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

Release 3.1

http://nexusformat.org

September 18, 2012

CONTENTS

Part I

User Manual and Reference Documentation

CHAPTER

ONE

COPYRIGHT

1.1 Licenses

The NeXus manual is licensed under the terms of the GNU Free Documentation License version 1.3. See the FDL license file included with the source of this manual or the *FDL: GNU Free Documentation License* (page 139) section of this manual or refer to http://www.gnu.org/licenses/fdl-1.3.txt for more details.

The examples in the NeXus manual are licensed under the terms of the GNU Lesser General Public License version 3. See the LGPL license file included with the source of this manual or the *LGPL: GNU Lesser Gnu Public License* (page 148) section of this manual or refer to http://www.gnu.org/licenses/lgpl-3.0.txt for more details.

CHAPTER

TWO

NEXUS: USER MANUAL

2.1 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.

2.1.1 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                  @axes = "bins"
6                  @long_name = "strip detector 1-D array"
7                  @signal = 1
8              bins:[0, 1, 2, ... 1023]
9                  @long_name = "bin index numbers"
10     sample:NXsample
11         name = "zeolite"
12     data:NXdata
13         data --> /entry/instrument/detector/data
14         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. 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, 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.)

2.1.2 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` ... This nomemclature 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.

2.2 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. Additional background information is given in *Brief history of the NeXus format* (page 61).

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.

2.2.1 What is NeXus?

The NeXus data format has four components:

***A set of design principles* (page 8)** to help people understand what is in the data files.

***A set of data storage objects* (page 12)** (*ClassDefinitions-Base* and *ClassDefinitions-Application*) to allow the development of portable analysis software.

***A set of subroutines* (page 13)** (*NeXus Utilities* (page 75)) to make it easy to read and write NeXus data files.

***A Scientific Community* (page 14)** 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 *Fileformat* 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* (page 17). (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 the *NeXus Design* (page 21) chapter.

Base classes, which comprise the data storage objects used in NeXus data files, are detailed in the *ClassDefinitions-Base* 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 the *NeXus Utilities* (page 75) chapter.

A Set of Design Principles

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

***Data Groups* (page 21)** Data groups are like folders that can contain a number of fields and/or other groups.

***Data Fields* (page 22)** 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).

***Data Attributes* (page 22)** Extra information required to describe a particular group or field, such as the data units, can be stored as a data attribute.

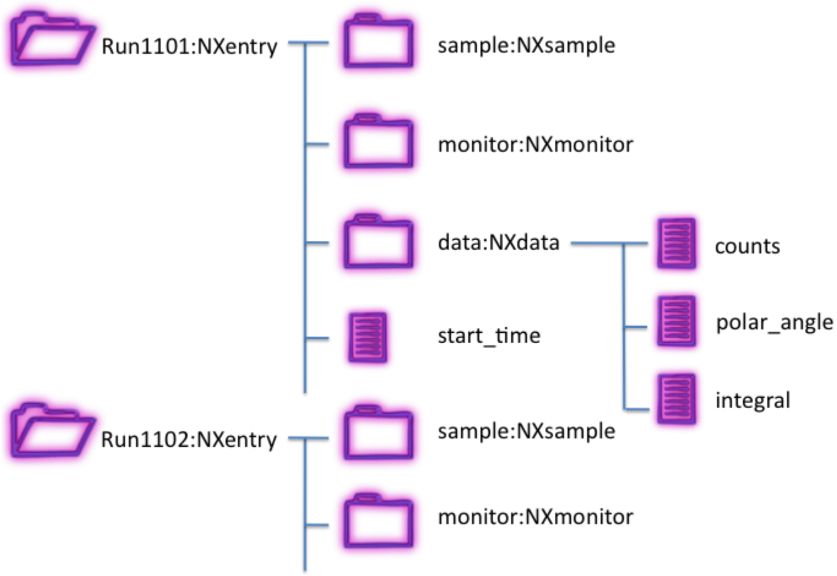
***Links* (page 23)** Links are used to reference the plottable data from `NXdata` when the data is provided in other groups such as `NXmonitor` or `NXdetector`.

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* (page 21) chapter.

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

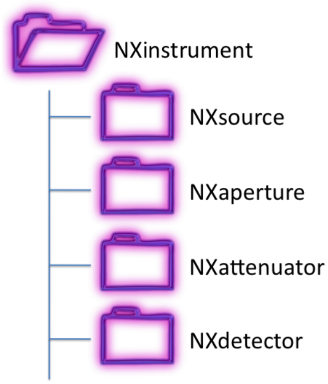
NXentry *Required:* The top level of any NeXus file contains one or more groups with the class `NXentry`. These contain all the data that is required to describe an experimental run or scan. Each `NXentry` typically contains a number of groups describing sample information (class `NXsample`), instrument 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 comprise 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. In the next example, a NeXus data file is shown as 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:

Using NeXpy to write a very simple NeXus HDF5 Data file

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

```

```python
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.

A very simple NeXus Definition Language (NXDL) file

```xml
<?xml version="1.0" ?>
<definition
  xmlns="http://definition.nexusformat.org/nxdl/3.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://definition.nexusformat.org/nxdl/3.1 ../nxdl.xsd"
```

```
  category="base"
  name="verysimple"
  version="1.0"
  svnid="$Id: verysimple.nxdl.xml 1091 2012-05-28 21:10:09Z 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>
```

For complete examples of reading and writing NeXus data files, refer to the *Examples of writing and reading NeXus data files* (page 93) chapter in Volume II. 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* (page 52).

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* (page 63) 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 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: NeXus Application Programmer Interface* (page 81)) and the NeXus class definitions.

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* (page 15)

2. A *Unified format for reduction and analysis* (page 15)
3. A *Defined dictionary of terms* (page 16)

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.

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 unified format 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 part of the scientific community with a desire to write and record scientific data, has small data volumes and a large aversion to the requirement of a complicated API necessary to access data in binary files such as HDF. For such information, the NeXus API (NAPI) has been extended by the addition of the eXtensible Markup Language (XML) [1] as an alternative to HDF. XML is a text-based format that supports compression and structured data and has broad usage in business and e-commerce. While possibly complicated, XML files are human readable, and tools for translation and extraction are plentiful. The API has routines to read and write XML data and to convert between HDF and XML.

[1] XML: http://www.w3.org/XML/. There are many other descriptions of XML, for example: http://en.wikipedia.org/wiki/XML

NeXus as a Common Data Exchange Format By the late 1980s, it had become common practice for a scientific instrument or facility to define its own data format, often at the convenience of the local computer system. Data from these facilities were not easily interchanged due to various differences in computer systems and the compression schemes of binary data. It was necessary to contact the facility to obtain a description so that one could write an import routine in software. Experience with facilities closing (and subsequent lack of access to information describing the facility data format) revealed a significant limitation with this common practice. Further, there existed a *N* * *N* number of conversion routines necessary to convert data between various formats. In *N separate file formats* (page 16), circles represent different data file formats while arrows represent conversion routines. Note that the red circle only maps to one other format.

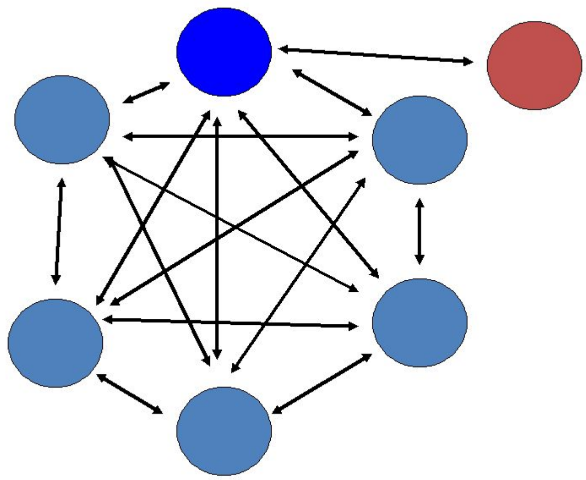


Figure 2.1: *N* separate file formats

One early idea has been for NeXus to become the common data exchange format, and thereby reduce the number of data conversion routines from *N* * *N* down to *2N*, as show in *N separate file formats joined by a common NeXus converter* (page 16).

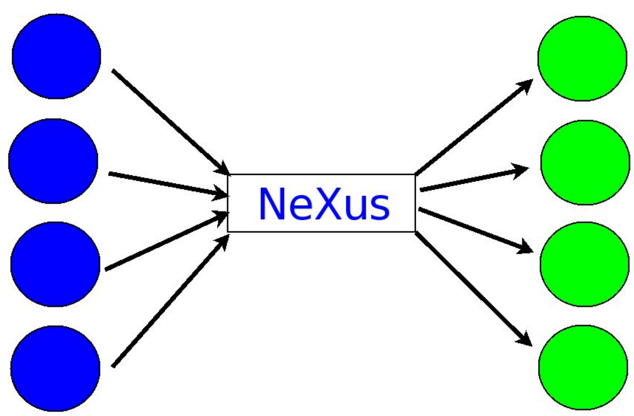


Figure 2.2: *N* separate file formats joined by a common NeXus converter

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. The NeXus Application Definitions provide the means to document the lexicography for use in data files of that scientific specialty.

2.2.2 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.

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 in Volume II of this documentation and obtained from the NeXus development site. [2]

For an even more detailed description of the internal workings of NAPI see *NeXusIntern.pdf* [3] in 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)* (page 93), in the NAPI chapter of Volume II, 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 of *fig.simple-example*, 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

```
#include "napi.h"

 int main()
 {
    NXhandle fileID;
    NXopen ("NXfile.nxs", NXACC_CREATE, &fileID);
```

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

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

```
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 */
17          NXmakedata (fileID, "counts", NX_FLOAT32, 1, &n);
18          NXopendata (fileID, "counts");
19            NXputdata (fileID, counts);
20          NXclosedata (fileID);  /* counts */
21        NXclosegroup (fileID);  /* data */
22      NXclosegroup (fileID);  /* Scan */
23    NXclose (&fileID);
24    return;
25  }
```

program analysis

1. **line 6:** Open the file `NXfile.nxs` with *create* access (implying write access). NAPI [4] 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()`. [5]
3. **line 9:** 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. [6]
5. **line 14:** Write the data using `NXputdata()`.
6. **line 15:** With the field still open, we can also add some data attributes, such as the data units, [7] [8] which are specified as a character string (`type="NX_CHAR"` [9]) 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.

[4] *NAPI: NeXus Application Programmer Interface* (page 81)

[5] See the chapter about NeXus *ClassDefinitions-Base* for more information.

[6] The *NeXus Data Types* (page 40) section describes the available data types, such as `NX_FLOAT32` and `NX_CHAR`.

[7] *NeXus Data Units* (page 41)

[8] The NeXus rule about data units is described in the *NeXus Data Units* (page 41) section.

[9] *nxdl-types*

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 *ex.simple.write* above. (Again, compare this example with one in the NAPI chapter of Volume II [10] using the native HDF5 commands.)

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* (page 17) distribution. The following is an example session of using `nxbrowse` to view a data file. The following commands are used in *ex.NXbrowse.lrmecs* in this session:

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)
```

[10] *native.hdf5.simple.read*

```
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
```

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` [11] provides an example of how to write a NeXus reader. The test programs included in the *NeXus API* (page 17) may also be useful to study.

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

2.3 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.

Note: In this manual, we use the terms *field*, *data field*, and *data item* synonymously to be consistent with their meaning between NeXus data file instances and NXDL specification files.

2.3.1 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:

***Data Groups* (page 21)** Group data fields and other groups together. Groups represent levels in the NeXus hierarchy

***Data Fields* (page 22)** Multidimensional arrays and scalars representing the actual data to be stored

***Data Attributes* (page 22)** Additional metadata which can be assigned to groups or data fields

***Links* (page 23)** Elements which point to data stored in another place in the file hierarchy

***NeXus Base Classes* (page 24)** Dictionaries of names possible in the various types of NeXus groups

***NeXus Application Definitions* (page 25)** 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.

Data 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.

Data Fields

Data fields 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). Data fields are identified by their names, which must be unique within the group in which they are stored.

Examples of data fields

`file_name` **(*NX_CHAR*)** File name of original NeXus file to assist in identification if the external name has been changed

`file_time` **(*ISO 8601*)** Date and time of file creation

`file_update_time` **(*ISO 8601*)** Date and time of last file change at close

`NeXus_version` **(*NX_CHAR*)** Version of NeXus API used in writing the file

`creator` **(*NX_CHAR*)** Facility or program where the file originated

Data Attributes

Attributes are extra (meta-)information that are associated with particular fields. They are used to annotate the 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 possible attributes can be found in table *data attributes*. Finally, NeXus files themselves have global attributes which are listed in the *global attributes*. table that identify the NeXus version, file creation time, etc. Attributes are identified by their names, which must be unique in each field.

Examples of data attributes

`units` **(*NX_CHAR*)** Data units given as character strings, must conform to the NeXus units standard. See the *NeXus Data Units* (page 41) section for details.

`signal` **(*NX_INT*)** Defines which data set contains the signal to be plotted, use `signal="1"` for main signal

`axes` **(*NX_CHAR*)** axes defines the names of the dimension scales for this data set as a colon-delimited list. Note that some legacy data files may use a comma as delimiter.

For example, suppose `data` is an array with elements `data[j][i]` (C) or `data(i,j)` (Fortran), with dimension scales `time_of_flight[i]` and `polar_angle[j]`, then `data` would have an attribute `axes="polar_angle:time_of_flight"` in addition to an attribute `signal="1"`.

`axis` **(*NX_INT*)** The original way of designating data for plotting, now superceded by the `axes` attribute. This defines the rank of the signal data for which this data set is a dimension scale in order of the fastest varying index (see a longer discussion in the section on

NXdata structure), i.e. if the array being stored is data, with elements data[j][i] in C and data(i,j) in Fortran, axis would have the following values: ith dimension (axis="1"), jth dimension (axis="2"), etc.

primary (*NX_INT32*) Defines the order of preference for dimension scales which apply to the same rank of signal data. Use primary="1" to indicate preferred dimension scale

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

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

stride (*NX_INT*) Rank values of steps to use when incrementing the dimension

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

interpretation (*NX_CHAR*) Describes how to display the data. Allowed values include:

- scaler (0-D data)
- spectrum (1-D data)
- image (2-D data)
- vertex (3-D data)

Links

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* (page 23) for a more descriptive representation of the concept of linking.

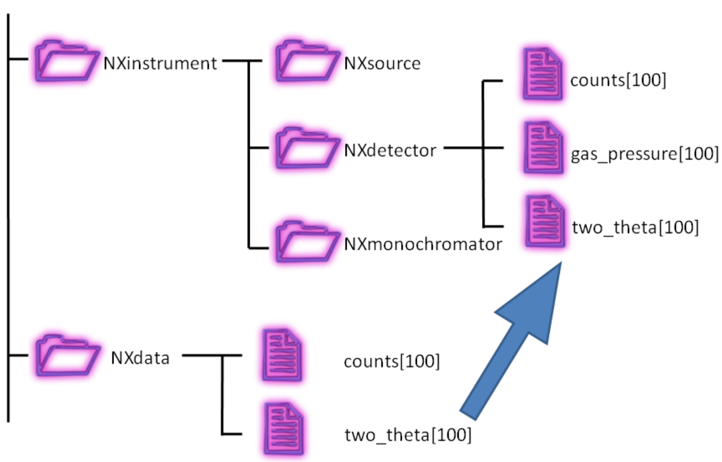


Figure 2.3: Linking in a NeXus file

NeXus also allows for links to external files. Here is an example (from Diamond Light Source) of 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 data fields and/or other data 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.

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.

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.

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.

NXgeometry `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. Here is the way an automatic plotting program ought to work:

1. Search for `NXentry` groups
2. Open an `NXentry`
3. Search for `NXdata` groups
4. Open an `NXdata` group
5. Identify the plottable data.
    (a) Search for a dataset with attribute `signal=1`. This is your main dataset. (There should be *only one* dataset that matches.)
    (b) Try to read the `axes` attribute of the main dataset, if it exists.
        i. The value of `axes` is a colon- or comma-separated list of the datasets describing the dimension scales (such as `axes="polar_angle:time_of_flight"`).
        ii. Parse `axes` and open the datasets to describe your dimension scales
    (c) If `axes` does not exist:
        i. Search for datasets with attributes `axis=1`, `axis=2`, etc. These are the datasets describing your axis. There may be several datasets for any axis, i.e. there may be multiple datasets with the attribute `axis=1`. Among them the dataset with the attribute `primary=1` is the preferred one. All others are alternative dimension scales.
        ii. Open the datasets to describe your dimension scales.
6. Having found the default plottable data and its dimension scales: make the plot

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. 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* 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.

2.3.2 NeXus Coordinate Systems

NeXus uses the **McStas coordinate system** as its laboratory coordinate system.

Coordinate systems in NeXus have undergone significant development. Initially, just motor positions of relevant motors were stored without further standardization. This soon proved to be to little and the *NeXus polar coordinate* system was developed. This system still is very close to angles 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** 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 figure *McStas Coordinate System*.

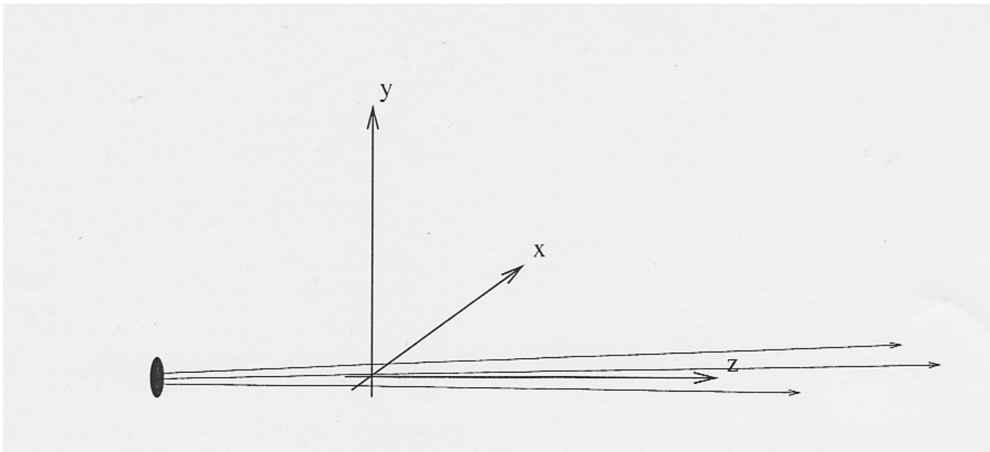


Figure 2.4: The McStas Coordinate System

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.

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* (2θ). 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* (page 28).

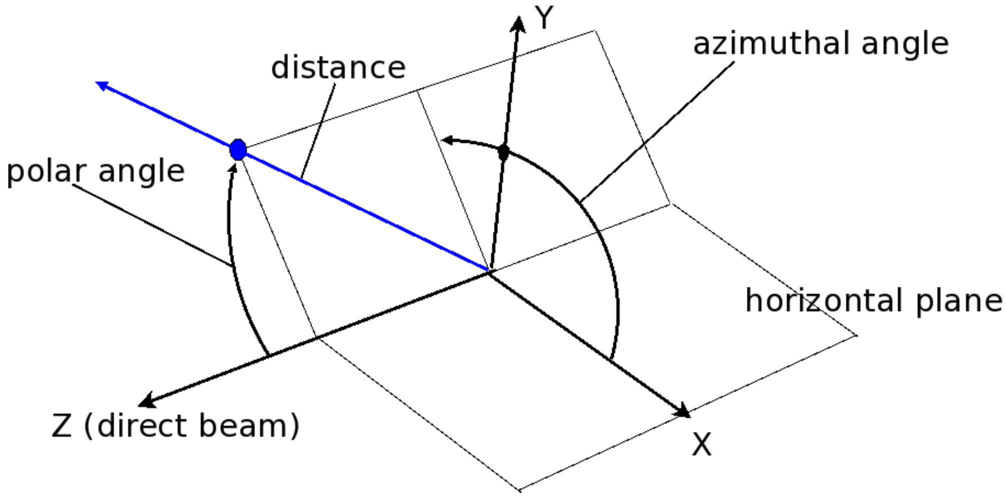


Figure 2.5: NeXus Simple (Spherical Polar) Coordinate System

Coordinate Transformations

Another way to look at coordinates is through the use of transformation matrices. 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 are commonly 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 morsels 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.

How NeXus describes the order of operations to apply has not yet been decided. The authors favourite scheme is to use a special field at each instrument component, named *transform* which describes the operations to apply to get the component into its position as a list of colon separated paths to the operations to apply relative to the current NXentry. For paths in the same group, only the name need to be given. Detectors may need two such fields: the transform field to get the detector as a whole into its position and a *transform_pixel* field which describes how the absolute position of a detector pixel can be calculated.

For the NeXus spherical coordinate system, 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.

2.3.3 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* (page 29) are provided.

Content of a Raw Data `NXentry` Group

An example raw data hierarchy is shown in figure *Raw Data* (page 29) (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) which has the attribute `signal=1`. Note that `[,]` represents a 2D array.

NeXus Raw Data Hierarchy

