional) *NX_DATE_TIME*

Optional date for this data processing or analysis step. [2]

description: (optional) *NX_CHAR*

Optional description for this data processing or analysis step

term: (optional) *NX_CHAR*

Specifies the value of a single variable, parameter, or term (while defined here as a string, it could be a number) related to the *SASprocess* step.

Note: The name *term* is not required, it could take any name, as long as the name is unique within this group.

(note): (optional) *NXnote*

Any additional notes or subprocessing steps will be documented here.

An **NXnote** group can be added to any NeXus group at or below the **NXentry** group. It is shown here as a suggestion of a good place to *consider* its use.

sasprocessnote: (optional) *NXcollection*

Describes anything about *SASprocess* that is not already described.

Any content not defined in the canSAS standard can be placed at this point.

Note: The name *sasprocessnote* is not required, it could take any name, as long as the name is unique within the **NXprocess** group.

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASprocessnote

Obligatory value: `SASprocessnote`

(collection): (optional) *NXcollection*

Free form description of anything not covered by other elements.

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SASnote

Obligatory value: `SASnote`

(data): (optional) *NXdata*

The *SAStransmission_spectrum* element

This describes certain data obtained from a variable-wavelength source such as pulsed-neutron source.

[2] ISO-8601 standard time representation.
NeXus dates and times are reported in ISO-8601 (e.g., `yyyy-mm-ddThh:mm:ss`) or modified ISO-8601 (e.g., `yyyy-mm-dd hh:mm:ss`).
See: http://www.w3.org/TR/NOTE-datetime or http://en.wikipedia.org/wiki/ISO_8601 for more details.

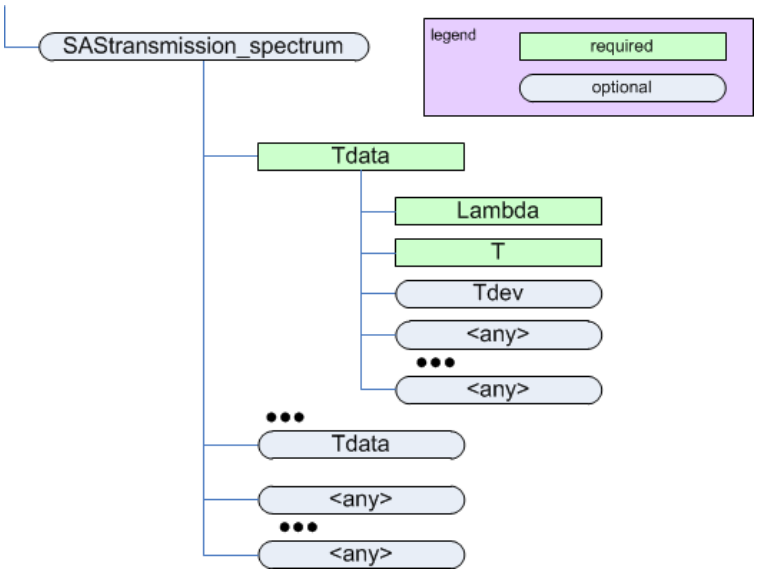


Fig. 3.19: The *SAStransmission_spectrum* element (`full image`)

@canSAS_class: *NX_CHAR*

Official canSAS group: NXcanSAS (contributed definition); SAStransmission_spectrum

Obligatory value: `SAStransmission_spectrum`

@signal: *NX_CHAR*

Name of the default data field.

Obligatory value:

- `T`: For **SAStransmission_spectrum**, this is always "T".

@T_axes: *NX_CHAR*

Obligatory value:

- `T`: the wavelengths field (as a dimension scale) corresponding to this transmission

@T_uncertainties: *NX_CHAR*

Estimate of the uncertainty of each transmission T.

@name: *NX_CHAR*

Identify what type of spectrum is being described. It is expected that this value will take either of these two values:

value	meaning
sample	measurement with the sample and container
can	measurement with just the container

@timestamp: *NX_DATE_TIME*

ISO-8601 time [2]

lambda: *NX_NUMBER* {units=*NX_WAVELENGTH*}

Wavelength of the radiation.

T: *NX_NUMBER* {units=*NX_DIMENSIONLESS*}

Transmission value (I/I_0)

Tdev: *NX_NUMBER* {units=*NX_PER_LENGTH*}

Estimated uncertainty (usually standard deviation) in T. Must have the same units as T.

When present, the name of this field is also recorded in the *uncertainties* attribute of *T*, as in:

```
T/@uncertainties="Tdev"
```

NXDL Source: https://github.com/nexusformat/definitions/blob/master/applications/NXcanSAS.nxdl.xml

NXcontainer

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

State of a container holding the sample under investigation.

A container is any object in the beam path which absorbs the beam and whose contribution to the overall attenuation/scattering needs to be determined to process the experimental data. Examples of containers include glass capillary tubes, vanadium cans, windows in furnaces or diamonds in a Diamond Anvil Cell. The following figures show a complex example of a container:

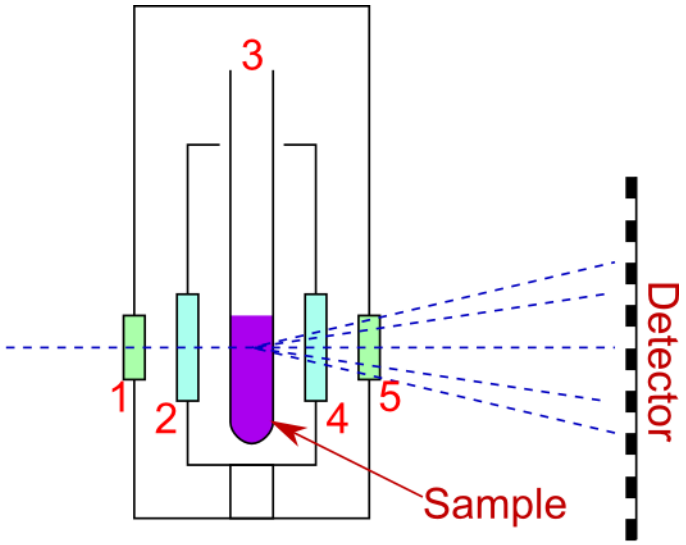


Fig. 3.20: A hypothetical capillary furnace. The beam passes from left to right (blue dashes), passing through window 1, then window 2, before passing through the downstream wall of the capillary. It is then scattered by the sample with scattered beams passing through the upstream wall of the capillary, then windows 4 and 5. As part of the corrections for a PDF experiment it is necessary to subtract the PDF of the empty container (i.e. each of the windows and the capillary). To calculate the PDF of the empty container it is necessary to have the measured scattering data and to know the nature (e.g. density, elemental composition, etc.) of the portion of the container which the beam passed through.

