NeXus: a common data format for neutron, x-ray, and muon science

Release 2011-08

http://nexusformat.org

August 17, 2011

CONTENTS

NeXus is a common data format for neutron, x-ray, and muon science. It is developed as an international standard by scientists and programmers representing major scientific facilities in Europe, Asia, Australia, and North America in order to facilitate greater cooperation in the analysis and visualization of neutron, x-ray, and muon data.

Part I

Preface

With this edition of the manual, NeXus introduces a complete version of the documentation of the NeXus standard. The content from the wiki has been converted, augmented (in some parts significantly), clarified, and indexed. The NeXus Definition Language (NXDL) is introduced now to define base classes and application definitions. NXDL replaces the previous method (meta-DTD) to define NeXus classes. NeXus base classes and instrument definitions are now assigned to one of three classifications:

1. *base classes* (that represent the components used to build a NeXus data file),
2. *application definitions* (used to define a minimum set of data for a specific purpose such as scientific data processing or an instrument definition), and
3. *contributed definitions* (definitions and specifications that are in an incubation status before ratification by the NIAC).

Additional examples have been added to respond to inquiry from the users of the NeXus standard about implementation and usage. Hopefully, the improved documentation with more examples and the new NXDL will reduce the learning barriers incurred by those new to NeXus.

CHAPTER

ONE

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:

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

Some words on the notation:

- Hierarchy is represented by indentation. Objects on the same indentation level are in the same group
- The combination `name:NXclass` denotes a NeXus group with name `name` and class `NXclass`.
- A simple name (no following class) denotes a data 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 separated by comma. The `@` symbol only indicates this is an attribute. The `@` symbol 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, 10, and 12] 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).

[Lines 5-7] There are three attributes of `/entry/instrument/detector/data`: `axes`, `long_name`, and `signal`.

[Line 8] (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 9] 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 11] (reading `name = "zeolite"`) shows how a string value is represented.

[Lines 13-14] The `/entry/data` group has two datasets that are actually linked as shown. (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.)

1.1 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` inside a group of type `NXinstrument` inside a group of type `NSentry`. This nomenclature is used when the exact name of each group is either unimportant or not specified. Often, this will be used in a NXDL specification to indicate the connections of a link.

Part II

NeXus: User Manual

Contents:

CHAPTER

TWO

NEXUS INTRODUCTION

In recent years, 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 become more challenging, individual scientists, or even institutions, have found 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. For additional background information see *History*.

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

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.

2.1 What is NeXus?

The NeXus data format has four components:

1. A set of *design principles* to help people understand what is in the data files.
2. A set of *data storage objects* (base classes and application definitions) to allow the development of more portable analysis software.
3. A set of *subroutines* (utilities) to make it easy to read and write NeXus data files.
4. *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 *Fileformat*.

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* (*volume1/introduction:introduction-napi). (Further details are provided in the NAPI chapter of Volume II of this documentation.)

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

Base classes and applications, which comprise the data storage objects used in NeXus data files, are detailed in the *Class Definitions* chapter of Volume II of this documentation.

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

2.1.1 A Set of Design Principles

NeXus data files contain four types of entity: data groups, data fields, attributes, and links. See *Design-Groups* for more details.

1. **Data Groups** *Data groups* are like folders that can contain a number of fields and/or other groups.
2. **Data Fields** *Data 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). In HDF, fields are represented as HDF *Scientific Data Sets* (also known as SDS).
3. **Data Attributes** Extra information required to describe a particular group or field, such as the data units, can be stored as a data attribute.
4. **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.

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 section (*Design*).

Note: `NXentry` and `NXdata` are the only two classes **required** in a valid NeXus data file.