```
entry:NXentry
    instrument:NXinstrument
        source:NXsource
        ....
        detector:NXdetector
            data:NX_INT32[512,512]
                @signal = 1
```

```
        sample:NXsample
        control:NXmonitor
        data:NXdata
            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* (page 30), 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
    reduction:NXprocess
        program_name = "pyDataProc2010"
        version = "1.0a"
        input:NXparameter
            filename = "sn2013287.nxs"
    sample:NXsample
    data:NXdata
        data
            @signal = 1
```

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.

See figure *NeXus Multi Method Hierarchy* (page 31) for an example hierarchy. Note that [,] represents a 2D array.

NeXus Multi Method Hierarchy

```
entry:NXentry
    user:NXuser
    sample:NXsample
    instrument:NXinstument
        SASdet:NXdetector
            data:[,]
                @signal = 1
        fluordet:NXdetector
            data:[,]
                @signal = 1
        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
            detector --> /entry/instrument/fluordet/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* (page 25).

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* (page 32) 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 data fields have their normal dimensions.

NeXus Simple Scan Example

```
entry:NXentry
    instrument:NXinstrument
        detector:NXdetector
            data[NP]
                @signal = 1
    sample:NXsample
        rotation_angle[NP]
            @axis=1
    control:NXmonitor
        data[NP]
    data:NXdata
        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* (page 33) for an overview.

NeXus Simple Scan Example with Area Detector

```
entry:NXentry
    instrument:NXinstrument
        detector:NXdetector
            data:[NP,xsize,ysize]
                @signal = 1
    sample:NXsample
        rotation_angle[NP]
            @axis=1
    control:NXmonitor
        data[NP]
    data:NXdata
        data --> /entry/instrument/detector/data
        rotation_angle --> /entry/sample/rotation_angle
```

Complex *hkl* scan The next example involves a complex movement along an 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 datasets in NXdata must have the appropriate attributes as described in the axis location section.

- All other data fields have their normal dimensions.

NeXus Complex *hkl* Scan

```
entry:NXentry
    instrument:NXinstrument
        detector:NXdetector
            data[NP]
                @signal = 1
            polar_angle[NP]
                @axis = 1
            name
    sample:NXsample
        name
        rotation_angle[NP]
            @axis=1
        chi[NP]
            @axis=1
        phi[NP]
            @axis=1
        h[NP]
            @axis=1
            @primary=1
        k[NP]
            @axis=1
        l[NP]
            @axis=1
    control:NXmonitor
        data[NP]
    data:NXdata
        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
    instrument:NXinstrument
        I:NXdetector
            data:NX_NUMBER[NE,NT]
                @signal = 1
                @axes = "energy:temperature"
            energy --> /entry/monochromator/energy
            temperature --> /entry/sample/temperature
        I0:NXdetector
            data:NX_NUMBER[NE,NT]
                @signal = 1
                @axes = "energy:temperature"
            energy --> /entry/monochromator/energy
            temperature --> /entry/sample/temperature
    sample:NXsample
        temperature:NX_NUMBER[NT]
    monochromator:NXmonochromator
        energy:NX_NUMBER[NE]
    I_data:NXdata
        data --> /entry/instrument/I/data
        energy --> /entry/monochromator/energy
        temperature --> /entry/sample/temperature
    I0_data:NXdata
        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 t0o 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 fields in data files.

Naming Conventions

Group and field Names used within NeXus follow a naming convention which is made up from 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 this regular expression syntax regular expression *regular expression syntax*:

Regular expression pattern for NXDL group and field names

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

Note that this name pattern starts with a letter (upper or lower case) or "_" (underscore), then letters, numbers, and "_" and is limited to no more than 63 characters (imposed by the HDF5 rules for names).

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

```
number_of_lenses
```

For all data fields, only names from the NeXus base class dictionaries should be used. If a data field name or even a complete component is missing, please suggest the addition to the *NIAC*. 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*.

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* (page 40).

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 peoples names in `NXuser` and text notes in `NXnotes`. Because the few places where encoding is important also have unpredictable content, as well as the way in which current operating systems handle character encoding, it is practically impossible to test the encoding used. Hence, `nxvalidate` provides no messages relating to character encoding.

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 [12] 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, compatibility with the ISO 8601 standard is not assured with this substitution. The `strftime()` format specifier for this is "`%Y-%m-%d %H:%M:%S%z`".

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

NeXus Data Units

Given the plethora of possible applications of NeXus, it is difficult to define units to use. Therefore, the general rule is that you are free to store data in any unit you find fit. However, any data field must have a units attribute which describes the units, Wherever possible, SI units are preferred. NeXus units are written as a string attribute (`NX_CHAR`) and describe the engineering units. The string should be appropriate for the value. Values for the NeXus units must be specified in a format compatible with Unidata UDunits [13] Application definitions may specify units to be used for fields using an `enumeration`.

Linking Multi Dimensional Data with Axis Data

NeXus allows to store multi dimensional arrays of data. 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, at a time, there was only one global namespace for dimension scales. Thus NeXus had to come up with 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 two methods of linking each data dimension to its respective dimension scale. The preferred method uses the `axes` attribute to specify the names of each dimension scale. The original method uses the `axis` attribute to identify with an integer the axis whose value is the number of the dimension. After describing each of these methods, the two methods will be compared. A prerequisite for both methods is that the data fields describing the axis are stored together with the multi dimensional data set whose axes need to be defined in the same NeXus group. If this leads to data duplication, use links.

Linking by name using the `axes` attribute The preferred method is to define an attribute of the data itself 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:

Preferred way of denoting axes

```
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
```

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

Linking by dimension number using the `axis` attribute The original method is to define an attribute of each dimension scale 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:

Original way of denoting axes

```
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 of linking.

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.

Discussion of the two linking methods In general the method using the `axes` attribute on the multi dimensional data set should be preferred. This leaves the actual axis describing data sets unannotated and allows them to be used as an axis for other multi dimensional data. This is especially a concern as an axis describing a data set may be linked into another group where it may describe a completely different dimension of another data set.

Only when alternative axes definitions are needed, the `axis` method should be used to specify an axis of a data set. This is shown in the example above for the `some_other_angle` field where `axis="1"` denotes another possible primary axis for plotting. The default axis for plotting carries the `primary="1"` attribute.

Both methods of linking data axes will be supported in NeXus utilities that identify dimension scales, such as `NXUfindaxis()`.

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 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

Any program whose aim is to identify plottable data should use the following procedure:

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. Check to see if this field has an attribute called `axes`. If so, the 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. And then you can skip the next two steps.

5. If the `axes` attribute is not defined, search for the one-dimensional NeXus fields with attribute `primary="1"`.
6. These are the dimension scales to label the axes of each dimension of the data.
7. Link each dimension scale to the respective data dimension by the `axis` attribute (`axis="1"`, `axis="2"`, ... up to the rank of the data).
8. If necessary, close the `NXdata` group, open the next one and repeat steps 3 to 6.
9. If necessary, close the `NXentry` group, open the next one and repeat steps 2 to 7.

Consult the *NeXus API* (page 17) section, which describes the routines available to program these operations. In the course of time, generic NeXus browsers will provide this functionality automatically.

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), [14] . Rather then developing an own 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 HDF4 *vgroups* or HDF5 *groups*, NeXus data sets (or *fields*) are HDF4 *SDS (scientific data sets)* or HDF5 *datasets*. Attributes map directly to HDF group or dataset attributes.

The only special case is the NeXus class name. HDF4 supports a group class which is set with the `Vsetclass()` call and read with `VGetclass()`. HDF-5 has no group class. Thus the NeXus class is stored as an attribute to the HDF-5 group with the name `NX_class` and value of the NeXus class name.

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

Note: Examples are provided in the *Examples of writing and reading NeXus data files* (page 93) chapter of Volume II of this manual. These examples include software to write and read NeXus data files

[14] The HDF Group: http://www.hdfgroup.org/

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: - *ex.simple.write* - *ex.simple.read* - *native.hdf5.simple.write* - *native.hdf5.simple.read* - *Example-H5py-BasicWriter* - *Example-H5py-Reader*

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:

- *h5dump_group*: created in C NAPI by:

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

- *h5dump_field*: created in C NAPI by:

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

- *h5dump_attribute*: created in C NAPI by:

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

- ***h5dump_link*** –tba–

See the sections *NAPI Simple 2-D Write Example (C, F77, F90)* (page 93) and *NAPI Python Simple 3-D Write Example* (page 96) in the *Examples of writing and reading NeXus data files* (page 93) chapter of Volume II 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

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

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

```

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

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

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

Mapping NeXus into XML

This takes a bit more work than HDF. At the root of NeXus XML file is a XML element with the name `NXroot`. Further XML attributes to `NXroot` define the NeXus file level attributes. An example NeXus XML data file is provided in the *NeXus Introduction* (page 7) chapter as Example *ex.verysimple.xml*

NeXus groups are encoded into XML as elements with the name of the NeXus class and an XML attribute `name` which defines the NeXus name of the group. Further group attributes become XML attributes. An example:

NeXus group element in XML

```
    <NXentry name="entry">
    </NXentry>
```

NeXus data sets are encoded as XML elements with the name of the data. An attribute `NAPItype` defines the type and dimensions of the data. The actual data is stored as `PCDATA` [15] in the element. Another example:

NeXus data elements

```
    <mode NAPItype="NX_CHAR[7]">
        monitor
    </mode>
    <counts NAPItype="NX_INT32[4]">
```

[15] `PCDATA` is the XML term for *parsed character data* (see: http://www.w3schools.com/xml/xml_cdata.asp).

```
        21 456  127876 319
    </counts>
```

Data are printed in appropriate formats and in C storage order. The codes understood for `NAPItype` are all the NeXus data type names. The dimensions are given in square brackets as a comma separated list. No dimensions need to be given if the data is just a single value. Data attributes are represented as XML attributes. If the attribute is not a text string, then the attribute is given in the form: *type:value*, for example: `signal="NX_INT32:1"`.

NeXus links are stored in XML as XML elements with the name `NAPIlink` and a XML attribute `target` which stores the path to the linked entity in the file. If the item is linked under a different name, then this name is specified as a XML attribute name to the element `NAPIlink`.

The authors of the NeXus API worked with the author of the miniXML XML library to create a reasonably efficient way of handling numeric data with XML. Using the NeXus API handling something like 400 detectors versus 2000 time channels in XML is not a problem. But you may hit limits with XML as the file format when data becomes to large or you try to process NeXus XML files with general XML tools. General XML tools are normally ill prepared to process large amounts of numbers.

Special Attributes

NeXus makes use of some special attributes for its internal purposes. These attributes are stored as normal group or data set attributes in the respective file format. These are:

target This attribute is automatically created when items get linked. The target attribute contains a text string with the path to the source of the item linked.

napimount The `napimount` attribute is used to implement external linking in NeXus. The string is a URL to the file and group in the external file to link too. The system is meant to be extended. But as of now, the only format supported is:

```
nxfile://path-to-file#path-infile
```

This is a NeXus file in the file system at *path-to-file* and the group *path-infile* in that NeXus file.

NAPIlink NeXus supports linking items in another group under another name. This is only supported natively in HDF-5. For HDF-4 and XML a crutch is needed. This crutch is a special class name or attribute `NAPIlink` combined with the target attribute. For groups, `NAPILink` is the group class, for data items a special attribute with the name `NAPIlink`.

2.4 Constructing NeXus Files and Application Definitions

In *NeXus Design* (page 21), 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* (page 81) chapter of Volume II. For code examples, refer to *Code Examples that use the NAPI* (page 93) chapter of Volume II. Alternatively, there are examples in the *native-HDF5-Examples* 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* (page 105) section.

2.4.1 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.