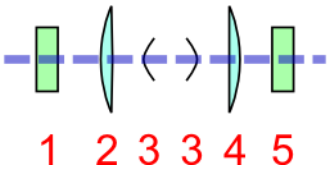


Fig. 3.21: A complete description of the shapes of the container elements with their orientation relative to the beam and also information on whether they are upstream or downstream of the sample is also therefore important. For example, although the windows 2 and 4 have the same shape, the path taken through them by the beam is very different and this needs to be modelled. Furthermore, it is not inconceivable that windows might move during an experiment and thus the changes to the beampath would need to be accounted for.

This class encodes the position of the container with respect to the sample and allows the calculation of the beampath through the container. It also includes sufficient data to model beam absorption of the container and a link to a dataset containing a measurement of the container with nothing inside, to allow data corrections (at a specific beam energy/measurement time) to be made.

Symbols:

No symbol table

Groups cited: *NXbeam*, *NXshape*, *NXtransformations*

Structure:

name: *NX_CHAR*

Descriptive name of container.

description: *NX_CHAR*

Verbose description of container and how it fits into the wider experimental set up.

chemical_formula: *NX_CHAR*

Chemical composition of the material the container is made from. Specified using CIF conventions. Abbreviated version of CIF standard:

- Only recognized element symbols may be used.
- Each element symbol is followed by a 'count' number. A count of '1' may be omitted.
- A space or parenthesis must separate each cluster of (element symbol + count).
- Where a group of elements is enclosed in parentheses, the multiplier for the group must follow the closing parentheses. That is, all element and group multipliers are assumed to be printed as subscripted numbers.
- Unless the elements are ordered in a manner that corresponds to their chemical structure, the order of the elements within any group or moiety depends on whether or not carbon is present.
- If carbon is present, the order should be:
  - C, then H, then the other elements in alphabetical order of their symbol.
  - If carbon is not present, the elements are listed purely in alphabetic order of their symbol.
- This is the *Hill* system used by Chemical Abstracts.

density[n_comp]: *NX_FLOAT* {units=*NX_MASS_DENSITY*}

Density of the material the container is made from.

packing_fraction[n_comp]: *NX_FLOAT* {units=*NX_UNITLESS*}

Fraction of the volume of the container occupied by the material forming the container.

relative_molecular_mass[n_comp]: *NX_FLOAT* {units=*NX_MASS*}

Relative molecular mass of container.

beam: *NXbeam*

Details of beam incident on container, including the position relative to the sample (to determine whether the container is upstream or downstream of the sample).

shape: *NXshape*

Shape of the container. In combination with orientation this should allow the beampath through the container to be modelled to allow the adsorption to be calculated.

orientation: *NXtransformations*

The angle the container makes to the beam and how it may change during the experiment.In combination with shape this should allow the beampath through the container to be modelled to allow the adsorption of the container to be calculated.

reference_measurement –> /NXentry

A link to a full data collection which contains the actual measured data for this container within the experimental set up (with no sample or inner container(s)). This data set will also include the wavelength/energy, measurement time and intensity for which these data are valid.

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXcontainer.nxdl.xml

NXelectrostatic_kicker

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

definition for a electrostatic kicker.

Symbols:

No symbol table

Groups cited: *NXlog*

Structure:

description: *NX_CHAR*

extended description of the kicker.

beamline_distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

define position of beamline element relative to production target

timing: (optional) *NX_FLOAT* {units=*NX_TIME*}

kicker timing as defined by `description` attribute

@description: *NX_CHAR*

set_current: (optional) *NX_FLOAT* {units=*NX_CURRENT*}

current set on supply.

set_voltage: (optional) *NX_FLOAT* {units=*NX_VOLTAGE*}

volage set on supply.

read_current: (optional) *NXlog*

current read from supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_voltage: (optional) *NXlog*

voltage read from supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXelectrostatic_kicker.nxdl.xml

NXmagnetic_kicker

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

definition for a magnetic kicker.

Symbols:

No symbol table

Groups cited: *NXlog*

Structure:

description: *NX_CHAR*

extended description of the kicker.

beamline_distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

define position of beamline element relative to production target

timing: (optional) *NX_FLOAT* {units=*NX_TIME*}

kicker timing as defined by `description` attribute

@description: *NX_CHAR*

set_current: (optional) *NX_FLOAT* {units=*NX_CURRENT*}

current set on supply.

set_voltage: (optional) *NX_FLOAT* {units=*NX_VOLTAGE*}

voltage set on supply.

read_current: (optional) *NXlog*

current read from supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_voltage: (optional) *NXlog*

voltage read from supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXmagnetic_kicker.nxdl.xml

NXquadrupole_magnet

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

definition for a quadrupole magnet.

Symbols:

No symbol table

Groups cited: *NXlog*

Structure:

description: *NX_CHAR*

extended description of the magnet.

beamline_distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

define position of beamline element relative to production target

set_current: (optional) *NX_FLOAT* {units=*NX_CURRENT*}

current set on supply.

read_current: (optional) *NXlog*

current read from supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_voltage: (optional) *NXlog*

voltage read from supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXquadrupole_magnet.nxdl.xml

NXreflections

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

This is a definition for reflection data from diffraction experiments

Symbols:

No symbol table

Groups cited: *NXentry*

Structure:

(entry): *NXentry*

definition: *NX_CHAR*

NeXus NXDL schema to which this file conforms

Obligatory value: `NXreflections`

experiments: *NX_CHAR*

The experiments from which the reflection data derives

h: *NX_INT*

The h component of the miller index

@description: *NX_CHAR*

Describes the dataset

k: *NX_INT*

The k component of the miller index

@description: *NX_CHAR*

Describes the dataset

l: *NX_INT*

The l component of the miller index

@description: *NX_CHAR*

Describes the dataset

id: *NX_INT*

The id of the experiment which resulted in the reflection. If the value is greater than 0, the experiments must link to a multi-experiment NXmx group