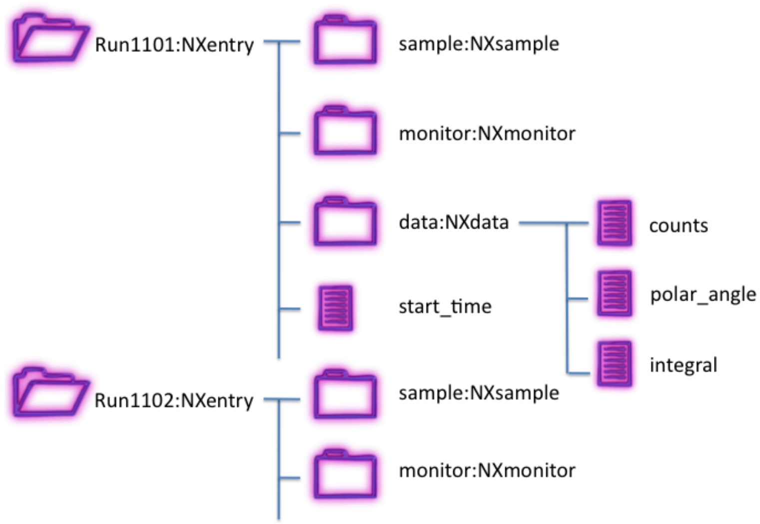

Figure 2.1: Example of a NeXus data file

`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, collimations, chopper frequencies, etc.

Since an instrument can comprise several beamline components each defined by several parameters, they 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 Data File 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.)

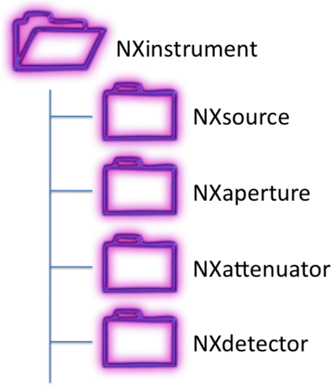

Figure 2.2: NXinstrument excerpt

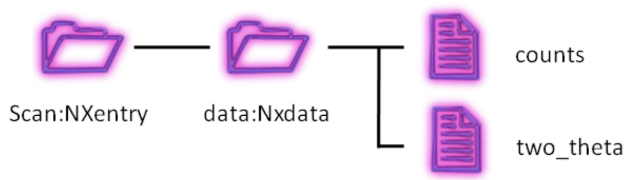

Figure 2.3: Simple Data File Example

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. In the next example, a NeXus data file is shown as XML:

`verysimple.xml`: A very simple NeXus Data file (in XML)

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <NXroot NeXus_version="4.3.0" XML_version="mxml"
        file_name="verysimple.xml"
        xmlns="http://definition.nexusformat.org/schema/3.1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://definition.nexusformat.org/schema/3.1
                            http://definition.nexusformat.org/schema/3.1/BASE.xsd"
        file_time="2010-11-12T12:40:17-06:00">
        <NXentry name="entry">
          <NXdata name="data">
                <counts
                  NAPItype="NX_INT64[15]"
                  long_name="photodiode counts"
                  signal="NX_INT32:1"
                  axes="two_theta">
                          1193        4474
                         53220      274310
                        515430      827880
                       1227100     1434640
                       1330280     1037070
```

```
                          598720     316460
                           56677       1000
                            1000
                </counts>
                <two_theta
                  NAPItype="NX_FLOAT64[15]"
                  units="degrees"
                  long_name="two_theta (degrees)">
                        18.90940         18.90960         18.90980         18.91000
                        18.91020         18.91040         18.91060         18.91080
                        18.91100         18.91120         18.91140         18.91160
                        18.91180         18.91200         18.91220
                </two_theta>
          </NXdata>
        </NXentry>
  </NXroot>
```

NeXus files are easy to create. This example NeXus file was created using a short Python program and NeXpy:

verysimple.py: Using NeXpy to write a very simple NeXus Data file (in HDF5)

```python
#
# This example uses NeXpy to build the verysimple.nx5 data file.

from nexpy.api import nexus

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]

two_theta = nexus.SDS(angle, name="two_theta",
               units="degrees",
           long_name="two_theta (degrees)")
counts = nexus.SDS(diode, name="counts", long_name="photodiode counts")
data = nexus.NXdata(counts,[two_theta])
data.nxsave("verysimple.nx5")

# The verysimple.xml file was built with this command:
#   nxconvert -x verysimple.nx5 verysimple.xml
# and then hand-edited (line breaks) for display.
```

2.1.2 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, (as described in the NXDL chapter in Volume II of this documentation) to specify the names of data fields, and other NeXus data objects. The following is an example of such a file for the simple NeXus file shown above.

`verysimple.nxdl.xml`: 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"
  svnid="$Id: introduction.rst 869 2011-08-17 23:19:06Z Pete Jemian $"
  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>
```

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 section titled Creating a NXDL Specification (*NXDL_Tutorial-CreatingNxdlSpec*).