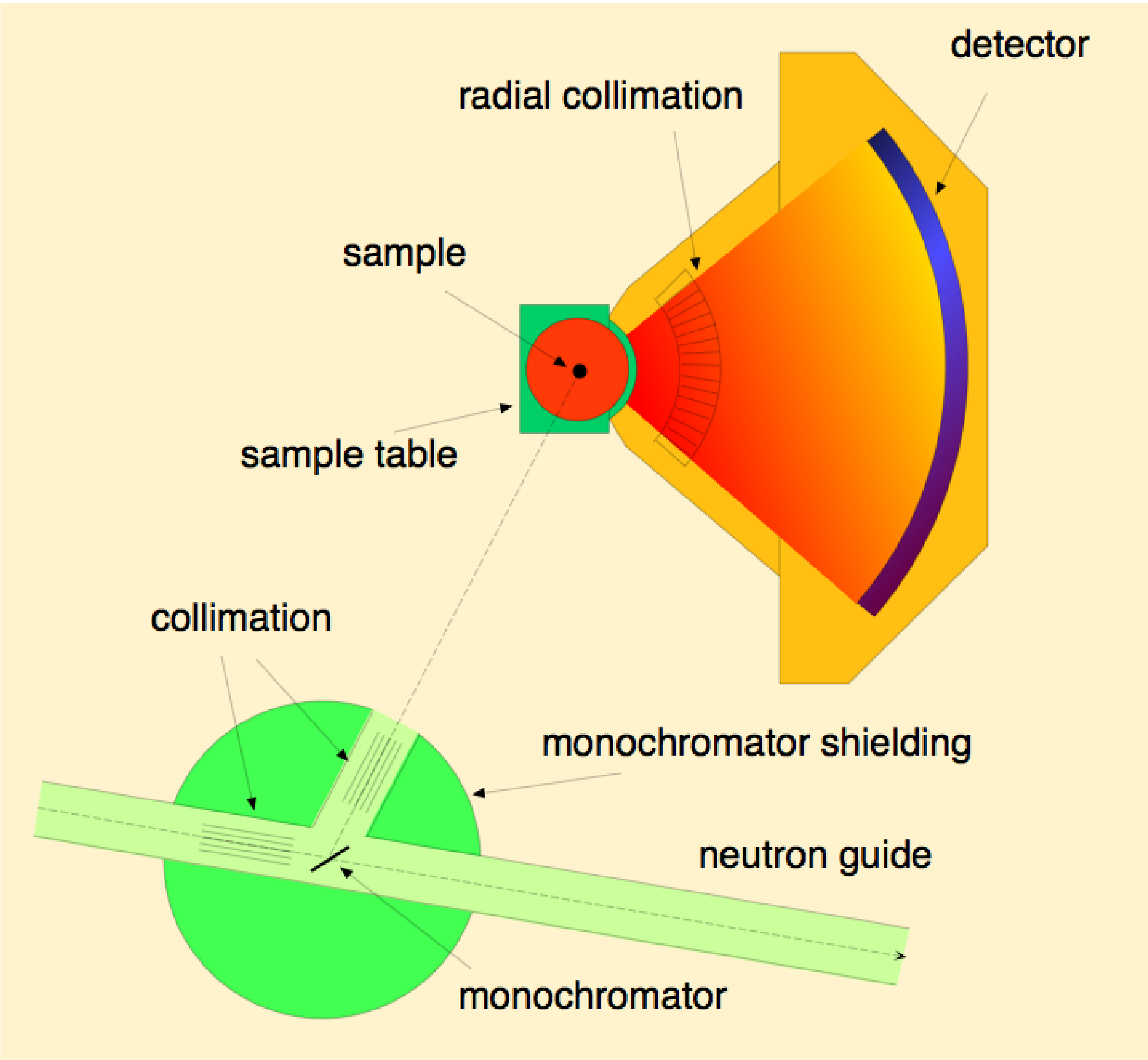


Figure 2.6: The (fictional) WONI example powder diffractometer

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* (page 49). 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* (page 50). There is a generous background to the data plus quite a number of diffraction peaks.

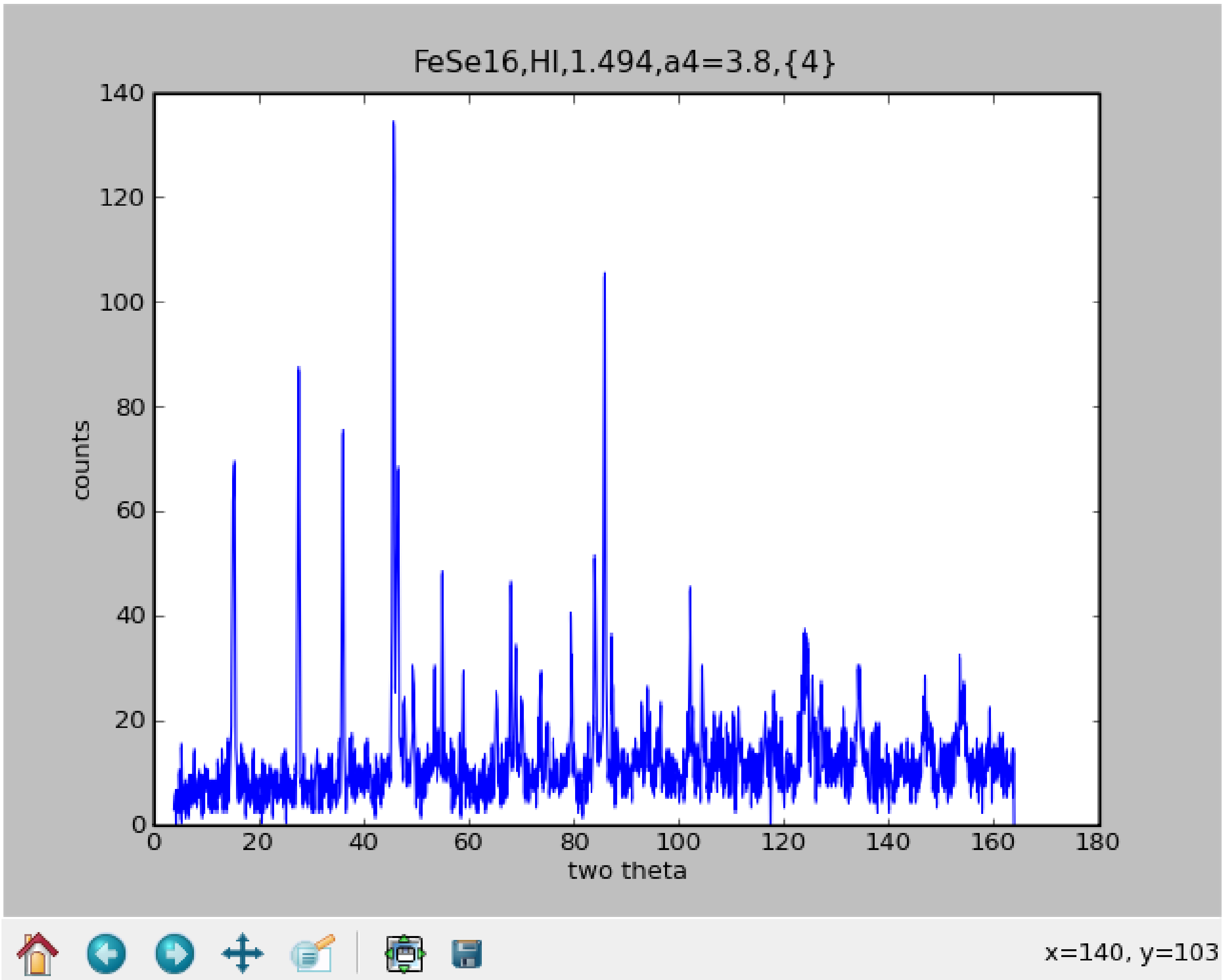


Figure 2.7: Example Powder Diffraction Plot from (fictional) WONI at HYNES

2.4.2 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 figure *FigShell*. 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 figure *FigShellMono*

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* (page 52).

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* (page 50). 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.

2.4.3 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* (page 50) 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* (page 60)) 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* (page 49)), 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 *TableWoniFullMapping*.

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 everything which possibly can occur. You do not have to give all that information. The key question is, as already said, *What is required for typical data analysis for this type of application?* You might also be unsure how to correctly store a particular data item. In such a case, contact the NIAC for help. 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 alltogether. In such a case, please feel free to contact the NIAC with a suggestion for an extension of the base classes in question.

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 *ExNxdlTemplate*. 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) 2008-2012 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

########### SVN repository information ###################
# $Date: 2012-05-28 23:10:09 +0200 (Mo, 28. Mai 2012) $
# $Author: Pete Jemian $
# $Revision: 1091 $
# $HeadURL: https://svn.nexusformat.org/definitions/branches/docbook2sphinx/manual/source/
# $Id: NX__template__.nxdl.xml 1091 2012-05-28 21:10:09Z Pete Jemian $
########### SVN repository information ###################
-->
<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* (page 57). 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) 2008-2012 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

########### SVN repository information ###################
# $Date: 2012-03-06 16:00:33 +0100 (Di, 06. Mär 2012) $
# $Author: Pete Jemian $
# $Revision: 1060 $
# $HeadURL: https://svn.nexusformat.org/definitions/branches/docbook2sphinx/applications/N
# $Id: NXmonopd.nxdl.xml 1060 2012-03-06 15:00:33Z Pete Jemian $
########### SVN repository information ###################
-->
<definition name="NXmonopd" extends="NXobject" type="group"
    category="application"
    xmlns="http://definition.nexusformat.org/nxdl/@NXDL_RELEASE@"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://definition.nexusformat.org/nxdl/@NXDL_RELEASE@ ../nxdl.xsd"
    version="1.0b"
    svnid="$Id: NXmonopd.nxdl.xml 1060 2012-03-06 15:00:33Z Pete Jemian $">
    <doc> Monochromatic Neutron and X-Ray Powder Diffraction. Instrument definition for a p
        diffractometer at a monochromatic neutron or X-ray beam. This is both suited for a
        diffractometer with a single detector or a powder diffractometer with a position se
        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
          through an omega-2theta scan like from a traditional single detector po
          diffractometer </doc>
      </field>
    </group>
    <group type="NXmonitor">
      <field name="mode">
        <doc>Count to a preset value based on either clock time (timer) or received
          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 *RegExpName*). 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.

2.4.4 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.

2.5 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.

2.5.1 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`

2.6 Brief history of the NeXus format

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.

June 1994 Mark Könnecke (then ISIS, now PSI) made a proposal using netCDF [16] for the European neutron scattering community while working at ISIS

August 1994 Jonathan Tischler (ORNL) proposed an HDF-based format [17] as a standard for data storage at APS

October 1994 Ray Osborn convened a series of three workshops called *SoftNeSS*. [18] 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.

September 1995 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.

[16] http://wiki.nexusformat.org/images/b/b8/European-Formats.pdf
[17] http://www.neutron.anl.gov/softness
[18] http://wiki.nexusformat.org/images/d/d5/Proposed_Data_Standard_for_the_APS.pdf

August 1996 The HDF-4 API is quite complex. Thus a NeXus Abstract Programmer Interface (NAPI) EDIT_ME was released which simplified reading and writing NeXus files.

October 1996 At *SoftNeSS 1996* (at ANL), after reviewing the different scientific data formats discussed, it was decided to use HDF-4 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. [19]

July 1997 SINQ at PSI started writing NeXus files to store raw data.

Summer 2001 MLNSC at LANL started writing NeXus files to store raw data

September 2002 NeXus API version 2.0.0 is released. This version brought support for the new version of HDF, HDF-5, released by the HDF group. HDF-4 imposed limits on file sizes and the number of objects in a file. These issues were resolved with HDF-5. The NeXus API abstracted the difference between the two physical file formats away form the user.

June 2003 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.

October 2003 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.

July 2005 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.

May 2007 NeXus API version 4.0.0 is released with broader support for scripting languages and the feature to link with external files.

October 2007 NeXus API version 4.1.0 is released with many bug-fixes.

October 2008 *NXDL* 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.

April 2009 NeXus API version 4.2.0 is released with additional C++, IDL, and python/numpy interfaces.

September 2009 NXDL and draft `NXsas` presented to canSAS at SAS2009 conference

[19] http://wiki.nexusformat.org/images/9/9a/NeXus_Proposal.pdf

January 2010 NXDL presented to ESRF HDF5 workshop on hyperspectral data

2.7 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.

2.7.1 NIAC: The NeXus International Advisory Committee

The purpose of the NeXus International Advisory Committee (NIAC) [20] 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.
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:

[20] 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: *NIAC*

- 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.

Footnote

2.7.2 NeXus Mailing Lists

There are several mailing lists associated with NeXus.

NeXus Mailing List We invite anyone who is associated with neutron and/or X-ray synchrotron scattering 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

NeXus International Advisory Committee (NIAC) Mailing List This list contains discussions of the *NeXus International Advisory Committee (NIAC)* (page 63), EDIT_ME 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 confined 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

NeXus Developers Mailing List This mailing list is for discussions concerning the technical development of NeXus (the Definitions, NXDL, and the NeXus Application Program Interface).

- List Address: *nexus-developers@nexusformat.org*
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-developers
- Archive: http://lists.nexusformat.org/pipermail/nexus-developers

Subversion (http://subversion.apache.org) is the revision control system used by the NeXus developers.

TRAC (http://trac.edgewall.org) is the issue tracking and bug reporting system used by the NeXus developers.

NeXus Code Subversion Mailing List Members of this list will receive an email whenever a commit is made to the *NeXus code repository* (page 65). This list cannot be posted to - all questions should instead be sent to the NeXus Developers Mailing List (*nexus-developers@nexusformat.org*).

- List Address: *nexus-code-svn@nexusformat.org*
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-code-svn
- Archive: http://lists.nexusformat.org/pipermail/nexus-code-svn

NeXus Code Tickets Mailing List Members of this list will receive an email whenever a ticket (bug/issue/task) associated with NeXus code library development is modified on the Nexus *code* TRAC server. The list of ticket updates and subversion changesets is available on the *code* repository TRAC timeline. This list cannot be posted to - see the section on *Issue Reporting* (page 68).

- List Address: *nexus-code-tickets@nexusformat.org*
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-code-tickets
- Archive: http://lists.nexusformat.org/pipermail/nexus-code-tickets
- TRAC Timeline: http://trac.nexusformat.org/code/report/1

NeXus Definitions Subversion Mailing List Members of this list will receive an email whenever a commit is made to the *NeXus definitions repository* (page 65). This list cannot be posted to - all questions should instead be sent to the NeXus Developers Mailing List (*nexus-developers@nexusformat.org*).

- List Address: *nexus-definitions-svn@nexusformat.org*
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-definitions-svn
- Archive: http://lists.nexusformat.org/pipermail/nexus-definitions-svn

NeXus Definitions Tickets Mailing List Members of this list will receive an email whenever a ticket (bug/issue/task) associated with NeXus definitions development is modified on the Nexus *definitions* TRAC server. The list of ticket updates and subversion changesets is available on the *definitions* repository TRAC timeline. This list cannot be posted to - see the section on *Issue Reporting* (page 68).

- List Address: *nexus-definitions-tickets@nexusformat.org*
- Subscriptions: http://lists.nexusformat.org/mailman/listinfo/nexus-definitions-tickets
- Archive: http://lists.nexusformat.org/pipermail/nexus-definitions-tickets
- TRAC Timeline: http://trac.nexusformat.org/definitions/report/1

2.7.3 NeXus Subversion Repositories

NeXus NXDL class definitions (both base classes and instruments) and the NeXus code library source are held in a *subversion repository*. The repository is world readable and though you can browse the *NeXus code library and applications* or *NeXus NXDL class definitions* repositories directly, a better looking interface is provided by the *ViewVC* or *TRAC* browsers.

- Browse the NeXus code (library and applications) repository using *ViewVC* or *TRAC*

- Browse NeXus definitions (NXDL classes) repository using *ViewVC* or *TRAC*

The repository can alse be interrogated for recent updates via a *query form*, such as:

```
http://svn.nexusformat.org/viewvc/NeXusCode/trunk/?view=queryform
```

For example, show me all changes in the last month for the *code (library and applications)* repository

```
http://svn.nexusformat.org/viewvc/NeXusCode/trunk/?view=query&date=month&limit_changes=100
```

or *Definition* repository

```
http://trac.nexusformat.org/definitions/timeline?daysback=30
```

If you wish to receive an email when a change is made to the repository you should join the appropriate *Mailing Lists* (page 64).

XML RSS Feed	icon
Alternatively, you can use an RSS feed to keep abreast of changes. TRAC provides a link to its RSS feed on pages with an orange *XML RSS Feed* icon at their foot such as:	

There are pages that show the subversion repository activity in a timeline format or a tabular (revision log) format.

code (library and applications) repository timeline http://trac.nexusformat.org/code/timeline

definitions repository timeline http://trac.nexusformat.org/definitions/timeline

code repository revision log http://trac.nexusformat.org/code/log

definitions repository revision log http://trac.nexusformat.org/definitions/log

Login

To update files in these repositories you will need to use a subversion client such as *TortoiseSVN/* [21] for Microsoft Windows or `svn` for command-line shells and also provide your NeXus Wiki username and password. Note that for subversion write access:

- If your Wiki username contains a space, write it with a space (i.e. do not replace the space with an _ as is done in WIKI URLs)
- You cannot use a *temporary password* (i.e. one that was emailed to you in response to a request). You must first log into MediaWiki with the temporary password and then go to account *NeXus wiki Preferences* and change the password.
- Your Wiki account must have an email address associated with it and this address must have been validated. To provide and/or validate your email address, log in and go to your account *NeXus wiki Preferences*. section.
- If you have login problems and have not changed your WIKI password since 20th October 2006, please go to the *NeXus wiki login* page and request to be emailed a new password. To synchronise TRAC/Subversion/MediaWiki required some changes to the authentication system which will have invalidated passwords set prior to that date.

[21] http://tortoisesvn.tigris.org/‘

Here are the URLs to access the subversion repositories as a developer:

code for library/applications https://svn.nexusformat.org/code/trunk

definitions for NXDL classes https://svn.nexusformat.org/definitions/trunk

checkout the code trunk

```
svn co --username "use your WIKI Username" https://svn.nexusformat.org/code/trunk nexu
```

Please report any problems via the *Issue Reporting* (page 68) system.