@description: *NX_CHAR*

Describes the dataset

reflection_id: *NX_INT*

The id of the reflection. Multiple partials from the same reflection should all have the same id

@description: *NX_CHAR*

Describes the dataset

entering: *NX_BOOLEAN*

Is the reflection entering or exiting the Ewald sphere

@description: *NX_CHAR*

Describes the dataset

det_module: *NX_INT*

The detector module on which the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

flags: *NX_INT*

Status flags describing the reflection

@description: *NX_CHAR*

Describes the dataset

d: *NX_FLOAT*

The resolution of the reflection

@description: *NX_CHAR*

Describes the dataset

partiality: *NX_FLOAT*

The partiality of the reflection

@description: *NX_CHAR*

Describes the dataset

prd_frame: *NX_FLOAT* {units=*NX_UNITLESS*}

The frame on which the bragg peak of the reflection is predicted

@description: *NX_CHAR*

Describes the dataset

prd_mm_x: *NX_FLOAT* {units=*NX_LENGTH*}

The x millimetre position at which the bragg peak of the reflection is predicted

@description: *NX_CHAR*

Describes the dataset

prd_mm_y: *NX_FLOAT* {units=*NX_LENGTH*}

The y millimetre position at which the bragg peak of the reflection is predicted

@description: *NX_CHAR*

Describes the dataset

prd_phi: *NX_FLOAT* {units=*NX_ANGLE*}

The phi angle at which the bragg peak of the reflection is predicted

@description: *NX_CHAR*

Describes the dataset

prd_px_x: *NX_FLOAT* {units=*NX_UNITLESS*}

The x pixel position at which the bragg peak of the reflection is predicted

@description: *NX_CHAR*

Describes the dataset

prd_px_y: *NX_FLOAT* {units=*NX_UNITLESS*}

The y pixel position at which the bragg peak of the reflection is predicted

@description: *NX_CHAR*

Describes the dataset

obs_frame_val: *NX_FLOAT* {units=*NX_UNITLESS*}

The estimate of the frame at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_frame_var: *NX_FLOAT* {units=*NX_UNITLESS*}

The variance on the estimate of the frame at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_px_x_val: *NX_FLOAT* {units=*NX_UNITLESS*}

The estimate of the pixel x position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_px_x_var: *NX_FLOAT* {units=*NX_UNITLESS*}

The variance on the estimate of the pixel x position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_px_y_val: *NX_FLOAT* {units=*NX_UNITLESS*}

The estimate of the pixel y position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_px_y_var: *NX_FLOAT* {units=*NX_UNITLESS*}

The variance on the estimate of the pixel y position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_phi_val: *NX_FLOAT* {units=*NX_ANGLE*}

The estimate of the phi angle at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_phi_var: *NX_FLOAT* {units=*NX_ANGLE*}

The variance on the estimate of the phi angle at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_mm_x_val: *NX_FLOAT* {units=*NX_LENGTH*}

The estimate of the millimetre x position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_mm_x_var: *NX_FLOAT* {units=*NX_LENGTH*}

The variance on the estimate of the millimetre x position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_mm_y_val: *NX_FLOAT* {units=*NX_LENGTH*}

The estimate of the millimetre y position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

obs_mm_y_var: *NX_FLOAT* {units=*NX_LENGTH*}

The variance on the estimate of the millimetre y position at which the central impact of the reflection was recorded

@description: *NX_CHAR*

Describes the dataset

bbx0: *NX_INT* {units=*NX_UNITLESS*}

The lower pixel x position of the bounding box around the recorded reflection

@description: *NX_CHAR*

Describes the dataset

bbx1: *NX_INT* {units=*NX_UNITLESS*}

The upper pixel x position of the bounding box around the recorded reflection

@description: *NX_CHAR*

Describes the dataset

bby0: *NX_INT* {units=*NX_UNITLESS*}

The lower pixel y position of the bounding box around the recorded reflection

@description: *NX_CHAR*

Describes the dataset

bby1: *NX_INT* {units=*NX_UNITLESS*}

The upper pixel y position of the bounding box around the recorded reflection

@description: *NX_CHAR*

Describes the dataset

bbz0: *NX_INT* {units=*NX_UNITLESS*}

The lower frame number of the bounding box around the recorded reflection

@description: *NX_CHAR*

Describes the dataset

bbz1: *NX_INT* {units=*NX_UNITLESS*}

The upper frame number of the bounding box around the recorded reflection

@description: *NX_CHAR*

Describes the dataset

bkg_mean: *NX_FLOAT*

The mean background under the reflection peak

@description: *NX_CHAR*

Describes the dataset

int_prf_val: (optional) *NX_FLOAT*

The estimate of the reflection intensity by profile fitting

@description: *NX_CHAR*

Describes the dataset

int_prf_var: (optional) *NX_FLOAT*

The variance on the estimate of the reflection intensity by profile fitting

@description: *NX_CHAR*

Describes the dataset

int_sum_val: *NX_FLOAT*

The estimate of the reflection intensity by summation

@description: *NX_CHAR*

Describes the dataset

int_sum_var: *NX_FLOAT*

The variance on the estimate of the reflection intensity by summation

@description: *NX_CHAR*

Describes the dataset

lp: *NX_FLOAT*

The LP correction factor to be applied to the reflection intensities

@description: *NX_CHAR*

Describes the dataset

prf_cc: (optional) *NX_FLOAT*

The correlation of the reflection profile with the reference profile used in profile fitting

@description: *NX_CHAR*

Describes the dataset

overlaps: (optional) *NX_INT*

An adjacency list specifying the spatial overlaps of reflections. The adjacency list is specified using an array data type where the elements of the array are the indices of the adjacent overlapped reflection

@description: *NX_CHAR*

Describes the dataset

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXreflections.nxdl.xml

NXseparator

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

definition for an electrostatic separator.