2.1.3 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.

2.1.4 NeXus Scientific Community

Note: TODO: Show how these work together.

- NIAC
- NeXus Wiki
- ...

2.2 NAPI: The NeXus Application Programming Interface

The NeXus API consists of routines to read and write NeXus data files and was written to shield (and hide) the complexity of the HDF API from scientific programmers and users of the NeXus Data Standard.

Further documentation of the NeXus Application Programming Interface (NAPI) for bindings to specific programming language can be obtained from the NeXus development site. [1]

For a more detailed description of the internal workings of NAPI that is maintained (mostly) concurrent with code revisions, see the NAPI chapter in Volume II of this documentation and also NeXusIntern.pdf in the NeXus code repository. [2] Likely this is only interesting for experienced programmers who wish to hack the

[1] http://download.nexusformat.org
[2] http://svn.nexusformat.org/code/trunk/doc/api/NeXusIntern.pdf

NAPI.

2.2.1 How do I write a NeXus file?

The NeXus Application Program Interface (API) 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. Access from other languages, such as Python, is anticipated in the near future. It is also possible to read NeXus HDF files in a number of data analysis tools, such as LAMP, ISAW, IgorPro, and Open GENIE. NeXus XML files can be read by any program or library that supports XML.

The API uses a very simple *state* model to navigate through a NeXus file. When you open a file, the API provides a file *handle*, which then 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 of *fig.simple-example*, we walk through some parts of a typical NeXus program written in C.

Writing a simple NeXus file

```
1  #include "napi.h"
2
3   int main()
4   {
5          NXhandle fileID;
6          NXopen (’NXfile.nxs’, NXACC_CREATE, &fileID);
7            NXmakegroup (fileID, "Scan", "NXentry");
8            NXopengroup (fileID, "Scan", "NXentry");
9                  NXmakegroup (fileID, "data", "NXdata");
10                 NXopengroup (fileID, "data", "NXdata");
11                 /* somehow, we already have arrays tth and counts, each length n*/
12                   NXmakedata (fileID, "two_theta", NX_FLOAT32, 1, &n);
13                   NXopendata (fileID, "two_theta");
14                         NXputdata (fileID, tth);
15                         NXputattr (fileID, "units", "degrees", 7, NX_CHAR);
16                   NXclosedata (fileID);  /* two_theta */, NX_INT32, 1, &n);
17                   NXopendata (fileID, "counts");
18                         NXputdata (fileID, counts);
19                   NXclosedata (fileID);  /* counts */
20                 NXclosegroup (fileID);  /* data */
21           NXclosegroup (fileID);  /* Scan */
22         NXclose (&fileID);
23         return;
24 }
```

[line 6] Open the file `NXfile.nxs` with *create* access (implying write access). NAPI returns a file identifier of type `NXhandle`.

[line 7] Next, we create an `NXentry` group to contain the scan using `NXmakegroup()` and then open it for access using `NXopengroup()`.

[line 9] The plottable data is contained within an `NXdata` group, which must also be created and opened.

[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.

[line 14] Write the data using `NXputdata()`.

[line 15] With the field still open, we can also add some data attributes, such as the data units, which are specified as a character string (type `NX_CHAR`) that is 7 bytes long.

[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.

[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.

[line 20] Finally, close the groups (`NXdata` and `NXentry`) before closing the file itself.

2.2.2 How do I read a NeXus file?

Reading a NeXus file is almost identical to writing one. Obviously, it is not necessary to call `NXmakedata()` since the item already exists, but it is necessary to call one of the query routines to find out the rank and length of the data before allocating an array to store it.