Committing Changes

As well as needing a valid account, you will not be able to check-in changes unless you indicate (in the log message attached to the commit) which current issues on the *Issue Reporting* (page 68) system the changes either fix or refer to. This is done by enclosing special phrases in the commit message of the form:

```
command #1
command #1, #2
command #1 & #2
command #1 and #2
```

where `command` is one of the commands detailed below and `#1` means *issue number 1* on the system, etc. You can have more then one command in a message. The following commands are supported and there is more then one spelling for each command (to make this as user-friendly as possible):

`closes, fixes` The specified issue numbers are closed with the contents of this commit message being added to it.

`references, refs, addresses, re` The specified issue numbers are left in their current status, but the contents of this commit message are added to their notes.

For example, the commit message

```
Changed blah and foo to do this or that. Fixes #10 and #12, and refs #12.
```

This will close issues #10 and #12, and add a note to #12 on the *Issue Reporting* (page 68) system. For a list of current issues, see:

- **Active tickets for the NeXus code library:** http://trac.nexusformat.org/code/report/1
- **Active tickets for NeXus definitions:** http://trac.nexusformat.org/definitions/report/1

URLs described in this section

Many Uniform Resource Locators (URLs) have been used in this section. This is a table describing them.

Subversion revision management software http://subversion.apache.org/

ViewVC versions control repository viewing software http://www.viewvc.org/

TRAC issue management software http://trac.edgewall.org

TortoiseSVN, Windows subversion client http://tortoisesvn.tigris.org/

NeXus code (library and applications) subversion repository http://svn.nexusformat.org/code/

NeXus definitions subversion repository http://svn.nexusformat.org/definitions/

ViewVC view of NeXus code (library and applications) repository http://svn.nexusformat.org/viewvc/NeXusCode

ViewVC view of NeXus definitions repository http://svn.nexusformat.org/viewvc/NeXusDefinitions

TRAC view of NeXus code (library and applications) repository http://trac.nexusformat.org/code/browser

NeXus code (library and applications) revision log http://trac.nexusformat.org/code/log

Active tickets for the NeXus code repository http://trac.nexusformat.org/code/report/1

NeXus code repository timeline http://trac.nexusformat.org/code/timeline

TRAC view of NeXus definitions repository http://trac.nexusformat.org/definitions/browser

NeXus definitions revision log http://trac.nexusformat.org/definitions/log

Active tickets for NeXus definitions http://trac.nexusformat.org/definitions/report/1

NeXus definitions repository timeline http://trac.nexusformat.org/definitions/timeline

NeXus code repository (password required) https://svn.nexusformat.org/code/trunk

NeXus definitions repository (password required) https://svn.nexusformat.org/definitions/trunk

Footnote

2.7.4 NeXus Issue Reporting

NeXus is using *TRAC* [22] for problem/issue 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. As such, the TRAC tickets for the *code* and *definitions* repositories form the basis of a *roadmap* for NeXus. You can browse issues without logging on, but to report issues you will need to login using your NeXus WIKI username and password (the *subversion login notes* (page 65) mentioned for write access to the *Subversion Server* (page 65) apply to TRAC login, too).

Whenever an update is made to a ticket, a message is also posted to the appropriate *ticket mailing list* (page 64).

NeXus Code (Library and Applications)

Report a new issue: http://trac.nexusformat.org/code

View current issues: http://trac.nexusformat.org/code/report/1

Archive of ticket update emails: http://lists.nexusformat.org/pipermail/nexus-code-tickets/

repository timeline (recent ticket and code changes): http://trac.nexusformat.org/code/timeline

repository roadmap: http://trac.nexusformat.org/code/roadmap

[22] http://trac.edgewall.org

NeXus Definitions (NXDL base classes and application definitions)

Report a new issue: http://trac.nexusformat.org/definitions

View current issues: http://trac.nexusformat.org/definitions/report/1

Archive of ticket update emails: http://lists.nexusformat.org/pipermail/nexus-definitions-tickets/

repository timeline (recent ticket and definition changes): http://trac.nexusformat.org/definitions/timeline

repository roadmap: http://trac.nexusformat.org/definitions/roadmap

Footnote

2.8 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 Public License.

The source code and binary versions for some popular platforms can be found on http://download.nexusformat.org/kits/. Up to date instructions can be found on the *Wiki* In case you need help feel free to contact the *nexus mailing list*.

2.8.1 Precompiled Binary Installation

Prerequisites

HDF5/HDF4

Since NeXus uses HDF as the main underlying binary format, it is necessary first to install the HDF subroutine libraries and include files before compiling the NeXus API. It is not usually necessary to download the HDF source code since precompiled object libraries exist for a variety of operating systems including Windows, Mac OS X, Linux, and various other flavors of Unix. Check the HDF web pages for more information: http://www.hdfgroup.org/

Packages for HDF4 and HDF5 are available for both Fedora (hdf, hdf5, hdf-devel, hdf5-devel) and Ubuntu/Debian (libhdf4g, libhdf5).

XML

The NeXus API also supports using XML as the underlying on-disk format. This uses the Mini-XML library, developed by Michael Sweet, which is also available as a precompiled binary library for several operating systems. Check the Mini-XML web pages for more information: http://www.minixml.org/

Packages for MXML are available for both Fedora (mxml, mxml-devel) and Ubuntu/Debian (libmxml1).

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
```

Microsoft Windows Installation Kit

A Windows MSI based installation kit is available and can be downloaded from: http://download.nexusformat.org/kits/windows/

Mac OS X Installation Kit

An installation disk image (.dmg) can be downloaded from: http://download.nexusformat.org/kits/macosx/

2.8.2 Source Installation

NeXus Source Code Distribution

The build uses `autoconf` (so autools are required) to determine what features will be available by your system. You must have the *development* libraries installed for all the file backends you want support for (see above). If you intend to build more than the C language bindings, you need to have the respective build support in a place where autoconf will pick them up (i.e. python development files, a Java Development Kit, etc.).

For more information see the README in the toplevel of the source distribution. In case you need help, feel free to contact the developers using the *nexus-developers mailing list*.

Download the appropriate gzipped tar file, unpack it, and run the standard configure procedure from the resulting nexus directory. For example, for version 4.2.1;

```
$ tar zxvf nexus-4.2.1.tar.gz
$ cd nexus-4.2.1
$ ./configure
```

To find out how to customize the installation, e.g., to choose different installation directories, type

```
$ ./configure --help
```

Carefully check the final output of the `configure` run. Make sure all features requested are actually enabled.

```
$ make
$ make install
```

See the README file for further instructions.

Cygwin Kits

HDF4 is not supported under CYGWIN - both HDF5 and MXML are supported and can be downloaded and built as usual. When configuring HDF5 you should explicitly pass a prefix to the configure script to make sure the libraries are installed in a "usual" location i.e.

```
./configure --prefix=/usr/local/hdf5
```

Otherwise you will have to use the `--with-hdf5=/path/to/hdf5` option later when configuring NeXus to tell it where to look for hdf5. After building hdf5, configure and build NeXus using the instructions for source code distribution above.

2.9 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. NeXus uses various automated tools to validate files. These tools include conversion of content from HDF to XML and transformation (via XSLT) from XML format to another such as NXDL, XSD, and Schematron. This chapter will first provide an overview of the process, then define the terms used in validation, then describe how multiple base classes or application definitions might apply to a given NeXus data file, and then describe the various validation techniques in more detail. 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.

2.9.1 Overview

NeXus files adhere to a set of rules and can be tested against these rules for compliance. The rules are implemented using standard tools and can themselves be tested to verify compliance with the standards for such definitions. Validation includes the testing of both NeXus data files and the NXDL specifications that describe the rules.

The rules for writing NeXus data files are different than the rules for writing NeXus class definitions. To validate a NeXus data file, these two rule sets must eventually merge, as shown in the next figure. The data file (either HDF4, HDF5, or XML) is first converted into an internal format to facilitate validation, including data types, array dimensions, naming, and other items. Most of the data is not converted since data validation is non-trivial. Also note that the units are not validated. All the NXDL files are converted into a single Schematron file (again, internal use for validation) only when NXDL revisions are checked into the NeXus definitions repository as NXDL changes are not so frequent.

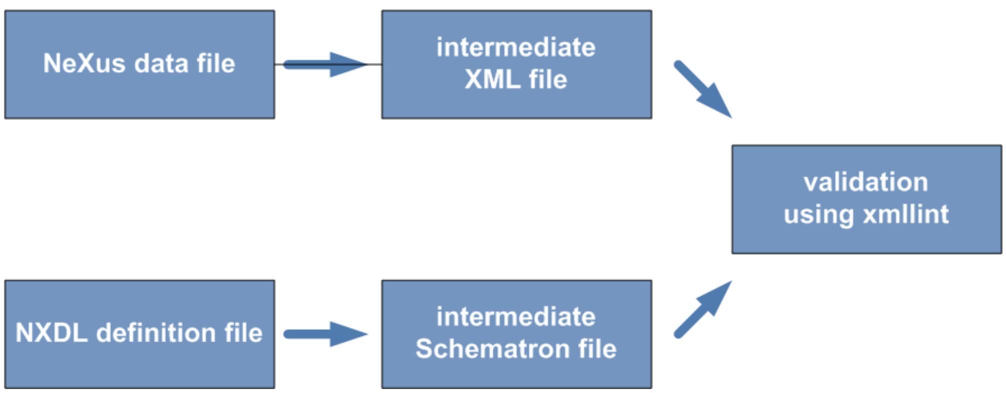


Figure 2.8: Flowchart of the NeXus validation process.

NeXus data files NeXus data files (also known as NeXus data file instances) are validated to ensure the various parts of the data file are arranged according to the governing NXDL specifications used in that file instance.

> **Note:** Since NeXus has several rules that are quite difficult to apply in either XSD or Schematron, direct validation of data files using standard tools is not possible. To validate NeXus data files, it is necessary to use `nxvalidate`.

NeXus Definition Language (NXDL) specification files NXDL files are validated to ensure they adhere to the rules for writing NeXus base classes and application definitions.

2.9.2 Definitions of these terms

Let's be clear about some terms used in this section.

HDF Hierarchical Data Format from The HDF Group. NeXus data files using HDF may be stored in either version 4 (HDF4) or version 5 (HDF5). New NeXus HDF files should only use HDF5. The preferred file extensions (but not required) include `.hdf`, `.h5`, `.nxs`, and `.nx5`.

NXDL NeXus Definition Language files define the spcifications for NeXus base classes, application definitions, and contributed classes and definitions. It is fully described in the NXDL chapter in Volume II of this documentation.

Schematron Schematron [23] is an alternative to XSD and is used to validate the content and structure of an XML file. NeXus uses Schematron internally to validate data files.

Validation File validation is the comparison of file contents, in an unbiased way, with the set of rules that define the structure of such files.

XML The eXtensible Markup Language (XML) [24] is a standard business tool for the exchange of information. It is broadly supported by a large software library in many languages. NeXus uses XML for several purposes: data files, NXDL definitions, rules, and XSLT transformations.

XSD XML files are often defined by a set of rules (or *schema*). A common language used to implement these rules is XML Schema (XSD) [25] Fundamentally, all XML, XSD, XSLT, and Schematron files are XML.

XSLT XML files can be flexible enough to convert from one set of rules to another. An example is when one company wishes to exchange catalog or production information with another. The XML StyLsheet Transformation (XSLT) [26] language is often used to describe each direction of the conversion of the XML files between the two rule sets.

2.9.3 NeXus data files may use multiple base classes or application definitions

NeXus data files may have more than one data set or may have multiple instances of just about any base class or even application definitions. The NeXus data file validation is prepared to handle this without any special effort by the provider of the data file.

2.9.4 Validation techniques

File validation is the process to determine if a given file is prepared consistent with a set of guidelines or rules. In NeXus, there are several different types of files. First, of course, is the data file yet it can be provided in one of several forms: HDF4, HDF5, or XML. Specifications for data files are provided by one or (usually) more NeXus definition files (NXDL, for short). These NXDL files are written in XML and validated by the NXDL specification which is written in the XML Schema (XSD) language. Thus, automated file verification is available for data files, definition files, and the rules for definition files.

[23] http://www.schematron.com
[24] http://www.w3schools.com/xml
[25] http://www.w3schools.com/schema
[26] http://www.w3schools.com/xsl/

Validation of NeXus data files

Each NeXus data file can be validated against the NXDL rules. (The full suite of NXDL specifications is converted into Schematron rules by an XSLT transformation and then combined into a single file. It is not allowed to have a NeXus base class and also an application definition with the same name since one will override the other in the master Schematron file) The validation is done using Schematron and the `NXvalidate` program. Schematron was selected, rather than XML Schema (XSD), to permit established rules for NeXus files, especially the rule allowing the nodes within `NXentry` to appear in any order.

The validation process is mainly checking file structure (presence or absence of groups/fields) - it is usually impossible to check the actual data itself, other than confirm that it is of the correct data type (string, float etc.). The only exception is when the NXDL specification is either a fixed value or an enumeration - in which case the data is checked.

During validation, the NeXus data file instance (either HDF or XML) is first converted into an XML file in a form that facilitates validation (e.g with large numeric data removed). Then the XML file is validated by Schematron against the `schema/all.sch` file.

Validation of NeXus Definition Language (NXDL) specification files

Each NXDL file must be validated against the rules that define how NXDL files are to be arranged. The NXDL rules are specified in the form of XML Schema (XSD).

Standard tools (validating editor or command line or support library) can be used to validate any NXDL file. Here's an example using `xmllint` from a directory that contains `nxdl.xsd`, `nxdlTypes.xsd`, and `applications/NXsas.nxdl.xml`:

Use of `xmllint` to validate a NXDL specification.

```
xmllint --noout --schema nxdl.xsd applications/NXsas.nxdl.xml
```

Validation of the NXDL rules

NXDL rules are specified using the rules of XML Schema (XSD). The XSD syntax of the rules is validated using standard XML file validation tools: either a validating editor (such as *oXygen*, *xmlSpy*, or *eclipse*) or common UNIX/Linux command line tools

Use of `xmllint` to validate the NXDL rules.

```
xmllint --valid  nxdl.xsd
```

The validating editor method is used by the developers while the `xmllint` command line tool is the automated method used by the NeXus definitions subversion repository.

Validation of XSLT files

XSLT transformations are validated using standard tools such as a validating editor or xmllint.

Transformation of NXDL files to Schematron

Schematron [1] is a rule-based language that allows very specific validation of an XML document. Its advantages over using XSD schema are that:

- more specific pattern-based rules based on data content can be written
- full XSLT/XPath expression syntax available for writing validation tests
- error messages can be customised and thus more meaningful
- It is easier to validate documents when entities can occur in any order.

XSD does provide a mechanism for defining a class structure and inheritance, so its usage within NeXus in addition to schematron has not been ruled out. But for a basic validation of file content, schematron looks best.

The NXDL definition files are converted into a set of Schematron rules using the `xslt/nxdl2sch.xsl` XSLT stylesheet. The NeXus instance file (either in XML, HDF4, or HDF5) is turned into a reduced XML validation file. This file is very similar to a pure NeXus XML file, but with additional metadata for dimensions and also with most of the actual numeric data removed.

The validation process then compares the set of Schematron rules against the *reduced XML* validation file. Schematron itself is implemented as a set of XSLT transforms. NeXus includes the Schematron files, as well as the Java based XSLT engine `saxon`.

The java based `nxvalidate` GUI can be run to validate files.

Currently, the structure of the file is validated (i.e. valid names are used at the correct points), but this will be extended to array dimensions and link targets. Error messages are printed about missing mandatory fields, and informational messages are printed about fields that are neither optional or mandatory (in case they are a typing error). Even non-standard names must comply with a set of rules (e.g. no spaces are allowed in names). Enumerations are checked that they conform to an allowed value. The data type is checked and the units will also be checked.

2.10 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 (HDF4, HDF5, or XML).

2.10.1 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 From the source code documentation:

> "Utility to convert a NeXus file into HDF4/HDF5/XML/..."

Note: this command-line tool is different than the newer Java GUI program: `NXvalidate`.

NXvalidate Java program (in development in 2010) to check any NeXus data file for conformance with the NeXus NXDL-based standard. Note: This Java GUI is different than the command-line tool: `nxvalidate`.

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.)

2.10.2 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.

GDA (http://www.opengda.org) The GDA project is an open-source framework for creating customised data acquisition and analysis software for science facilities such as neutron and X-ray sources.

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.

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.

NeXpy (http://trac.mcs.anl.gov/projects/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.

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.

2.10.3 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.

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.

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.

2.11 Frequently Asked Questions

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

1. 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.

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

   NeXus files are not per se binary. If you use the XML backend the data are stored in a relatively human readable form (see *this example*). This backend however is only recommended for very small data sets. With the multidimensional data that is routinely recorded on many modern instruments it is very difficult anyway to retrieve useful information on a VT100 terminal. 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.

3. 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. The XML backend is available for special use cases. Choose this option with care considering the space and speed implications.

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

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

5. 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.

6. I want to produce an application definition. How do I go about it?

   Read the NXDL Tutorial in *Creating a NXDL Specification* (page 52) 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. [27]

7. What is the purpose of `NXdata`?

   `NXdata` contains links to the data stored elsewhere in the `NXentry`. It identifies the default plottable data. This is one of the basic motivations (see *Simple plotting* (page 15)) for the NeXus standard. The choice of the name `NXdata` is historic and does not really reflect its function.

8. How do I identify the plottable data?

   See the section: *Find the plottable data* (page 43).

9. How can I specify reasonable axes for my data?

   See the section: *Linking Multi Dimensional Data with Axis Data* (page 41).

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

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

11. Specifications are complicated and often provide too much information for what I need. Where can I find some good example data files?

[27] Refer to the most recent version of the NIAC constitution on the NIAC wiki: http://www.nexusformat.org/NIAC

There are a few checked into the *definitions repository*. At the moment the selection is quite limited and not very representative. This repository will be edited as more example files become available.

12. 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. You would be wise to consult the algorithms in the Java version of `NXvalidate` (see *NXvalidate-java*) for more details.

13. Why do I need to specify the `NAPItype`? My programming language does not need that information and I don't care about C and colleagues. Can I leave it out?

    `NAPItype` is necessary. When implementing the NeXus-XML API we strived to make this as general as HDF and reasonably efficient for medium sized datasets. This is why we store arrays as a large bunch of numbers in C-storage order. And we need the `NAPItype` to figure out the dimensions of the dataset.

14. 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 or XML data file is written. Validation of data files written without the NAPI is strongly encouraged.

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

    Note: NeXus uses HDF5 hard links

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

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

16. **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 *ClassDefinitions-Application*, 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.

CHAPTER

THREE

NEXUS: REFERENCE DOCUMENTATION

3.1 NAPI: NeXus Application Programmer Interface

3.1.1 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.

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