Symbols:

No symbol table

Groups cited: *NXlog*

Structure:

description: *NX_CHAR*

extended description of the separator.

beamline_distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

define position of beamline element relative to production target

set_Bfield_current: (optional) *NX_FLOAT* {units=*NX_CURRENT*}

current set on magnet supply.

set_Efield_voltage: (optional) *NX_FLOAT* {units=*NX_VOLTAGE*}

current set on HT supply.

read_Bfield_current: (optional) *NXlog*

current read from magnet supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_Bfield_voltage: (optional) *NXlog*

voltage read from magnet supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

read_Efield_current: (optional) *NXlog*

current read from HT supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_Efield_voltage: (optional) *NXlog*

voltage read from HT supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXseparator.nxdl.xml

NXsnsevent

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

This is a definition for event data from Spallation Neutron Source (SNS) at ORNL.

Symbols:

No symbol table

Groups cited: *NXaperture*, *NXattenuator*, *NXcollection*, *NXcrystal*, *NXdata*, *NXdetector*, *NXdisk_chopper*, *NXentry*, *NXevent_data*, *NXgeometry*, *NXinstrument*, *NXlog*, *NXmoderator*, *NXmonitor*, *NXnote*, *NXorientation*, *NXpolarizer*, *NXpositioner*, *NXsample*, *NXshape*, *NXsource*, *NXtranslation*, *NXuser*

Structure:

(entry): *NXentry*

collection_identifier: *NX_CHAR*

collection_title: *NX_CHAR*

definition: *NX_CHAR*

Official NXDL schema after this file goes to applications.

Obligatory value: `NXsnsevent`

duration: *NX_FLOAT* {units=*NX_TIME*}

end_time: *NX_DATE_TIME*

entry_identifier: *NX_CHAR*

experiment_identifier: *NX_CHAR*

notes: *NX_CHAR*

proton_charge: *NX_FLOAT* {units=*NX_CHARGE*}

raw_frames: *NX_INT*

run_number: *NX_CHAR*

start_time: *NX_DATE_TIME*

title: *NX_CHAR*

total_counts: *NX_UINT* {units=*NX_UNITLESS*}

total_uncounted_counts: *NX_UINT* {units=*NX_UNITLESS*}

DASlogs: *NXcollection*

Details of all logs, both from cvinfo file and from HistoTool (frequency and proton_charge).

(log): *NXlog*

average_value: *NX_FLOAT*

average_value_error: *NX_FLOAT*

description: *NX_CHAR*

duration: *NX_FLOAT*

maximum_value: *NX_FLOAT*

minimum_value: *NX_FLOAT*

time[nvalue]: *NX_FLOAT*

value[nvalue]: *NX_FLOAT*

(positioner): (optional) *NXpositioner*

Motor logs from cvinfo file.

average_value: *NX_FLOAT*

average_value_error: *NX_FLOAT*

description: *NX_CHAR*

duration: *NX_FLOAT*

maximum_value: *NX_FLOAT*

minimum_value: *NX_FLOAT*

time[numvalue]: *NX_FLOAT*

value[numvalue]: *NX_FLOAT*

SNSHistoTool: *NXnote*

SNSbanking_file_name: *NX_CHAR*

SNSmapping_file_name: *NX_CHAR*

author: *NX_CHAR*

command1: *NX_CHAR*

Command string for event2nxl.

date: *NX_CHAR*

description: *NX_CHAR*

version: *NX_CHAR*

(data): *NXdata*

data_x_y –> /NXentry/NXinstrument/NXdetector/data_x_y

x_pixel_offset –> /NXentry/NXinstrument/NXdetector/x_pixel_offset

y_pixel_offset –> /NXentry/NXinstrument/NXdetector/y_pixel_offset

(event_data): *NXevent_data*

event_index –> /NXentry/NXinstrument/NXdetector/event_index

event_pixel_id –> /NXentry/NXinstrument/NXdetector/event_pixel_id

event_time_of_flight –> /NXentry/NXinstrument/NXdetector/event_time_of_flight

pulse_time –> /NXentry/NXinstrument/NXdetector/pulse_time

instrument: *NXinstrument*

SNSdetector_calibration_id: *NX_CHAR*

Detector calibration id from DAS.

SNSgeometry_file_name: *NX_CHAR*

SNStranslation_service: *NX_CHAR*

beamline: *NX_CHAR*

name: *NX_CHAR*

SNS: *NXsource*

frequency: *NX_FLOAT* {units=*NX_FREQUENCY*}

name: *NX_CHAR*

probe: *NX_CHAR*

type: *NX_CHAR*

(detector): *NXdetector*

azimuthal_angle[numx, numy]: *NX_FLOAT* {units=*NX_ANGLE*}

data_x_y[numx, numy]: *NX_UINT*

expect `signal=2 axes="x_pixel_offset,y_pixel_offset“`

distance[numx, numy]: *NX_FLOAT* {units=*NX_LENGTH*}

event_index[numpulses]: *NX_UINT*

event_pixel_id[numevents]: *NX_UINT*

event_time_of_flight[numevents]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

pixel_id[numx, numy]: *NX_UINT*

polar_angle[numx, numy]: *NX_FLOAT* {units=*NX_ANGLE*}

pulse_time[numpulses]: *NX_FLOAT* {units=*NX_TIME*}

total_counts: *NX_UINT*

x_pixel_offset[numx]: *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_offset[numy]: *NX_FLOAT* {units=*NX_LENGTH*}

origin: *NXgeometry*

orientation: *NXorientation*

value[6]: *NX_FLOAT*

Six out of nine rotation parameters.

shape: *NXshape*

description: *NX_CHAR*

shape: *NX_CHAR*

size[3]: *NX_FLOAT* {units=*NX_LENGTH*}

translation: *NXtranslation*

distance[3]: *NX_FLOAT* {units=*NX_LENGTH*}

(disk_chopper): (optional) *NXdisk_chopper*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

moderator: *NXmoderator*

coupling_material: *NX_CHAR*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

temperature: *NX_FLOAT* {units=*NX_TEMPERATURE*}

type: *NX_CHAR*

(aperture): (optional) *NXaperture*

x_pixel_offset: *NX_FLOAT* {units=*NX_LENGTH*}

origin: *NXgeometry*

orientation: *NXorientation*

value[6]: *NX_FLOAT*

Six out of nine rotation parameters.