Here is part of a program to read the two-theta array from the file created by *Writing a simple NeXus file* (*volume1/introduction:ex-simple-write) above.

Reading a simple NeXus file

```
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);
```

2.2.3 How do I browse a NeXus file?

NeXus files can also be viewed by a command-line browser, `NXbrowse`, which is included with the NeXus API (*NAPI: The NeXus Application Programming Interface* (*volume1/introduction:introduction-napi)). The following is an example session of using `nxbrowse` to view a data file from the LRMECS spectrometer at IPNS. The following commands are used in *Using NXbrowse* (*volume1/introduction:ex-nxbrowse-lrmecs) in this session (see the `nxbrowse` web page):

Using `NXbrowse`

```
1  %> nxbrowse lrcs3701.nxs
2
3  NXBrowse 3.0.0. Copyright (C) 2000 R. Osborn, M. Koennecke, P. Klosowski
4          NeXus_version = 1.3.3
5          file_name = lrcs3701.nxs
6          file_time = 2001-02-11 00:02:35-0600
7          user = EAG/RO
8  NX> dir
9    NX Group : Histogram1 (NXentry)
10   NX Group : Histogram2 (NXentry)
11 NX> open Histogram1
12 NX/Histogram1> dir
13   NX Data  : title[44] (NX_CHAR)
14   NX Data  : analysis[7] (NX_CHAR)
15   NX Data  : start_time[24] (NX_CHAR)
16   NX Data  : end_time[24] (NX_CHAR)
17   NX Data  : run_number (NX_INT32)
18   NX Group : sample (NXsample)
19   NX Group : LRMECS (NXinstrument)
20   NX Group : monitor1 (NXmonitor)
21   NX Group : monitor2 (NXmonitor)
22   NX Group : data (NXdata)
23 NX/Histogram1> read title
24   title[44] (NX_CHAR) = MgB2 PDOS 43.37g 8K 120meV E0@240Hz T0@120Hz
25 NX/Histogram1> open data
26 NX/Histogram1/data> dir
27   NX Data  : title[44] (NX_CHAR)
28   NX Data  : data[148,750] (NX_INT32)
29   NX Data  : time_of_flight[751] (NX_FLOAT32)
30   NX Data  : polar_angle[148] (NX_FLOAT32)
31 NX/Histogram1/data> read time_of_flight
32   time_of_flight[751] (NX_FLOAT32) = [ 1900.000000 1902.000000 1904.000000 ...]
33         units = microseconds
34         long_name = Time-of-Flight [microseconds]
35 NX/Histogram1/data> read data
36   data[148,750] (NX_INT32) = [ 1 1 0 ...]
37         units = counts
38         signal = 1
39         long_name = Neutron Counts
40         axes = polar_angle:time_of_flight
41 NX/Histogram1/data> close
42 NX/Histogram1> close
43 NX> quit
```

[line 1] Start `NXbrowse` from the UNIX command line and open file `lrcs3701.nxs` from IPNS/LRMECS.

[line 8] List the contents of the current group.

[line 11] Open the NeXus group `Histogram1`.

[line 23] Print the contents of the NeXus data labelled `title`.

[line 41] Close the current group.

[line 43] Quits `NXbrowse`.

The source code of `NXbrowse` [3] provides an example of how to write a NeXus reader. The test programs included in the NeXus API (*NAPI: The NeXus Application Programming Interface* (*volume1/introduction:introduction-napi)) may also be useful to study.

[3] https://svn.nexusformat.org/code/trunk/applications/NXbrowse/NXbrowse.c

Part III

NeXus: Reference Documentation

Contents:

Part IV

Documentation Authors

These people have made substantial contributions to the NeXus manual:

- Ray Osborn, Argonne National Laboratory, `<rosborn@anl.gov>`
- Mark Koennecke, Paul Scherrer Institut, `<Mark.Koennecke@psi.ch>`
- Przemek Klosowski, U. of Maryland and NIST, `<przemek.klosowski@nist.gov>`
- Frederick Akeroyd, Rutherford Appleton Laboratory, `<freddie.akeroyd@stfc.ac.uk>`
- Peter F. Peterson, Spallation Neutron Source, `<petersonpf@ornl.gov>`
- Pete R. Jemian, Advanced Photon Source, `<jemian@anl.gov>`
- Stuart I. Campbell, Oak Ridge National Laboratory, `<campbellsi@ornl.gov>`
- Tobias Richter, Diamond Light Source Ltd., `<Tobias.Richter@diamond.ac.uk>`

This manual is also available in a PDF version.