```
    // $Id: napi-java-prog1.java 1091 2012-05-28 21:10:09Z Pete Jemian $
    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

```
// $Id: napi-java-datarw1.java 1091 2012-05-28 21:10:09Z Pete Jemian $
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

```
// $Id: napi-java-datarw2.java 1091 2012-05-28 21:10:09Z Pete Jemian $
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

```
// $Id: napi-java-datarw2.java 1091 2012-05-28 21:10:09Z Pete Jemian $
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()`

```
/* $Id: frag-c-api-sig-getinfo.c 1091 2012-05-28 21:10:09Z Pete Jemian $ */
/* C -API */
NXstatus NXgetinfo(NXhandle handle, int *rank, int iDim[],
                 int *datatype);
```

Java API signature of `getinfo()`

```
// $Id: frag-c-api-sig-getinfo.java 1091 2012-05-28 21:10:09Z Pete Jemian $
// 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

```
// $Id: napi-java-inquiry1.java 1091 2012-05-28 21:10:09Z Pete Jemian $
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

```
// $Id: napi-java-inquiry2.java 1091 2012-05-28 21:10:09Z Pete Jemian $
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]
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.

Footnote

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`, `xmllint`, and `DocBook`. 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* (page 48) in Volume I, 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 through the TRAC site: http://trac.nexusformat.org/definitions/browser/trunk in the `base_classes`, `applications`, and `contributed` directories. The rules (expressed as XML

[1] http://download.nexusformat.org/doxygen/html-java/

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 [2] available to ensure that the files are *well-formed*. [3] 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* (page 90) 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* (page 90) 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.

3.2.2 Data Types allowed in NXDL specifications

Data Types for use in NXDL specifications 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.

3.2.3 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 section on *NeXus units* (page 41) in Volume I. The values for unit categories are restricted (by an enumeration) to the following table.

[2] For example *XML Copy Editor:xml-copy-editor.sourceforge.net*

[3] http://en.wikipedia.org/wiki/XML#Well-formedness_and_error-handling

3.2.4 Historical notes about the Development of NXDL

This section contains a few brief notes about the history of NXDL and the motivations for its creation.

Previously, the structure of NeXus data files was described using *Meta-DTD*, an XML format that provided a compact description. The terse format was not obvious to all and was difficult to machine-process. NXDL was conceived to be a simpler syntax than Meta-DTD. The switch to NXDL was not intended to change what was in the data files, just to provide an easier (and more generic) way of describing data files.

The NeXus Design page lists the group classes from which a NeXus file is constructed. They 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). When planning to include a particular piece of metadata, consult the class definitions to find out what to call it. 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.

As part of the NeXus standard, the NIAC identified a number of generic instruments that describe an appreciable number of existing instruments around the world. Although not identical in every detail, they share many common characteristics, and more importantly, they require sufficiently similar modes of data analysis, enough to make a standard description useful. Many of the application definitions were built from these instrument definitions using the NeXus Definition Language (NXDL) format.

Class definitions in NeXus prior to 2008 had been in the form of base classes and instrument definitions. All of these were in the same category. As the development of NeXus had been led mostly by scientists from neutron sources, this represented their typical situations.

Both those new to NeXus and also those familiar saw the previous emphasis on instrument definitions as a deficiency that limited flexibility and possibly usage. The point was made that NeXus should attempt to describe better reduced data and also data for analysis since synchrotron instruments are rarely adhering to a fixed definition.

The design of NeXus is moving towards an object-oriented approach where the base classes will be the objects and the application definitions will use the objects to specify the required components as fits some application. Here, *application* is very loosely defined to include:

- specification of a scientific instrument (example: TOF-USANS at SNS)
- specification of what is expected for a scientific technique (example: small-angle scattering data for common analysis programs)
- specification of generic data acquisition stream (example: TOFRAW - raw time-of-flight data from a pulsed neutron source)
- specification of input or output of a specific software program

The point of the *NeXus Application Definition* is that all of these start with `NX` and all have been approved by the NIAC.

Those NXDL specifications not yet approved by the NIAC fall into the category of *NeXus contributed definitions* for which NeXus has a place in the repository. Consider the NXDL files in the `contributed` directory as *in incubation*. This category is the place to put an NXDL (a candidate for a base class or application definition) for the NIAC to consider approving.

3.3 NeXus classes

Information is stored in a NeXus data file by grouping together similar parts. For example, information about the sample could include a descriptive name, the temperature, and other items. NeXus specifies the contents of these groupings using *classes*. In some parts of this manual, these classes might be called *group type* or some similar term. In this section, the NeXus classes are described in detail. Each class is specified using the *NeXus Definition Language* (NXDL). The rules and structure of NXDL are described in a separate chapter.

There are three types of NeXus class file: base classes, application definitions, and contributed definitions. 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. 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.

3.3.1 Overview of NeXus classes

Each of the NeXus classes is described in two basic ways. First, a short list of descriptive information is provided as a header, then a condensed listing of the basic structure, then a table providing documentation for the various components of the NeXus class.

category The category of NXDL, either: + `base` (base class) + `application` (application definition) + `contributed` (contributed definition)

NXDL source Name of the NeXus class and a URL to the source listing in the NeXus subversion repository.

version A string that documents this particular version of this NXDL.

SVN Id Subversion repository checkout identification, stripped of the surrounding dollar signs. (The *Id* is blank on files copied direct from the repository that are not checked out by a subversion client.)

NeXus Definition Language The *NeXus Definition Language* (page 87) (NXDL) (described in *NXDL*) is used to describe the components in the NeXus Base Classes, as well as application and contributed definitions. The intent of NXDL is to provide a rules-based method for defining a NeXus data file that is specific to either an instrument (where NeXus has been for years) or an area of scientific technique or analysis. NXDL replaces the meta-DTD method used previously to define the NeXus base classes.

extends class NeXus class extended by this class. Most NeXus base classes only extend the base class definition (NXDL).

other classes included List (including URLs) of other classes used to define this class.

symbol list List of the `symbols` (if present) that define mnemonics that represent the length of each dimension in a vector or array.

documentation Description of the NeXus class. DocBook markup (formatting is allowed).

Basic structure of the `class`\`

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 while links use the characters `-->` to represent the path to the intended source of the information.

The table has columns to describe the 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

In the above example, the fields might appear in a NeXus XML data file as

Example fragment of a NeXus XML data file

```
<program_name version="1.0a" configuration="standard">
    MaxSAS
</program_name>
<NXnote name="thumbnail" mime_type="image/*">
    <!-- contents of an NXnote would appear here -->
</NXnote>
<distance units="mm">125.6</distance>
<mode> Single Bunch </mode>
<target_material>depleted_U</target_material>
```

The columns in the table are described as follows:

Name (and attributes) Name of the data 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).

Type Type of data to be represented by this variable. The type is one of those specified in the *NeXus Definition Language* (page 87) (see *NXDL*). 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.

Units Data units, given as character strings, must conform to the NeXus units standard. See the *"NeXus units"* (page 41) section for details.

Description (and Occurrences) A simple text description of the data 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

3.4 Examples of writing and reading NeXus data files

Simple examples of reading and writing NeXus data files are provided in the *NeXus Introduction* (page 7) chapter of Volume I and also in the *NAPI: NeXus Application Programmer Interface* (page 81) chapter of Volume II. Here, three examples are provided showing how to write a NeXus data file without using the NAPI.

3.4.1 Code Examples that use the NAPI

Various examples are given that show how to read and write NeXus data files using the *NAPI: NeXus Application Programmer Interface* (page 81).

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 *native-HDF5-Examples*.

NAPI C Example: write simple NeXus file

```c
/* $Id: napi-example.c 1091 2012-05-28 21:10:09Z Pete Jemian $ */

#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.

```
! $Id: napi-example.f77 552 2010-04-19 22:24:42Z Pete Jemian $

      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

```
! $Id: napi-example.f90 552 2010-04-19 22:24:42Z Pete Jemian $

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 file may be retrieved from the repository of NeXus data file examples:

data http://svn.nexusformat.org/definitions/exampledata/simple3D.h5

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

nf = nxs.open("simple3D.h5", "w5")

nf.makegroup("entry","NXentry")
nf.opengroup("entry","NXentry")

nf.makegroup("data","NXdata")
nf.opengroup("data","NXdata")

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.makedata("test",'int32',[2,3,4])
nf.opendata("test")
nf.putdata(a)
nf.putattr("signal",1)
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`) that we describe in a later section (*h5toText support module* (page 123)) of this chapter. 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]
            ]
          ]
```

3.4.2 Code Examples that do not use the NAPI

Sometimes, for whatever reason, it is necessary to write or read NeXus files without using the routines provided by the *NAPI: NeXus Application Programmer Interface* (page 81). Each example in this section is written to support just one of the low-level file formats supported by NeXus (HDF4, HDF5, or XML).

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* (page 7) chapter. Compare these code examples with *NAPI-Examples*.

Writing a simple NeXus file using native HDF5 commands in C

```
/**
 * 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]);
  }

}
```

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 [4] (using `h5py` calls directly rather than using the NeXus NAPI). The actual data to be written was extracted (elsewhere) from a `spec` [5] 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 in *Example-H5py-Data* 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 to Volume I* (page 10)) except that the names will be different, as shown below:

Figure 3.1: data structure, (from Introduction)

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

our h5py example

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

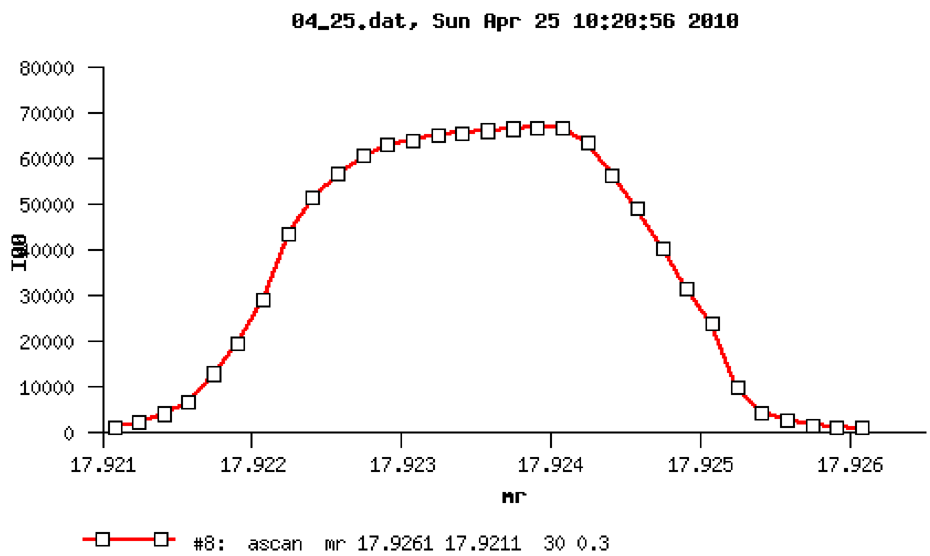


Figure 3.2: plot of our *mr_scan*

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
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.

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 *Data Groups* (page 21) and datasets (*Data Fields* (page 22)) are created, and how *Data Attributes* (page 22) are assigned. Note particularly the `NX_class` attribute on each HDF5 group that describes which of the NeXus *ClassDefinitions-Base* 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.

```python
#!/usr/bin/env python
'''
Writes the simplest NeXus HDF5 file using h5py
according to the example from Figure 1.3
in the Introduction chapter
'''

import h5py
import numpy

INPUT_FILE = 'input.dat'
```

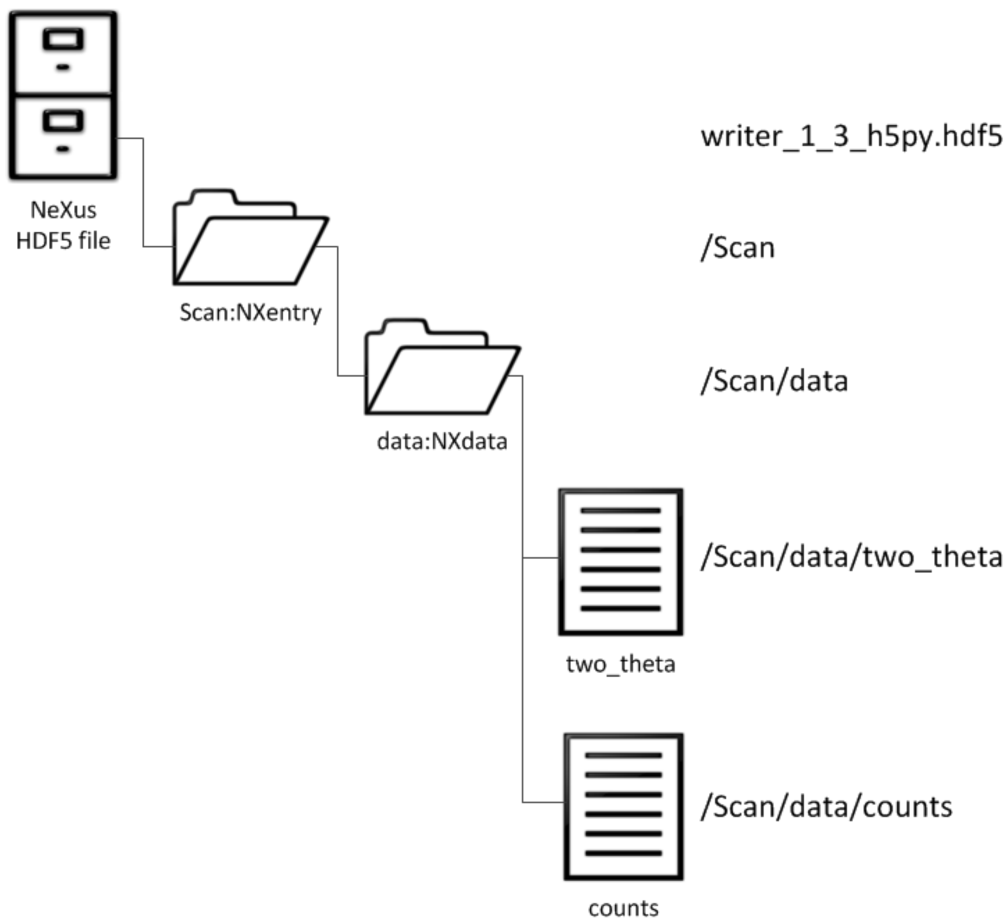


Figure 3.3: Simple Example

```
HDF5_FILE = 'writer_1_3_h5py.hdf5'

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

tthData, countsData = numpy.loadtxt(INPUT_FILE).T

f = h5py.File(HDF5_FILE, "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'

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

counts = nxdata.create_dataset("counts", data=countsData)
counts.attrs['units'] = "counts"
counts.attrs['signal'] = "1"
counts.attrs['axes'] = "two_theta"

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

We wish to make things a bit simpler for ourselves when creating the common structures we use in our data files. To help, we gather together some of the common concepts such as *create a file*, *create a NeXus group*, *create a dataset* and start to build a helper library. (See *mylib support module* (page 121) for more details.) Here, we call it `my_lib`. Applying it to the simple example above, our code only becomes a couple lines shorter! (Let's hope the library starts to help in larger or more complicated projects.) Here's the revision that replaces direct calls to `numpy` and `h5py` with calls to our library. It generates the file `writer_1_3.hdf5`.