shape: *NXshape*

description: *NX_CHAR*

shape: *NX_CHAR*

size[3]: *NX_FLOAT* {units=*NX_LENGTH*}

translation: *NXtranslation*

distance[3]: *NX_FLOAT* {units=*NX_LENGTH*}

(attenuator): (optional) *NXattenuator*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

(polarizer): (optional) *NXpolarizer*

(crystal): (optional) *NXcrystal*

type: *NX_CHAR*

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

origin: *NXgeometry*

description: *NX_CHAR*

orientation: *NXorientation*

value[6]: *NX_FLOAT*

Six out of nine rotation parameters.

shape: *NXshape*

description: *NX_CHAR*

shape: *NX_CHAR*

size: *NX_FLOAT* {units=*NX_LENGTH*}

translation: *NXtranslation*

distance[3]: *NX_FLOAT* {units=*NX_LENGTH*}

(monitor): (optional) *NXmonitor*

data[numtimechannels]: *NX_UINT*

expect `signal=1 axes="time_of_flight"`

distance: *NX_FLOAT* {units=*NX_LENGTH*}

mode: *NX_CHAR*

time_of_flight[numtimechannels + 1]: *NX_FLOAT* {units=*NX_TIME*}

sample: *NXsample*

changer_position: *NX_CHAR*

holder: *NX_CHAR*

identifier: *NX_CHAR*

name: *NX_CHAR*

Descriptive name of sample

nature: *NX_CHAR*

(user): *NXuser*

facility_user_id: *NX_CHAR*

name: *NX_CHAR*

role: *NX_CHAR*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXsnsevent.nxdl.xml

NXsnshisto

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

This is a definition for histogram data from Spallation Neutron Source (SNS) at ORNL.

Symbols:

No symbol table

Groups cited: *NXaperture*, *NXattenuator*, *NXcollection*, *NXcrystal*, *NXdata*, *NXdetector*, *NXdisk_chopper*, *NXentry*, *NXfermi_chopper*, *NXgeometry*, *NXinstrument*, *NXlog*, *NXmoderator*, *NXmonitor*, *NXnote*, *NXorientation*, *NXpolarizer*, *NXpositioner*, *NXsample*, *NXshape*, *NXsource*, *NXtranslation*, *NXuser*

Structure:

(entry): *NXentry*

collection_identifier: *NX_CHAR*

collection_title: *NX_CHAR*

definition: *NX_CHAR*

Official NXDL schema after this file goes to applications.

Obligatory value: `NXsnshisto`

duration: *NX_FLOAT* {units=*NX_TIME*}

end_time: *NX_DATE_TIME*

entry_identifier: *NX_CHAR*

experiment_identifier: *NX_CHAR*

notes: *NX_CHAR*

proton_charge: *NX_FLOAT* {units=*NX_CHARGE*}

raw_frames: *NX_INT*

run_number: *NX_CHAR*

start_time: *NX_DATE_TIME*

title: *NX_CHAR*

total_counts: *NX_UINT* {units=*NX_UNITLESS*}

total_uncounted_counts: *NX_UINT* {units=*NX_UNITLESS*}

DASlogs: *NXcollection*

Details of all logs, both from cvinfo file and from HistoTool (frequency and proton_charge).

(log): *NXlog*

average_value: *NX_FLOAT*

average_value_error: *NX_FLOAT*

description: *NX_CHAR*

duration: *NX_FLOAT*

maximum_value: *NX_FLOAT*

minimum_value: *NX_FLOAT*

time[nvalue]: *NX_FLOAT*

value[nvalue]: *NX_FLOAT*

(positioner): (optional) *NXpositioner*

Motor logs from cvinfo file.

average_value: *NX_FLOAT*

average_value_error: *NX_FLOAT*

description: *NX_CHAR*

duration: *NX_FLOAT*

maximum_value: *NX_FLOAT*

minimum_value: *NX_FLOAT*

time[numvalue]: *NX_FLOAT*

value[numvalue]: *NX_FLOAT*

SNSHistoTool: *NXnote*

SNSbanking_file_name: *NX_CHAR*

SNSmapping_file_name: *NX_CHAR*

author: *NX_CHAR*

command1: *NX_CHAR*

Command string for event2histo_nxl.

date: *NX_CHAR*

description: *NX_CHAR*

version: *NX_CHAR*

(data): *NXdata*

data –> /NXentry/NXinstrument/NXdetector/data

data_x_time_of_flight –> /NXentry/NXinstrument/NXdetector/data_x_time_of_flight

data_x_y –> /NXentry/NXinstrument/NXdetector/data_x_y

data_y_time_of_flight –> /NXentry/NXinstrument/NXdetector/data_y_time_of_flight

pixel_id –> /NXentry/NXinstrument/NXdetector/pixel_id

time_of_flight –> /NXentry/NXinstrument/NXdetector/time_of_flight

total_counts –> /NXentry/NXinstrument/NXdetector/total_counts

x_pixel_offset –> /NXentry/NXinstrument/NXdetector/x_pixel_offset

y_pixel_offset –> /NXentry/NXinstrument/NXdetector/y_pixel_offset

instrument: *NXinstrument*

SNSdetector_calibration_id: *NX_CHAR*

Detector calibration id from DAS.

SNSgeometry_file_name: *NX_CHAR*

SNStranslation_service: *NX_CHAR*

beamline: *NX_CHAR*

name: *NX_CHAR*

SNS: *NXsource*

frequency: *NX_FLOAT* {units=*NX_FREQUENCY*}

name: *NX_CHAR*

probe: *NX_CHAR*

type: *NX_CHAR*

(detector): *NXdetector*

azimuthal_angle[numx, numy]: *NX_FLOAT* {units=*NX_ANGLE*}

data[numx, numy, numtof]: *NX_UINT*

data_x_time_of_flight[numx, numtof]: *NX_UINT*

data_x_y[numx, numy]: *NX_UINT*

data_y_time_of_flight[numy, numtof]: *NX_UINT*

distance[numx, numy]: *NX_FLOAT* {units=*NX_LENGTH*}

pixel_id[numx, numy]: *NX_UINT*

polar_angle[numx, numy]: *NX_FLOAT* {units=*NX_ANGLE*}

time_of_flight[numtof + 1]: *NX_FLOAT* {units=*NX_TIME_OF_FLIGHT*}

total_counts: *NX_UINT*

x_pixel_offset[numx]: *NX_FLOAT* {units=*NX_LENGTH*}

y_pixel_offset[numy]: *NX_FLOAT* {units=*NX_LENGTH*}

origin: *NXgeometry*

orientation: *NXorientation*

value[6]: *NX_FLOAT*

Six out of nine rotation parameters.