Part V

Contents

CHAPTER

THREE

NEXUS: USER MANUAL

Contents:

3.1 NeXus Introduction

In recent years, 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 become more challenging, individual scientists, or even institutions, have found 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. For additional background information see *History*.

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

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.

3.1.1 What is NeXus?

The NeXus data format has four components:

1. A set of *design principles* to help people understand what is in the data files.
2. A set of *data storage objects* (base classes and application definitions) to allow the development of more portable analysis software.
3. A set of *subroutines* (utilities) to make it easy to read and write NeXus data files.
4. *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 *Fileformat*.

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* (*volume1/introduction:introduction-napi). (Further details are provided in the NAPI chapter of Volume II of this documentation.)

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

Base classes and applications, which comprise the data storage objects used in NeXus data files, are detailed in the *Class Definitions* chapter of Volume II of this documentation.

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

A Set of Design Principles

NeXus data files contain four types of entity: data groups, data fields, attributes, and links. See *Design-Groups* for more details.

1. **Data Groups** *Data groups* are like folders that can contain a number of fields and/or other groups.
2. **Data Fields** *Data 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). In HDF, fields are represented as HDF *Scientific Data Sets* (also known as SDS).
3. **Data Attributes** Extra information required to describe a particular group or field, such as the data units, can be stored as a data attribute.
4. **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.

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.

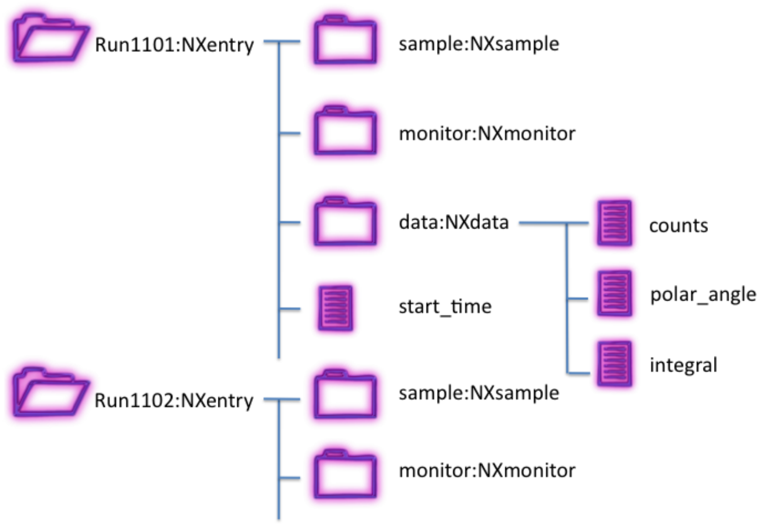

Figure 3.1: Example of a NeXus data file

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 section (*Design*).

Note: `NXentry` and `NXdata` are the only two classes **required** in a valid NeXus data file.

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

NXdata (**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, collimations, chopper frequencies, etc.

Since an instrument can comprise several beamline components each defined by several parameters, they 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.

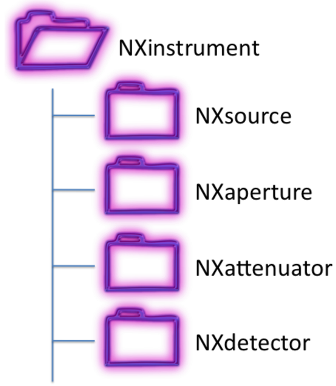

Figure 3.2: NXinstrument excerpt

Simple Data File 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.)