```
#!/usr/bin/env python
'''
Writes the simplest NeXus HDF5 file using
a simple helper library with h5py and numpy calls
according to the example from Figure 1.3
in the Introduction chapter
'''

import my_lib

INPUT_FILE = 'input.dat'
HDF5_FILE = 'writer_1_3.hdf5'

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

tthData, countsData = my_lib.get_2column_data(INPUT_FILE)

f = my_lib.makeFile(HDF5_FILE)
# since this is a simple example, no attributes are used at this point

```

```
nxentry = my_lib.makeGroup(f, 'Scan', 'NXentry')
nxdata = my_lib.makeGroup(nxentry, 'data', 'NXdata')

my_lib.makeDataset(nxdata, "two_theta", tthData, units='degrees')
my_lib.makeDataset(nxdata, "counts", countsData,
                   units='counts', signal='1', axes='two_theta')

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.

```
HDF5 "writer_1_3.hdf5" {
GROUP "/" {
   GROUP "Scan" {
      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 "data" {
         ATTRIBUTE "NX_class" {
            DATATYPE  H5T_STRING {
                  STRSIZE 6;
                  STRPAD H5T_STR_NULLPAD;
                  CSET H5T_CSET_ASCII;
                  CTYPE H5T_C_S1;
               }
            DATASPACE  SCALAR
            DATA {
            (0): "NXdata"
            }
         }
         DATASET "counts" {
            DATATYPE  H5T_STD_I32LE
            DATASPACE  SIMPLE { ( 31 ) / ( 31 ) }
            DATA {
```

```
            (0): 1037, 1318, 1704, 2857, 4516, 9998, 23819, 31662, 40458,
            (9): 49087, 56514, 63499, 66802, 66863, 66599, 66206, 65747,
            (17): 65250, 64129, 63044, 60796, 56795, 51550, 43710, 29315,
            (25): 19782, 12992, 6622, 4198, 2248, 1321
            }
            ATTRIBUTE "units" {
               DATATYPE  H5T_STRING {
                     STRSIZE 6;
                     STRPAD H5T_STR_NULLPAD;
                     CSET H5T_CSET_ASCII;
                     CTYPE H5T_C_S1;
                  }
               DATASPACE  SCALAR
               DATA {
               (0): "counts"
               }
            }
            ATTRIBUTE "signal" {
               DATATYPE  H5T_STRING {
                     STRSIZE 1;
                     STRPAD H5T_STR_NULLPAD;
                     CSET H5T_CSET_ASCII;
                     CTYPE H5T_C_S1;
                  }
               DATASPACE  SCALAR
               DATA {
               (0): "1"
               }
            }
            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"
               }
            }
         }
         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"
               }
            }
         }
      }
   }
}
}
```

Since the output of `h5dump` is verbose, a tool (see *h5toText support module* (page 123)) was created to print out the structure of HDF5 data files. This tool provides a simplified view of the NeXus file. It is run with a command like this: `python h5toText.py h5dump writer_1_3.hdf5`. Here is the output:

```
writer_1_3.hdf5:NeXus data file
  Scan:NXentry
    @NX_class = NXentry
    data:NXdata
      @NX_class = NXdata
      counts:NX_INT32[31] = __array
        @units = counts
        @signal = 1
        @axes = two_theta
        __array = [1037, 1318, 1704, '...', 1321]
      two_theta:NX_FLOAT64[31] = __array
        @units = degrees
        __array = [17.926079999999999, 17.925909999999998, 17.925750000000001, '...', 17.92
```

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

`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* (page 23) in the *NeXus Design* (page 21) chapter of the NeXus User Manual.) The *NeXus Design* (page 21) chapter provides a figure (*Linking in a NeXus file* (page 23)) 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.

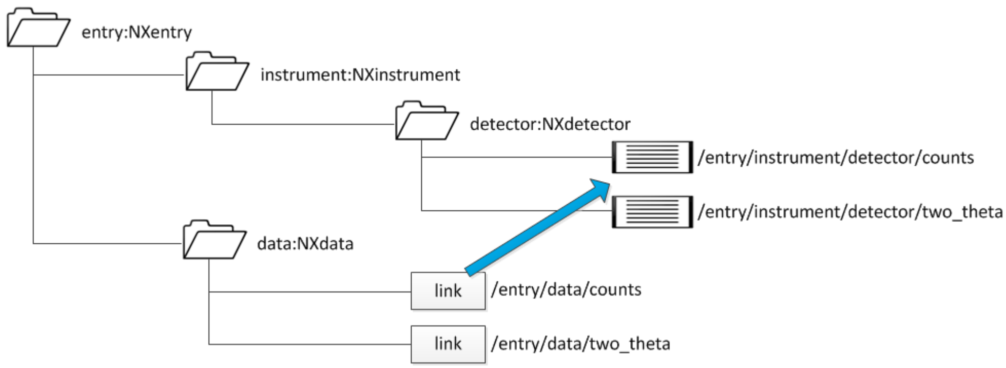


Figure 3.4: h5py example showing linking in a NeXus file

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 my_lib

INPUT_FILE = 'input.dat'
HDF5_FILE = 'writer_2_1.hdf5'

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

tthData, countsData = my_lib.get_2column_data(INPUT_FILE)

f = my_lib.makeFile(HDF5_FILE)   # create the HDF5 NeXus file

nxentry = my_lib.makeGroup(f, 'entry', 'NXentry')
nxinstrument = my_lib.makeGroup(nxentry, 'instrument', 'NXinstrument')
nxdetector = my_lib.makeGroup(nxinstrument, 'detector', 'NXdetector')

tth = my_lib.makeDataset(nxdetector, "two_theta", tthData, units='degrees')
counts = my_lib.makeDataset(nxdetector, "counts", countsData,
                   units='counts', signal='1', axes='two_theta')

nxdata = my_lib.makeGroup(nxentry, 'data', 'NXdata')
my_lib.makeLink(nxdetector, tth, nxdata.name+'/two_theta')
my_lib.makeLink(nxdetector, counts, nxdata.name+'/counts')

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.

```
HDF5 "writer_2_1.hdf5" {
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 "data" {
         ATTRIBUTE "NX_class" {
            DATATYPE  H5T_STRING {
                  STRSIZE 6;
                  STRPAD H5T_STR_NULLPAD;
                  CSET H5T_CSET_ASCII;
                  CTYPE H5T_C_S1;
               }
            DATASPACE  SCALAR
            DATA {
            (0): "NXdata"
            }
         }
         DATASET "counts" {
            DATATYPE  H5T_STD_I32LE
            DATASPACE  SIMPLE { ( 31 ) / ( 31 ) }
            DATA {
            (0): 1037, 1318, 1704, 2857, 4516, 9998, 23819, 31662, 40458,
            (9): 49087, 56514, 63499, 66802, 66863, 66599, 66206, 65747,
            (17): 65250, 64129, 63044, 60796, 56795, 51550, 43710, 29315,
            (25): 19782, 12992, 6622, 4198, 2248, 1321
            }
            ATTRIBUTE "units" {
               DATATYPE  H5T_STRING {
                     STRSIZE 6;
                     STRPAD H5T_STR_NULLPAD;
                     CSET H5T_CSET_ASCII;
                     CTYPE H5T_C_S1;
                  }
               DATASPACE  SCALAR
               DATA {
               (0): "counts"
               }
            }
            ATTRIBUTE "signal" {
               DATATYPE  H5T_STRING {
                     STRSIZE 1;
                     STRPAD H5T_STR_NULLPAD;
```

```
                        CSET H5T_CSET_ASCII;
                        CTYPE H5T_C_S1;
                     }
                  DATASPACE  SCALAR
                  DATA {
                  (0): "1"
                  }
               }
               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"
                  }
               }
               ATTRIBUTE "target" {
                  DATATYPE  H5T_STRING {
                        STRSIZE 33;
                        STRPAD H5T_STR_NULLPAD;
                        CSET H5T_CSET_ASCII;
                        CTYPE H5T_C_S1;
                     }
                  DATASPACE  SCALAR
                  DATA {
                  (0): "/entry/instrument/detector/counts"
                  }
               }
            }
            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"
               }
            }
            ATTRIBUTE "target" {
               DATATYPE  H5T_STRING {
                     STRSIZE 36;
                     STRPAD H5T_STR_NULLPAD;
                     CSET H5T_CSET_ASCII;
                     CTYPE H5T_C_S1;
                  }
               DATASPACE  SCALAR
               DATA {
               (0): "/entry/instrument/detector/two_theta"
               }
            }
         }
      }
      GROUP "instrument" {
         ATTRIBUTE "NX_class" {
            DATATYPE  H5T_STRING {
                  STRSIZE 12;
                  STRPAD H5T_STR_NULLPAD;
                  CSET H5T_CSET_ASCII;
                  CTYPE H5T_C_S1;
               }
            DATASPACE  SCALAR
            DATA {
            (0): "NXinstrument"
            }
         }
         GROUP "detector" {
            ATTRIBUTE "NX_class" {
               DATATYPE  H5T_STRING {
                     STRSIZE 10;
                     STRPAD H5T_STR_NULLPAD;
                     CSET H5T_CSET_ASCII;
                     CTYPE H5T_C_S1;
                  }
               DATASPACE  SCALAR
               DATA {
               (0): "NXdetector"
               }
            }
            DATASET "counts" {
               HARDLINK "/entry/data/counts"
            }
            DATASET "two_theta" {
               HARDLINK "/entry/data/two_theta"
            }
         }
      }
   }
}
```

```
}
```

Look carefully! It *appears* from 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:NeXus data file
  entry:NXentry
    @NX_class = NXentry
    data:NXdata
      @NX_class = NXdata
      counts --> /entry/instrument/detector/counts
      two_theta --> /entry/instrument/detector/two_theta
    instrument:NXinstrument
      @NX_class = NXinstrument
      detector:NXdetector
        @NX_class = NXdetector
        counts:NX_INT32[31] = __array
          @units = counts
          @signal = 1
          @axes = two_theta
          @target = /entry/instrument/detector/counts
          __array = [1037, 1318, 1704, '...', 1321]
        two_theta:NX_FLOAT64[31] = __array
          @units = degrees
          @target = /entry/instrument/detector/two_theta
          __array = [17.926079999999999, 17.925909999999998, 17.925750000000001, '...', 17
```

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

Writing the HDF5 file In the main code section of *BasicWriter.py*, a current time stamp is written in the format of *ISO 8601*. 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 `datatime` 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 our `my_lib` support 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/`. First, our support library calls `nxentry = f.create_group("entry")` to create the `NXentry` group called `entry` at the root level. 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="1"` as an attribute on `I00`, NeXus recognizes `I00` as the default *y* axis for the plot. The `axes="mr"` 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
import my_lib  # uses h5py

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 = my_lib.makeFile(fileName, file_name=fileName,
        file_time=timestamp,
        instrument="APS USAXS at 32ID-B",
        creator="$Id: BasicWriter.py 1091 2012-05-28 21:10:09Z Pete Jemian $",
        NeXus_version="4.3.0",
        HDF5_Version=h5py.version.hdf5_version,
        h5py_version=h5py.version.version)

nxentry = my_lib.makeGroup(f, "entry", "NXentry")
my_lib.makeDataset(nxentry, 'title', data='1-D scan of I00 v. mr')

```

```
nxdata = my_lib.makeGroup(nxentry, "mr_scan", "NXdata")

my_lib.makeDataset(nxdata,  "mr",  mr_arr, units='degrees', long_name='USAXS mr (degrees)'

my_lib.makeDataset(nxdata,  "I00",  i00_arr, units='counts',
       signal='1',     # Y axis of default plot
       axes='mr',          # name "mr" as X axis
       long_name='USAXS I00 (counts)')

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

print "wrote file:", fileName
```

Reading the HDF5 file 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 in *Example-H5py-Output*.

Output from `BasicReader.py`

```
file_name: prj_test.nexus.hdf5
file_time: 2010-10-18T17:17:04-0500
creator: $Id: BasicWriter.py 647 2010-10-19 22:34:01Z Pete Jemian $
```

```
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
```

Validating the HDF5 file Now we have an HDF5 file that contains our data. What makes this different from a NeXus data file? A NeXus file has a specific arrangement of groups and datasets in an HDF5 file.

To test that our HDF5 file conforms to the NeXus standard, we use the *NXvalidate-java* program. Referring to the next figure, we compare our HDF5 file with the rules for generic [6] data files (`all.nxdl.xml`). The only items that have been flagged are the "non-standard field names" *mr* and *I00*. Neither of these two names is specifically named in the NeXus NXDL definition for the `NXdata` base class. As we'll see shortly, this is not a problem.

Note: Note that `NXvalidate` shows only the first data field for *mr* and *I00*.

---

[6] generic NeXus data files: NeXus data files for which no application-specific NXDL applies

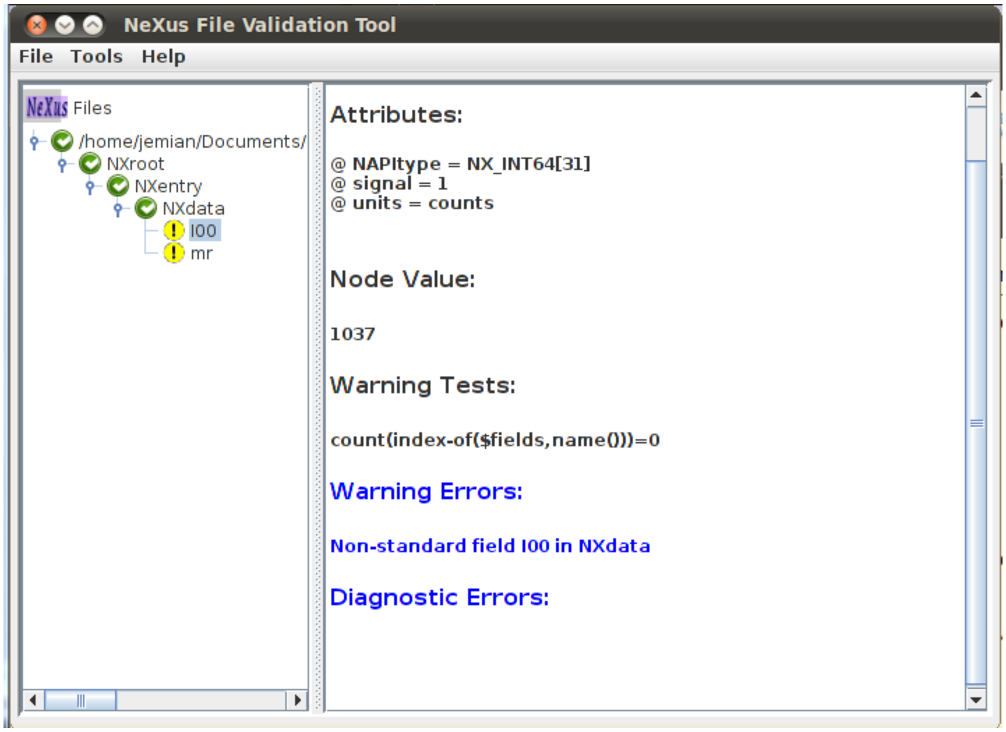


Figure 3.5: NeXus validation of our HDF5 file

Plotting the HDF5 file Now that we are certain our file conforms to the NeXus standard, let's plot it using the `NeXpy` [7] 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* (page 106) 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 by default.

Python Helper Modules for `h5py` Examples

Two additional Python modules were used to describe these `h5py` examples. The source code for each is given here. The first is a library we wrote that helps us create standard NeXus components using `h5py`. The second is a tool that helps us inspect the content and structure of HDF5 files.