shape: *NXshape*

description: *NX_CHAR*

shape: *NX_CHAR*

size[3]: *NX_FLOAT* {units=*NX_LENGTH*}

translation: *NXtranslation*

distance[3]: *NX_FLOAT* {units=*NX_LENGTH*}

(disk_chopper): (optional) *NXdisk_chopper*

Original specification called for NXchopper, which is not a valid NeXus base class. Select either NXdisk_chopper or NXfermi_chopper, as appropriate.

distance: *NX_FLOAT* {units=*NX_LENGTH*}

(fermi_chopper): (optional) *NXfermi_chopper*

Original specification called for NXchopper, which is not a valid NeXus base class. Select either NXdisk_chopper or NXfermi_chopper, as appropriate.

distance: *NX_FLOAT* {units=*NX_LENGTH*}

moderator: *NXmoderator*

coupling_material: *NX_CHAR*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

temperature: *NX_FLOAT* {units=*NX_TEMPERATURE*}

type: *NX_CHAR*

(aperture): (optional) *NXaperture*

x_pixel_offset: *NX_FLOAT* {units=*NX_LENGTH*}

origin: *NXgeometry*

orientation: *NXorientation*

value[6]: *NX_FLOAT*

Six out of nine rotation parameters.

shape: *NXshape*

description: *NX_CHAR*

shape: *NX_CHAR*

size[3]: *NX_FLOAT* {units=*NX_LENGTH*}

translation: *NXtranslation*

distance[3]: *NX_FLOAT* {units=*NX_LENGTH*}

(attenuator): (optional) *NXattenuator*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

(polarizer): (optional) *NXpolarizer*

(crystal): (optional) *NXcrystal*

type: *NX_CHAR*

wavelength: *NX_FLOAT* {units=*NX_WAVELENGTH*}

origin: *NXgeometry*

description: *NX_CHAR*

orientation: *NXorientation*

value[6]: *NX_FLOAT*

Six out of nine rotation parameters.

shape: *NXshape*

description: *NX_CHAR*

shape: *NX_CHAR*

size: *NX_FLOAT* {units=*NX_LENGTH*}

translation: *NXtranslation*

distance[3]: *NX_FLOAT* {units=*NX_LENGTH*}

(monitor): (optional) *NXmonitor*

data[numtimechannels]: *NX_UINT*

distance: *NX_FLOAT* {units=*NX_LENGTH*}

mode: *NX_CHAR*

time_of_flight[numtimechannels + 1]: *NX_FLOAT* {units=*NX_TIME*}

sample: *NXsample*

changer_position: *NX_CHAR*

holder: *NX_CHAR*

identifier: *NX_CHAR*

name: *NX_CHAR*

Descriptive name of sample

nature: *NX_CHAR*

(user): *NXuser*

facility_user_id: *NX_CHAR*

name: *NX_CHAR*

role: *NX_CHAR*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXsnshisto.nxdl.xml

NXsolenoid_magnet

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

definition for a solenoid magnet.

Symbols:

No symbol table

Groups cited: *NXlog*

Structure:

description: *NX_CHAR*

extended description of the magnet.

beamline_distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

define position of beamline element relative to production target

set_current: (optional) *NX_FLOAT* {units=*NX_CURRENT*}

current set on supply.

read_current: (optional) *NXlog*

current read from supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_voltage: (optional) *NXlog*

voltage read from supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXsolenoid_magnet.nxdl.xml

NXspecdata

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

Data collected by SPEC control and data acquisition software

SPEC [1] is software for instrument control and data acquisition in X-ray diffraction experiments.

Symbols:

No symbol table

Groups cited: *NXcrystal*, *NXdata*, *NXentry*, *NXinstrument*, *NXmonitor*, *NXnote*, *NXuser*

Structure:

@default: *NX_CHAR*

Declares which *NXentry* group contains the data to be shown by default. It is needed to resolve ambiguity when more than one *NXentry* group exists. The value is the name of the default *NXentry* group.

@HDF5_Version: *NX_CHAR*

Version of HDF5 library used in writing the file (as specified in *NXroot*).

Note this attribute is spelled with uppercase "V", different than other version attributes.

@h5py_version: *NX_CHAR*

version of h5py Python package used to write this HDF5 file

@SPEC_file: *NX_CHAR*

original SPEC data file name from **#F** line in file header

@SPEC_date: *NX_CHAR*

date from **#D** line in file header, in ISO8601 format

@SPEC_epoch: *NX_INT*

UNIX time epoch from **#E** line in file header

@SPEC_comments: *NX_CHAR*

any **#C** lines in file header, stored as one string with newlines between comments

@SPEC_num_headers: *NX_INT*

Number of header sections found in the spec file

(entry): *NXentry*

one scan from a SPEC data file, starts with a **#S** line

@default: *NX_CHAR*

Declares which *NXdata* group contains the data to be shown by default. It is needed to resolve ambiguity when more than one *NXdata* group exists. The value is the name of the default *NXdata* group.

definition: *NX_CHAR*

[1] SPEC: https://certif.com

Official NeXus NXDL schema to which this subentry conforms.

Obligatory value: `NXspecdata`

scan_number: *NX_NUMBER*

SPEC scan number

title: *NX_CHAR*

SPEC scan number and command, from **#S** line

SPEC data file line:

```
#S 1  cscan en 690 750 60 0
```

title:

```
1  cscan en 690 750 60 0
```

command: *NX_CHAR*

SPEC scan command, from **#S** line, after the scan number.

SPEC data file line `#S 1 cscan en 690 750 60 0`

command* `cscan en 690 750 60 0`

date: *NX_DATE_TIME*

date from **#D** line in scan header, in ISO8601 format

comments: *NX_CHAR*

Any **#C** lines in this scan, stored as one string with newlines between comments

Q: *NX_NUMBER*

#Q – Q (hkl) at start of scan

array of $[h\ k\ l]$

TEMP_SP: *NX_NUMBER*

#X – temperature set point

DEGC_SP: *NX_NUMBER*

#X – temperature set point (C)

(monitor): *NXmonitor*

@description: *NX_CHAR*

mode: *NX_CHAR*

Count to a preset value based on either clock time (timer) or received monitor counts (monitor).