Figure 3.3: Simple Data File Example

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. In the next example, a NeXus data file is shown as XML:

`verysimple.xml`: A very simple NeXus Data file (in XML)

```
<?xml version="1.0" encoding="UTF-8"?>
  <NXroot NeXus_version="4.3.0" XML_version="mxml"
        file_name="verysimple.xml"
        xmlns="http://definition.nexusformat.org/schema/3.1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://definition.nexusformat.org/schema/3.1
                            http://definition.nexusformat.org/schema/3.1/BASE.xsd"
        file_time="2010-11-12T12:40:17-06:00">
        <NXentry name="entry">
          <NXdata name="data">
                <counts
```

```
                  NAPItype="NX_INT64[15]"
                  long_name="photodiode counts"
                  signal="NX_INT32:1"
                  axes="two_theta">
                           1193        4474
                          53220      274310
                         515430      827880
                        1227100     1434640
                        1330280     1037070
                         598720      316460
                          56677        1000
                           1000
                </counts>
                <two_theta
                  NAPItype="NX_FLOAT64[15]"
                  units="degrees"
                  long_name="two_theta (degrees)">
                        18.90940         18.90960         18.90980         18.91000
                        18.91020         18.91040         18.91060         18.91080
                        18.91100         18.91120         18.91140         18.91160
                        18.91180         18.91200         18.91220
                </two_theta>
          </NXdata>
        </NXentry>
  </NXroot>
```

NeXus files are easy to create. This example NeXus file was created using a short Python program and NeXpy:

`verysimple.py`: Using NeXpy to write a very simple NeXus Data file (in HDF5)

```python
#
# This example uses NeXpy to build the verysimple.nx5 data file.

from nexpy.api import nexus

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]

two_theta = nexus.SDS(angle, name="two_theta",
               units="degrees",
           long_name="two_theta (degrees)")
counts = nexus.SDS(diode, name="counts", long_name="photodiode counts")
data = nexus.NXdata(counts,[two_theta])
data.nxsave("verysimple.nx5")

# The verysimple.xml file was built with this command:
#   nxconvert -x verysimple.nx5 verysimple.xml
```

```
# and then hand-edited (line breaks) for display.
```

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, (as described in the NXDL chapter in Volume II of this documentation) to specify the names of data fields, and other NeXus data objects. The following is an example of such a file for the simple NeXus file shown above.

`verysimple.nxdl.xml`: 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"
  svnid="$Id: introduction.rst 869 2011-08-17 23:19:06Z Pete Jemian $"
  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>
```

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 section titled Creating a NXDL Specification (*NXDL_Tutorial-CreatingNxdlSpec*).

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.

NeXus Scientific Community

Note: TODO: Show how these work together.

- NIAC
- NeXus Wiki
- ...

3.1.2 NAPI: The NeXus Application Programming Interface

The NeXus API consists of routines to read and write NeXus data files and was written to shield (and hide) the complexity of the HDF API from scientific programmers and users of the NeXus Data Standard.

Further documentation of the NeXus Application Programming Interface (NAPI) for bindings to specific programming language can be obtained from the NeXus development site. [1]

For a more detailed description of the internal workings of NAPI that is maintained (mostly) concurrent with code revisions, see the NAPI chapter in Volume II of this documentation and also NeXusIntern.pdf in the NeXus code repository. [2] Likely this is only interesting for experienced programmers who wish to hack the NAPI.

How do I write a NeXus file?

The NeXus Application Program Interface (API) 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. Access from other languages, such as Python, is anticipated in the near future. It is also possible to read NeXus HDF files in a number of data analysis tools, such as LAMP, ISAW, IgorPro, and Open GENIE. NeXus XML files can be read by any program or library that supports XML.

The API uses a very simple *state* model to navigate through a NeXus file. When you open a file, the API provides a file *handle*, which then 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 of *fig.simple-example*, we walk through some parts of a typical NeXus program written in C.

Writing a simple NeXus file