`mylib` support module The examples in this section make use of a small helper library that calls `h5py` to create the various NeXus data components of *Data Groups* (page 21), *Data Fields* (page 22), *Data Attributes* (page 22), and *Links* (page 23). In a smaller sense, this subroutine library (`my_lib`) fills the role of the NAPI for writing the data using h5py.

```
#!/usr/bin/env python
'''
my_lib Library of routines to support NeXus HDF5 files using h5py
'''
```

[7] *NeXpy*: http://trac.mcs.anl.gov/projects/nexpy

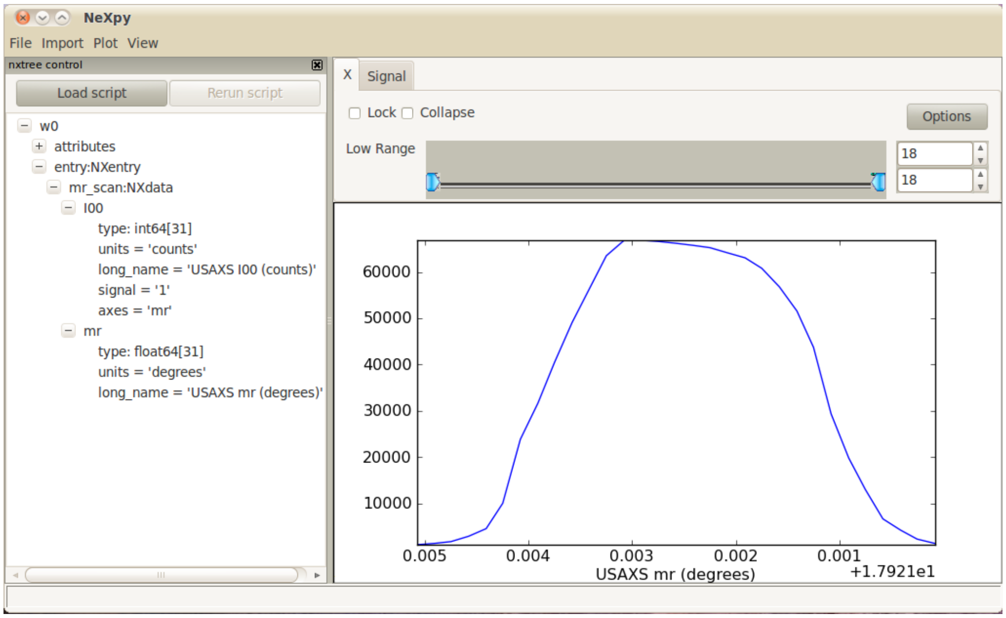


Figure 3.6: plot of our *mr_scan* using NeXpy

```python

import h5py    # HDF5 support
import numpy   # in this case, provides data structures

def makeFile(filename, **attr):
    """
    create and open an empty NeXus HDF5 file using h5py

    Any named parameters in the call to this method will be saved as
    attributes of the root of the file.
    Note that **attr is a dictionary of named parameters.

    :param str filename: valid file name
    :param attr: optional keywords of attributes
    :return: h5py file object
    """
    f = h5py.File(filename, "w")
    add_attributes(f, attr)
    return f

def makeGroup(parent, name, nxclass):
    """
    create a NeXus group

    :param obj parent: parent group
    :param str name: valid NeXus group name
    :param str nxclass: valid NeXus class name
    :return: h5py group object
    """
    group = parent.create_group(name)
```

```
    group.attrs["NX_class"] = nxclass
    return group

def makeDataset(parent, name, data = None, **attr):
    '''
    create and write data to a dataset in the HDF5 file hierarchy

    :param obj parent: parent group
    :param str name: valid NeXus dataset name
    :param obj data: the data to be saved
    :param attr: optional keywords of attributes
    '''
    if data == None:
        obj = parent.create_dataset(name)
    else:
        obj = parent.create_dataset(name, data=data)
    add_attributes(obj, attr)
    return obj

def makeLink(parent, sourceObject, targetName):
    """
    create a NeXus link in an HDF5 file.

    :param obj parent: parent group of source
    :param obj sourceObject: HDF5 object
    :param str targetName: HDF5 node path string, such as /entry/data/data
    """
    if not 'target' in sourceObject.attrs:
        # NeXus link, NOT an HDF5 link!
        sourceObject.attrs["target"] = str(sourceObject.name)
    parent._id.link(sourceObject.name, targetName, h5py.h5g.LINK_HARD)

def add_attributes(parent, attr):
    """
    add attributes to an h5py data item

    :param obj parent: h5py parent object
    :param dict attr: dictionary of attributes
    """
    if attr and type(attr) == type({}):
        # attr is a dictionary of attributes
        for k, v in attr.items():
            parent.attrs[k] = v

def get_2column_data(fileName):
    '''read two-column data from a file, first column is float, second column is integer''
    buffer = numpy.loadtxt(fileName).T
    xArr = buffer[0]
    yArr = numpy.asarray(buffer[1],'int32')
    return xArr, yArr
```

h5toText support module The module `h5toText` reads an HDF5 data file and prints out the structure of the groups, datasets, attributes, and links in that file. There is a command-line option to print out more or less of the data in the dataset arrays.

```
#!/usr/bin/env python

'''
Print the structure of an HDF5 file to stdout

$Id: h5toText.py 1091 2012-05-28 21:10:09Z Pete Jemian $
'''


########### SVN repository information ###################
# $Date: 2012-05-28 23:10:09 +0200 (Mo, 28. Mai 2012) $
# $Author: Pete Jemian $
# $Revision: 1091 $
# $URL: https://svn.nexusformat.org/definitions/branches/docbook2sphinx/manual/source/examp
# $Id: h5toText.py 1091 2012-05-28 21:10:09Z Pete Jemian $
########### SVN repository information ###################


import h5py
import os
import sys
import getopt


class H5toText(object):
    '''
    Example usage showing default display::

        mc = H5toText(filename)
        mc.array_items_shown = 5
        mc.report()
    '''
    filename = None
    requested_filename = None
    isNeXus = False
    array_items_shown = 5

    def __init__(self, filename, makeReport = False):
        ''' Constructor '''
        self.requested_filename = filename
        if os.path.exists(filename):
            self.filename = filename
            self.isNeXus = self.testIsNeXus()
            if makeReport:
                self.report()

    def report(self):
        ''' reporter '''
        if self.filename == None: return
        f = h5py.File(self.filename, 'r')
```

```
        txt = self.filename
        if self.isNeXus:
            txt += ":NeXus data file"
        self.showGroup(f, txt, indentation = "")
        f.close()

    def testIsNeXus(self):
        ''' test if the selected HDF5 file is a NeXus file '''
        result = False
        try:
            f = h5py.File(self.filename, 'r')
            for value in f.itervalues():
                if str(type(value)) in ("<class 'h5py.highlevel.Group'>"):
                    if 'NX_class' in value.attrs:
                        v = value.attrs['NX_class']
                        if type(v) == type("a string"):
                            if v == 'NXentry':
                                result = True
                                break
            f.close()
        except:
            pass
        return result

    def showGroup(self, obj, name, indentation = "  "):
        '''print the contents of the group'''
        nxclass = ""
        if 'NX_class' in obj.attrs:
            class_attr = obj.attrs['NX_class']
            nxclass = ":" + str(class_attr)
        print indentation + name + nxclass
        self.showAttributes(obj, indentation)
        group_equivalents = (
        "<class 'h5py.highlevel.File'>",
        "<class 'h5py.highlevel.Group'>",
        "<class 'h5py._hl.group.Group'>",
    )
        # show datasets (and links) first
        for itemname in sorted(obj):
            value = obj[itemname]
            if str(type(value)) not in group_equivalents:
                self.showDataset(value, itemname, indentation = indentation+"  ")
        # then show things that look like groups
        for itemname in sorted(obj):
            value = obj[itemname]
            if str(type(value)) in group_equivalents:
                self.showGroup(value, itemname, indentation = indentation+"  ")

    def showAttributes(self, obj, indentation = "  "):
        '''print any attributes'''
        for name, value in obj.attrs.iteritems():
            print "%s  @%s = %s" % (indentation, name, str(value))

```

```
    def showDataset(self, dset, name, indentation = "  "):
        '''print the contents and structure of a dataset'''
        shape = dset.shape
        if self.isNeXus:
            if "target" in dset.attrs:
                if dset.attrs['target'] != dset.name:
                    print "%s%s --> %s" % (indentation, name, dset.attrs['target'])
                    return
        txType = self.getType(dset)
        txShape = self.getShape(dset)
        if shape == (1,):
            value = " = %s" % str(dset[0])
            print "%s%s:%s%s%s" % (indentation, name, txType, txShape, value)
            self.showAttributes(dset, indentation)
        else:
        print "%s%s:%s%s = __array" % (indentation, name, txType, txShape)
            self.showAttributes(dset, indentation)  # show these before __array
            if self.array_items_shown > 2:
                value = self.formatArray(dset, indentation + '  ')
                print "%s  %s = %s" % (indentation, "__array", value)
            else:
                print "%s  %s: %s" % (indentation, "__array", "not shown")

    def getType(self, obj):
        ''' get the storage (data) type of the dataset '''
        t = str(obj.dtype)
        if t[0:2] == '|S':
            t = 'char[%s]' % t[2:]
        if self.isNeXus:
            t = 'NX_' + t.upper()
        return t

    def getShape(self, obj):
        ''' return the shape of the HDF5 dataset '''
        s = obj.shape
        l = []
        for dim in s:
            l.append(str(dim))
        if l == ['1']:
            result = ""
        else:
            result = "[%s]" % ",".join(l)
        return result

    def formatArray(self, obj, indentation = '  '):
        ''' nicely format an array up to rank=5 '''
        shape = obj.shape
        r = ""
        if len(shape) in (1, 2, 3, 4, 5):
            r = self.formatNdArray(obj, indentation + '  ')
        if len(shape) > 5:
            r = "### no arrays for rank > 5 ###"
        return r
```

```python

    def decideNumShown(self, n):
        ''' determine how many values to show '''
        if self.array_items_shown != None:
            if n > self.array_items_shown:
                n = self.array_items_shown - 2
        return n

    def formatNdArray(self, obj, indentation = '  '):
        ''' return a list of lower-dimension arrays, nicely formatted '''
        shape = obj.shape
        rank = len(shape)
        if not rank in (1, 2, 3, 4, 5): return None
        n = self.decideNumShown( shape[0] )
        r = []
        for i in range(n):
            if rank == 1: item = obj[i]
            if rank == 2: item = self.formatNdArray(obj[i, :])
            if rank == 3: item = self.formatNdArray(obj[i, :, :], indentation + '  ')
            if rank == 4: item = self.formatNdArray(obj[i, :, :, :], indentation + '  ')
            if rank == 5: item = self.formatNdArray(obj[i, :, :, :, :], indentation + '  ')
            r.append( item )
        if n < shape[0]:
            # skip over most
            r.append("...")
            #  get the last one
            if rank == 1: item = obj[-1]
            if rank == 2: item = self.formatNdArray(obj[-1, :])
            if rank == 3: item = self.formatNdArray(obj[-1, :, :], indentation + '  ')
            if rank == 4: item = self.formatNdArray(obj[-1, :, :, :], indentation + '  ')
            if rank == 5: item = self.formatNdArray(obj[-1, :, :, :, :], indentation + '  
            r.append( item )
        if rank == 1:
            s = str( r )
        else:
            s = "[\n" + indentation + '  '
            s += ("\n" + indentation + '  ').join(r)
            s += "\n" + indentation + "]"
        return s


if __name__ == '__main__':
    limit = 5
    filelist = []
    filelist.append('../Create/example1.hdf5')
    filelist.append('../Create/example2.hdf5')
    filelist.append('../Create/example3.hdf5')
    filelist.append('../Create/example4.hdf5')
    filelist.append('../../../NeXus/definitions/trunk/manual/examples/h5py/prj_test.nexus.h
    filelist.append('../../../NeXus/definitions/exampledata/code/hdf5/dmc01.h5')
    filelist.append('../../../NeXus/definitions/exampledata/code/hdf5/dmc02.h5')
    filelist.append('../../../NeXus/definitions/exampledata/code/hdf5/focus2007n001335.hdf
    filelist.append('../../../NeXus/definitions/exampledata/code/hdf5/NXtest.h5')
```

```python
    filelist.append('../../../NeXus/definitions/exampledata/code/hdf5/sans2009n012333.hdf')
    filelist.append('../Create/simple5.nxs')
    filelist.append('../Create/bad.h5')
    #filelist = []
    #filelist.append('testG.h5')
    #filelist.append('testG-pj.h5')
    if len(sys.argv) > 1:
        try:
            opts, args = getopt.getopt(sys.argv[1:], "n:")
        except:
            print
            print "SVN: $Id: h5toText.py 1091 2012-05-28 21:10:09Z Pete Jemian $"
            print "usage: ", sys.argv[0], " [-n ##] HDF5_file_name [another_HDF5_file_name]
            print "  -n ## : limit number of displayed array items to ## (must be 3 or more
            print
        for item in opts:
            if item[0] == "-n":
                if item[1].lower() == "none":
                    limit = None
                else:
                    limit = int(item[1])
        filelist = args
    for item in filelist:
        mc = H5toText(item)
        mc.array_items_shown = limit
        mc.report()
```

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.n
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.

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.

LRMECS `lrcs3701` data: 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

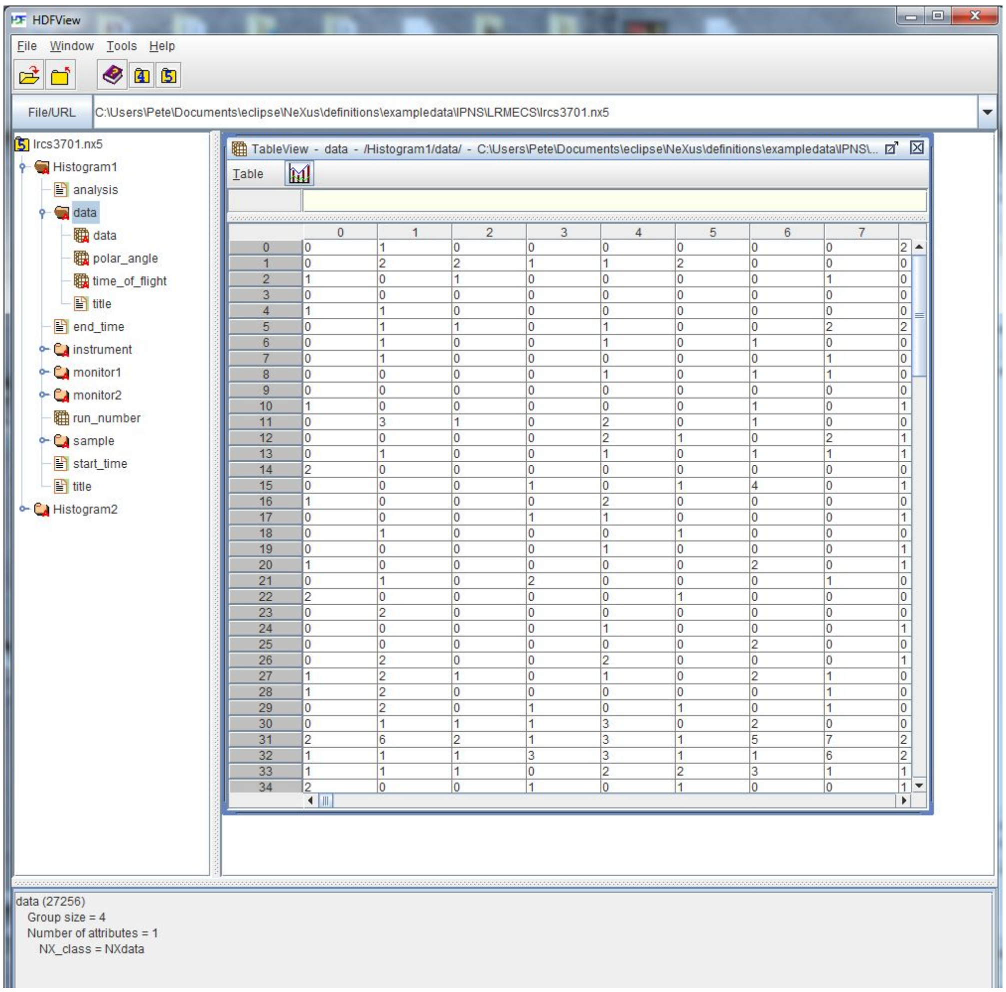


Figure 3.7: LRMECS `lrcs3701` data: *HDFview*

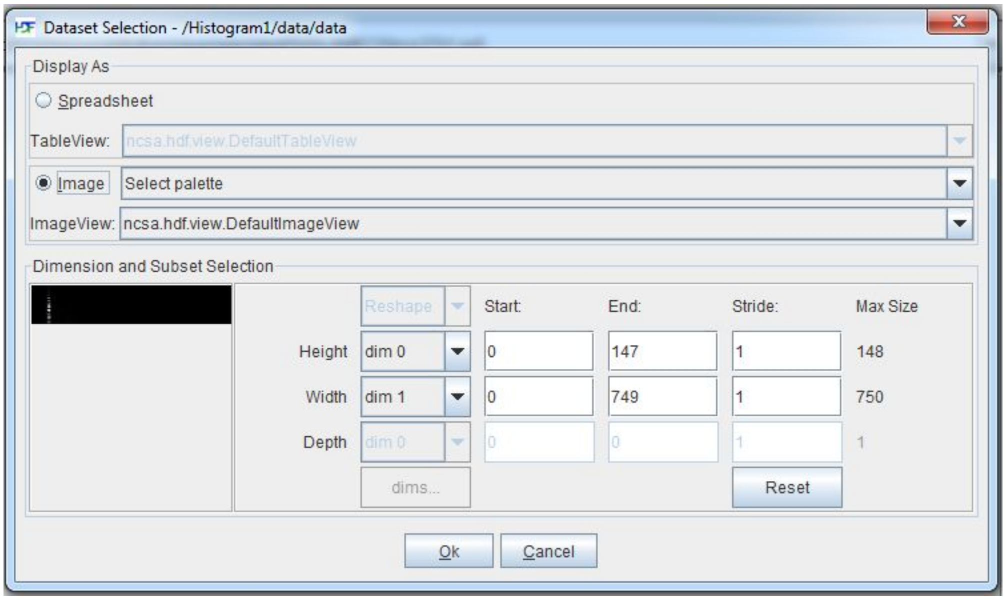


Figure 3.8: LRMECS `lrcs3701` data: *HDFview Open As* dialog

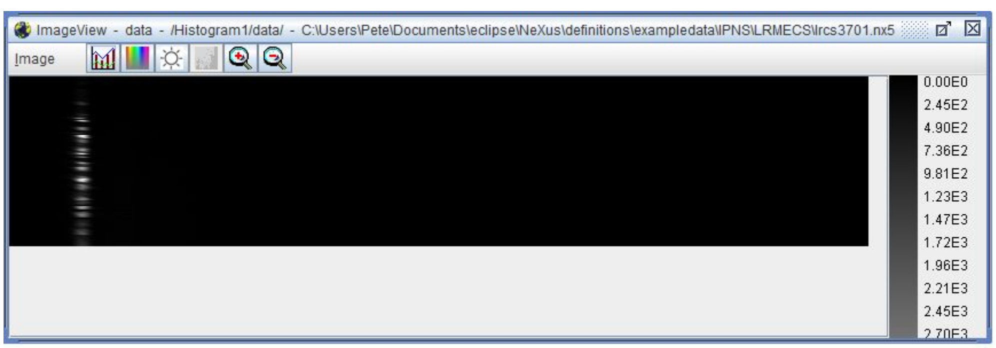


Figure 3.9: LRMECS `lrcs3701` data: *HDFview* Image

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.