Any of these values: `monitor|timer`

preset: *NX_NUMBER*

preset value for time or monitor

- **#M** – counting against this constant monitor count (see #T)
- **#T** – counting against this constant number of seconds (see #M)

@units: *NX_CHAR*

data: *NX_NUMBER*

array(s) of monitor data

count_time: *NX_NUMBER*

array(s) of monitor data

data: *NXdata*

detector (and MCA) data from this scan

@description: *NX_CHAR*

@signal: *NX_CHAR*

name of the field with the plottable data, typically the last column for 1-D scans

This is the primary dependent axis, such as two-theta detector. This field must exist (or be linked) in this *NXdata* group.

@axes: *NX_CHAR*

name(s) of the field(s) for plotting the data, typically the first column for 1-D scans

These are the independent axes, such as positioners. For 2-D or higher dimension data, there will be a field named for each dimension, separated by ":" (preferred) or "," or " " (whitespace).

Such as for 2-D data plotted against *energy* and *th*:

```
@axes = energy:th
```

This(these) field(s) must exist (or be linked) in this *NXdata* group.

@AXISNAME_indices: *NX_NUMBER*

For each field named in *@axes*, there will be an instance of this attribute, defining into which dimensions of the *@signal* data this field applies. The value of this attribute is a list of index numbers using 0-based indexing (first dimension is 0, seconds i 1, ...).

Such as for 2-D data plotted against *energy* and *th*:

```
@energy_indices = [0]
@th_indices = [1]
```

data: *NX_NUMBER*

one column of data from the scan

HDF5 requires that each member of a group must have a unique name.

Pick the name of column from **#L** but make it unique which means if the same name is used in more than one column, append a number to the extra instances to make them unique yet preserve their content, just in case they might be different.

Example: `seconds seconds` becomes `seconds` and `seconda_1`.

@spec_name: *NX_CHAR*

name as specified in **#L** line, before it was made unique for HDF5

@units: *NX_CHAR*

Unless stated otherwise, units (not declared in the SPEC data file) are assumed to be *counts* for detectors and "unknown" for positioners or other scan columns.

intensity_factor: *NX_NUMBER*

#I – intensity normalizing factor

mca: *NX_NUMBER*

_mca_channel_: *NX_NUMBER*

mca1: *NX_NUMBER*

_mca1_channel_: *NX_NUMBER*

counter_cross_reference: *NXnote*

associates values declared in **#J** and **#j** scan header lines

@comment: *NX_CHAR*

@description: *NX_CHAR*

positioner_cross_reference: *NXnote*

associates values declared in **#O** and **#o** scan header lines

@comment: *NX_CHAR*

@description: *NX_CHAR*

spec: *NXinstrument*

various metadata from the SPEC scan header that have well-known NeXus base clases

UB: *NXcrystal*

Orientation matrix of single crystal sample using Busing-Levy convention

orientation_matrix[3, 3]: *NX_FLOAT*

#G3 line in scan header

G: *NXnote*

SPEC geometry variables for this diffractometer geometry (instrument specific)

TODO: give interpreted name for each array value (need to figure out how to get the names)

@comment: *NX_CHAR*

@description: *NX_CHAR*

G0: *NX_NUMBER*

geometry parameters from G[] array (geo mode, sector, etc)

G1: *NX_NUMBER*

geometry parameters from U[] array (lattice constants, orientation reflections)

G2: *NX_NUMBER*

not used, although some files has a single zero value

G4: *NX_NUMBER*

geometry parameters from Q[] array (lambda, frozen angles, cut points, etc)

positioners: *NXnote*

names and values of all positioners (**#O** and **#P** lines) in scan header

@description: *NX_CHAR*

positioner: *NX_NUMBER*

one positioner from the scan header

HDF5 requires that each member of a group must have a unique name.

SPEC assigns a unique name to each positioner, no extra work is neccesary to comply with the HDF5 rule for unique names in a group.

MCA: *NXnote*

#@CALIB – coefficients to compute a scale based on the MCA channel number

@description: *NX_CHAR*

preset_time: *NX_NUMBER*

elapsed_live_time: *NX_NUMBER*

elapsed_real_time: *NX_NUMBER*

number_saved: *NX_NUMBER*

first_saved: *NX_INT*

last_saved: *NX_INT*

reduction_coef: *NX_NUMBER*

calib_a: *NX_NUMBER*

calib_b: *NX_NUMBER*

calib_c: *NX_NUMBER*

ROI: *NXnote*

roiN: *NX_CHAR*

numbered regions of interest, use an index number as part of the name

@description: *NX_CHAR*

`first_channel,last_channel`

@first_channel: *NX_INT*

@last_channel: *NX_INT*

metadata: *NXnote*

SPEC metadata (UNICAT-style #H and #V lines)

This is a block that may be unique to SPEC files acquired at certain APS beam lines. Other facilities or instruments may use this block for storing key:value pairs of data where the values have suitable attributes (such as units).

@description: *NX_CHAR*

SPEC_user: *NXuser*

SPEC_user: *NX_CHAR*

user name from first **#C** line in file header

_unrecognized: *NXnote*

Fallback for any SPEC data file control lines not otherwise placed into groups or fields elsewhere in this specification.

@comment: *NX_CHAR*

@description: *NX_CHAR*

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXspecdata.nxdl.xml

NXspin_rotator

Status:

contributed definition, extends *NXobject*, version 1.0

Description:

definition for a spin rotator.