```
#include "napi.h"

 int main()
 {
        NXhandle fileID;
        NXopen ('NXfile.nxs', NXACC_CREATE, &fileID);
          NXmakegroup (fileID, "Scan", "NXentry");
          NXopengroup (fileID, "Scan", "NXentry");
                NXmakegroup (fileID, "data", "NXdata");
                NXopengroup (fileID, "data", "NXdata");
                /* somehow, we already have arrays tth and counts, each length n*/
                  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 */, NX_INT32, 1, &n);
                  NXopendata (fileID, "counts");
                        NXputdata (fileID, counts);
                  NXclosedata (fileID);  /* counts */
                NXclosegroup (fileID);  /* data */
          NXclosegroup (fileID);  /* Scan */
        NXclose (&fileID);
        return;
}
```

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

[2] http://svn.nexusformat.org/code/trunk/doc/api/NeXusIntern.pdf

[line 6] Open the file `NXfile.nxs` with *create* access (implying write access). NAPI returns a file identifier of type `NXhandle`.

[line 7] Next, we create an `NXentry` group to contain the scan using `NXmakegroup()` and then open it for access using `NXopengroup()`.

[line 9] The plottable data is contained within an `NXdata` group, which must also be created and opened.

[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.

[line 14] Write the data using `NXputdata()`.

[line 15] With the field still open, we can also add some data attributes, such as the data units, which are specified as a character string (type `NX_CHAR`) that is 7 bytes long.

[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.

[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.

[line 20] Finally, close the groups (`NXdata` and `NXentry`) before closing the file itself.

How do I read a NeXus file?

Reading a NeXus file is almost identical to writing one. Obviously, it is not necessary to call `NXmakedata()` since the item already exists, but it is necessary to call one of the query routines to find out the rank and length of the data before allocating an array to store it.

Here is part of a program to read the two-theta array from the file created by *Writing a simple NeXus file* (*volume1/introduction:ex-simple-write) above.

Reading a simple NeXus file

```
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 with the NeXus API (*NAPI: The NeXus Application Programming Interface* (*volume1/introduction:introduction-napi)). The following is an example session of using `nxbrowse` to view a data file from the LRMECS spectrometer at IPNS. The following commands are used in *Using NXbrowse* (*volume1/introduction:ex-nxbrowse-lrmecs) in this session (see the `nxbrowse` web page):

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
```

```
42 NX/Histogram1> close
43 NX> quit
```

[line 1] Start `NXbrowse` from the UNIX command line and open file `lrcs3701.nxs` from IPNS/LRMECS.

[line 8] List the contents of the current group.

[line 11] Open the NeXus group `Histogram1`.

[line 23] Print the contents of the NeXus data labelled `title`.

[line 41] Close the current group.

[line 43] Quits `NXbrowse`.

The source code of `NXbrowse` [3] provides an example of how to write a NeXus reader. The test programs included in the NeXus API (*NAPI: The NeXus Application Programming Interface* (*volume1/introduction:introduction-napi)) may also be useful to study.

[3] https://svn.nexusformat.org/code/trunk/applications/NXbrowse/NXbrowse.c

CHAPTER

FOUR

NEXUS: REFERENCE DOCUMENTATION

Contents:

CHAPTER

FIVE

CHEATSHEET

This is a cheat sheet and will be removed later.

symbol	description
#	with overline, for parts
*	with overline, for chapters
=	for sections
-	for subsections
^	for subsubsections
"	for paragraphs

5.1 Symbols to mark Sections

Sidebar Title

Optional Sidebar Subtitle

This is a demo of a sidebar. Subsequent indented lines comprise the body of the sidebar, and are interpreted as body elements.

Enjoy inline math such as: $E = mc^2$ using LaTeX markup. You will need the `matplotlib` package in your Python. There is also separate math.

$$\tilde{I}(Q) = \frac{2}{l_o} \int_0^\infty I(\sqrt{(q^2 + l^2)})\, dl \tag{5.1}$$

This was possible with this definition in *conf.py*:

```
extensions = ['sphinx.ext.pngmath', 'sphinx.ext.ifconfig']
extensions.append( 'matplotlib.sphinxext.mathmpl' )
```

Part VI

Indices and tables

- *genindex*
- *search*