Symbols:

No symbol table

Groups cited: *NXlog*

Structure:

description: *NX_CHAR*

extended description of the spin rotator.

beamline_distance: (optional) *NX_FLOAT* {units=*NX_LENGTH*}

define position of beamline element relative to production target

set_Bfield_current: (optional) *NX_FLOAT* {units=*NX_CURRENT*}

current set on magnet supply.

set_Efield_voltage: (optional) *NX_FLOAT* {units=*NX_VOLTAGE*}

current set on HT supply.

read_Bfield_current: (optional) *NXlog*

current read from magnet supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_Bfield_voltage: (optional) *NXlog*

voltage read from magnet supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

read_Efield_current: (optional) *NXlog*

current read from HT supply.

value: *NX_CHAR* {units=*NX_CURRENT*}

read_Efield_voltage: (optional) *NXlog*

voltage read from HT supply.

value: *NX_CHAR* {units=*NX_VOLTAGE*}

NXDL Source: https://github.com/nexusformat/definitions/blob/master/contributed_definitions/NXspin_rotator.nxdl.xml

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 *Example NeXus C programs using native HDF5 commands*.

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.

More in-depth and up-to-date information about the NeXus Application Programming Interface for the various language backends is available on-line from http://download.nexusformat.org.

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

Doxygen documentation is provided online:

C http://download.nexusformat.org/doxygen/html-c/

C++ http://download.nexusformat.org/doxygen/html-cpp/

4.3.2 NAPI Fortran 77 Interface

Doxygen documentation is provided for the f77 NAPI. (http://download.nexusformat.org/doxygen/html-f77/)

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)`

Also see the doxygen documentation. (http://download.nexusformat.org/doxygen/html-f90/)

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 (http://download.nexusformat.org/kits/4.3.0/win64/), 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

```
    String ame = "Alle meine Entchen";
    nf.makedata("string_data",NexusFile.NX_CHAR,
            1,ame.length()+2);
    nf.opendata("string_data");
    nf.putdata(ame.getBytes());
```

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 Doxygen API documentation [1]

[1] http://download.nexusformat.org/doxygen/html-java/

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 Python Interface

Documentation available in pydoc and doxygen. (http://download.nexusformat.org/doxygen/html-python)

4.3.6 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 (http://htmlpreview.github.com/?https://github.com/nexusformat/code/blob/master/bindings/idl/README.html) for the NeXus IDL binding.

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. (http://wiki.nexusformat.org/IssueReporting)

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 through the NeXus wiki.

The NeXus International Advisory Committee (NIAC) supervises the development and maintenance of the NeXus common data format for neutron, X-ray, and muon science. The NIAC supervises a technical committee to oversee the NeXus Application Programmer Interface (NAPI) and the NeXus class definitions.

There are several mechanisms in place in order to coordinate the development of NeXus with the larger community.

5.1 NeXus Wiki

First of all, there is the NeXus wiki, http://wiki.nexusformat.org/, which provides all kinds of information, including membership, minutes, and discussions from the meetings of the NIAC and Technical Committee Code Camps, proposed designs for consideration by NeXus, the NeXus logo, as well as some legacy documentation that we have not quite managed to move into the manual.

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. To establish policies concerning the definition, use, and promotion of the NeXus format.

[1] For more details about the NIAC constitution, procedures, and meetings, refer to the NIAC wiki page: http://wiki.nexusformat.org/NIAC The members of the NIAC may be reached by email: nexus-committee@nexusformat.org

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 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 with the developers of the NeXus Application Programming Interface to ensure that it supports the use of the NeXus format in the neutron, X-ray, and muon communities, 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 ensure maximum 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 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

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.

- List Address: nexus-committee@nexusformat.org
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-committee
- Archive: http://lists.nexusformat.org/pipermail/nexus-committee

5.3.4 NeXus Video Conference Announcements

There are video conferences on NeXus roughly every other week. Agenda and joining details are posted on the wiki: http://wiki.nexusformat.org/NeXus_Teleconferences In addition calender 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 commuication 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: http://download.nexusformat.org/kits/

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: http://download.nexusformat.org/kits/windows/

6.1.3 Mac OS X Installation Kit

An installation disk image (.dmg) can be downloaded from: http://download.nexusformat.org/kits/macosx/

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.

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

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).

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 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.

nxvalidate The **nxvalidate** code has been re-written entirely in C, to rely on the NXDL class files (base classes and application definitions). The new code is called **cnxvalidate**.

From the *nxvalidate* source code documentation:

> "This is the first version of **nxvalidate** written in C. 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."

Note: this tool is lives in its own GitHub repository: [1] **cnxvalidate**.

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 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 (http://docs.codehaus.org/display/GUMTREE) Gumtree is an open source project, providing a graphical user interface for instrument status and control, data acquisition and data reduction.

IDL (http://www.ittvis.com/) 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)

OpenGENIE (http://www.opengenie.org/) A general purpose data analysis and visualisation package primarily developed at the ISIS Facility, Rutherford Appleton Laboratory.

[1] **cnxvalidate**: https://github.com/nexusformat/cnxvalidate

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 (http://spec2nexus.readthedocs.io) (Python code) Converts SPEC data files and scans into NeXus HDF5 files. Provides *h5toText* utility program to inspect HDF5 file content. Provides libraries:

- *spec2nexus.spec*: python binding to read SPEC [2] data files
- *spec2nexus.eznx*: (Easy NeXus) supports writing NeXus HDF5 files using h5py

7.3 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.

7.3.1 Language APIs

h5py (http://docs.h5py.org/) HDF5 for Python (h5py) is a general-purpose Python interface to HDF5.

[2] SPEC: http://www.certif.com

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.

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: http://wiki.nexusformat.org/2014_How_to_find_default_data

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*. [2] 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 [3] as a standard for data storage at APS

1994-06 Mark Könnecke (then ISIS, now PSI) made a proposal using netCDF [4] for the European neutron scattering community while working at ISIS

[1] http://wiki.nexusformat.org/images/9/9a/NeXus_Proposal.pdf
[2] http://www.neutron.anl.gov/softness
[3] http://wiki.nexusformat.org/images/d/d5/Proposed_Data_Standard_for_the_APS.pdf
[4] http://wiki.nexusformat.org/images/b/b8/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@diamond.ac.uk>, Diamond Light Source Ltd., Didcot, UK,

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.pocoo.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 2016-12-20 11:36:51 GMT+00:00.

See also:

This document is available in these formats online:

HTML http://download.nexusformat.org/doc/html/index.html

PDF http://download.nexusformat.org/doc/NeXusManual.pdf

A very brief overview (title: *NeXus for the Impatient*) is also available (separate from the manual).

HTML http://download.nexusformat.org/doc/impatient

PDF http://download.nexusformat.org/doc/NXImpatient.pdf

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z