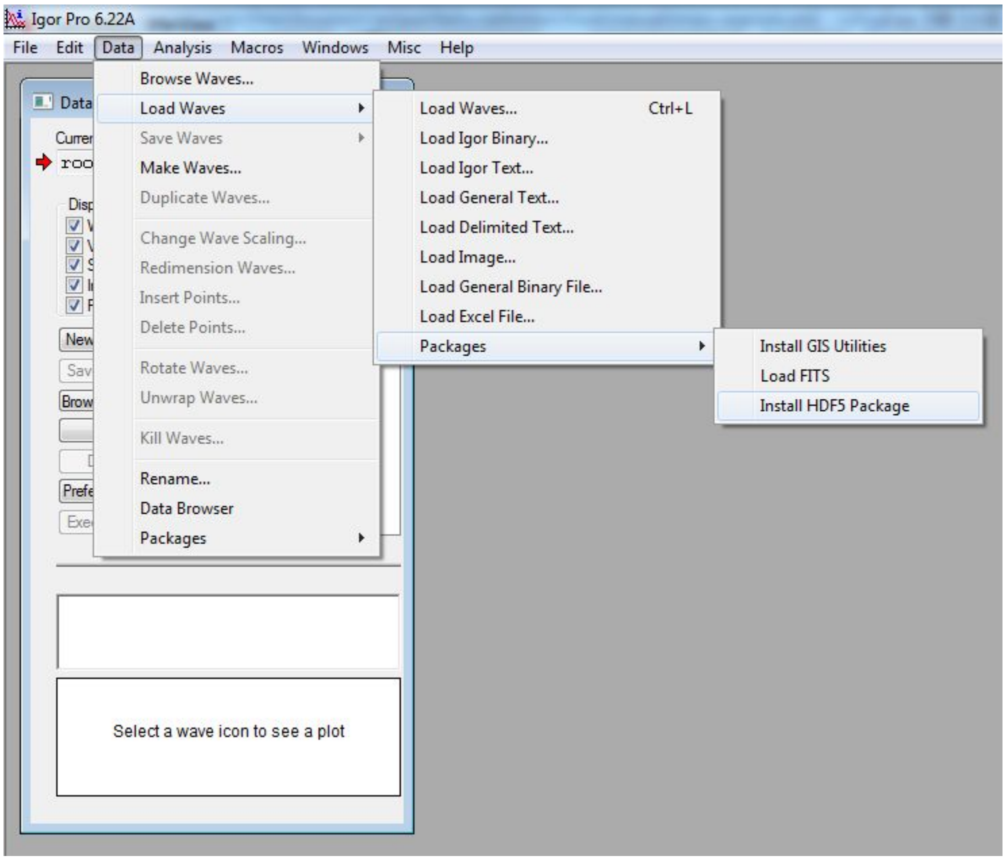


Figure 3.10: LRMECS `lrcs3701` data: *IgorPro* install HDF5 Browser

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

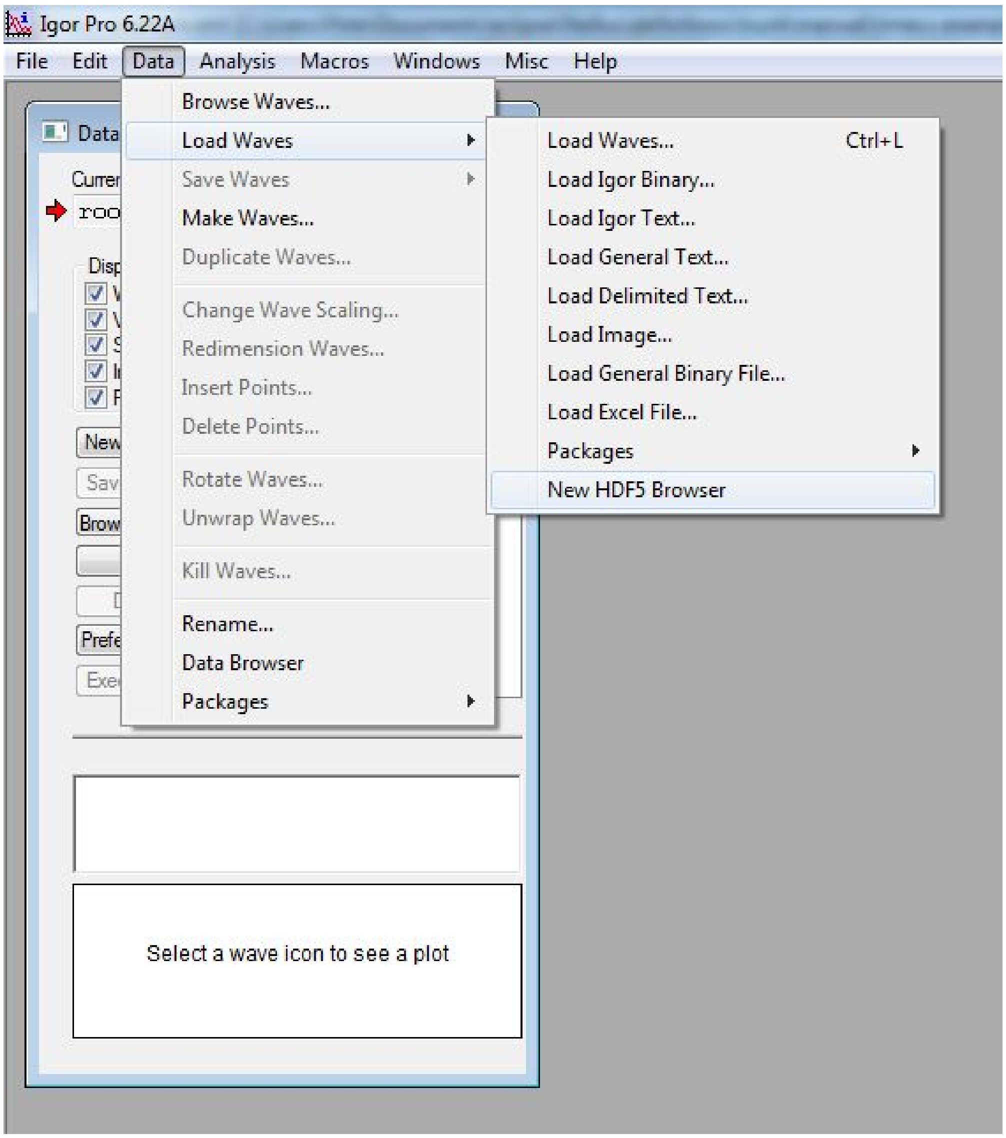


Figure 3.11: LRMECS `lrcs3701` data: *IgorPro HDFBrowser* dialog

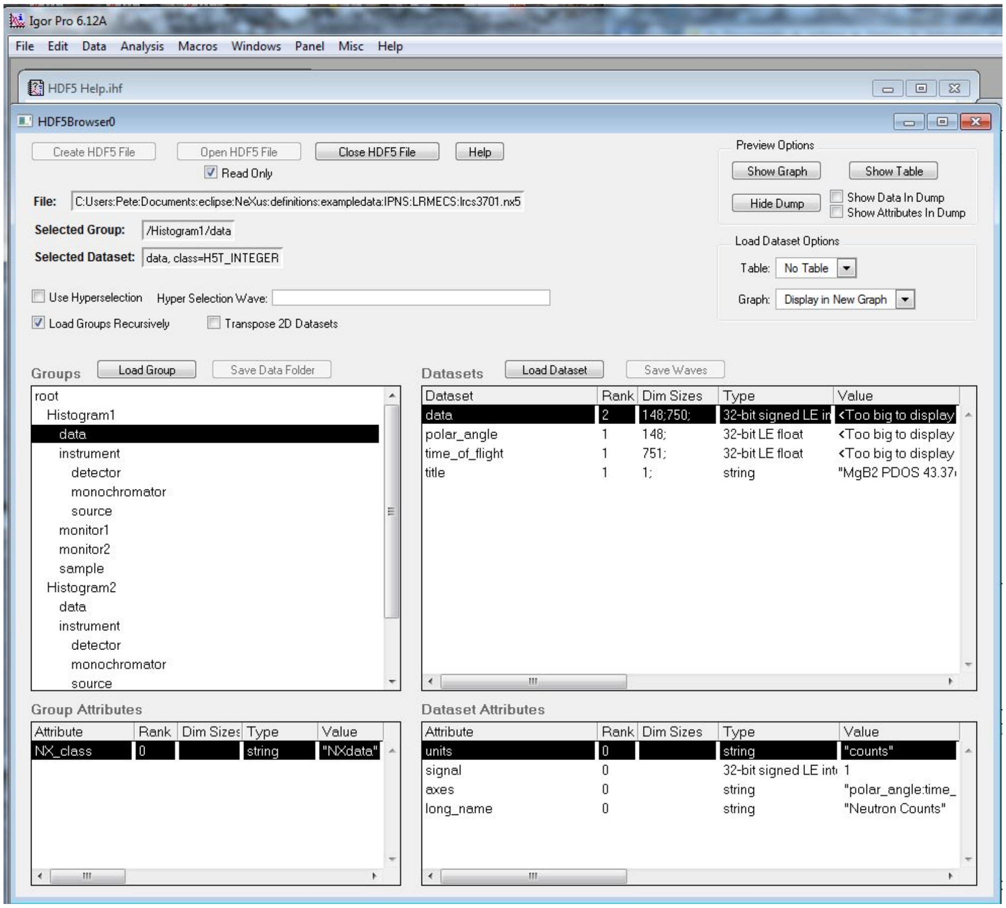


Figure 3.12: LRMECS `lrcs3701` data: *IgorPro HDFBrowser* dialog

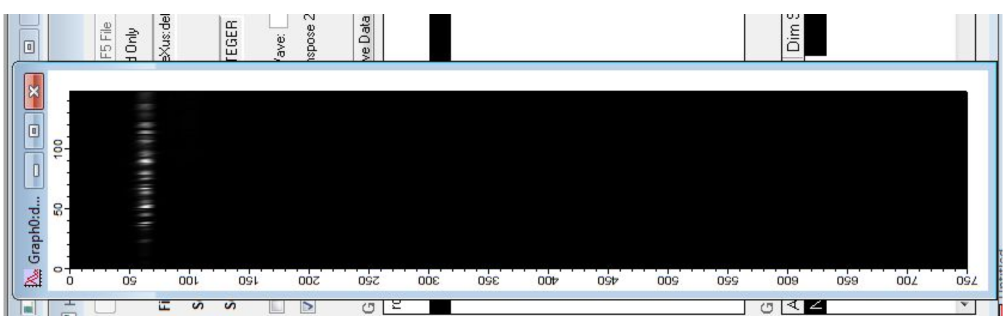


Figure 3.13: LRMECS `lrcs3701` data: *IgorPro* Image

CHAPTER

FOUR

AUTHORS

Ray Osborn `<rosborn@anl.gov>`, Argonne National Laboratory, Argonne, IL, USA,

Mark Könnecke Mark Könnecke, `<Mark.Koennecke@psi.ch>`, Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland,

Przemek Klosowski `<przemek.klosowski@nist.gov>`, U. of Maryland and NIST, Gaithersburg, MD, USA,

Frederick Akeroyd `<freddie.akeroyd@stfc.ac.uk>`, Rutherford Appleton Laboratory, Didcot, UK,

Peter F. Peterson `<petersonpf@ornl.gov>`, Spallation Neutron Source, Oak Ridge, TN, USA,

Pete Jemian `<jemian@anl.gov>`, Advanced Photon Source, Argonne, IL, USA,

Stuart I. Campbell `<campbellsi@ornl.gov>`, Oak Ridge National Laboratory, Oak Ridge, TN, USA,

Tobias Richter `<Tobias.Richter@diamond.ac.uk>`, Diamond Light Source Ltd., Didcot, UK

CHAPTER

FIVE

REVISION HISTORY

date	release	description	initials
2012-09		Documentation converted from DocBook to Sphinx	PRJ
2012-05	3.1	Ready for release.	PRJ
2012-02		Now using cmake to control multiplatform build and packaging.	PRJ
2011-11	1.0b	Preparing manual for initial release. Also preparing to convert manual source from DocBook to Sphinx for next release of manual.	PRJ
2010-11	draft	Nearly complete but still much finishing work remains. The description of dimensions and the description of the coordinate system needs major revision and improvement. More examples are needed. The manual is now divided into two volumes. Volume I is the User Manual, Volume II is the Reference Documentation. Much of the NXDL chapter in Volume II is autogenerated from the `nxdl.xsd` Schema and the NXDL source files. Initial release of NXDL, manual, and next release of NAPI (compatibility release) expected in mid-2011.	PRJ
2010 spring	initial draft	Most of the content from the old NeXus mediawiki documentation is included. Some new wiki content has been introduced but should be easy to identify for inclusion in the manual.	PRJ
2009-11		Started conversion from the old NeXus mediawiki documentation.	PFP

CHAPTER

SIX

LICENSES

The full texts of the software licenses governing this document (FDL) and the example in this document (LGPL) are provided in this appendix.

6.1 FDL: GNU Free Documentation License

```
                GNU Free Documentation License
                 Version 1.3, 3 November 2008


 Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
     <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other
functional and useful document "free" in the sense of freedom: to
assure everyone the effective freedom to copy and redistribute it,
with or without modifying it, either commercially or noncommercially.
Secondarily, this License preserves for the author and publisher a way
to get credit for their work, while not being considered responsible
for modifications made by others.

This License is a kind of "copyleft", which means that derivative
works of the document must themselves be free in the same sense.  It
complements the GNU General Public License, which is a copyleft
license designed for free software.

We have designed this License in order to use it for manuals for free
software, because free software needs free documentation: a free
program should come with manuals providing the same freedoms that the
software does.  But this License is not limited to software manuals;
it can be used for any textual work, regardless of subject matter or
whether it is published as a printed book.  We recommend this License
principally for works whose purpose is instruction or reference.
```

```

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that
contains a notice placed by the copyright holder saying it can be
distributed under the terms of this License.  Such a notice grants a
world-wide, royalty-free license, unlimited in duration, to use that
work under the conditions stated herein.  The "Document", below,
refers to any such manual or work.  Any member of the public is a
licensee, and is addressed as "you".  You accept the license if you
copy, modify or distribute the work in a way requiring permission
under copyright law.

A "Modified Version" of the Document means any work containing the
Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall
subject (or to related matters) and contains nothing that could fall
directly within that overall subject.  (Thus, if the Document is in
part a textbook of mathematics, a Secondary Section may not explain
any mathematics.)  The relationship could be a matter of historical
connection with the subject or with related matters, or of legal,
commercial, philosophical, ethical or political position regarding
them.

The "Invariant Sections" are certain Secondary Sections whose titles
are designated, as being those of Invariant Sections, in the notice
that says that the Document is released under this License.  If a
section does not fit the above definition of Secondary then it is not
allowed to be designated as Invariant.  The Document may contain zero
Invariant Sections.  If the Document does not identify any Invariant
Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed,
as Front-Cover Texts or Back-Cover Texts, in the notice that says that
the Document is released under this License.  A Front-Cover Text may
be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy,
represented in a format whose specification is available to the
general public, that is suitable for revising the document
straightforwardly with generic text editors or (for images composed of
pixels) generic paint programs or (for drawings) some widely available
drawing editor, and that is suitable for input to text formatters or
for automatic translation to a variety of formats suitable for input
to text formatters.  A copy made in an otherwise Transparent file
format whose markup, or absence of markup, has been arranged to thwart
or discourage subsequent modification by readers is not Transparent.
An image format is not Transparent if used for any substantial amount
of text.  A copy that is not "Transparent" is called "Opaque".
```

```
Examples of suitable formats for Transparent copies include plain
ASCII without markup, Texinfo input format, LaTeX input format, SGML
or XML using a publicly available DTD, and standard-conforming simple
HTML, PostScript or PDF designed for human modification.  Examples of
transparent image formats include PNG, XCF and JPG.  Opaque formats
include proprietary formats that can be read and edited only by
proprietary word processors, SGML or XML for which the DTD and/or
processing tools are not generally available, and the
machine-generated HTML, PostScript or PDF produced by some word
processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself,
plus such following pages as are needed to hold, legibly, the material
this License requires to appear in the title page.  For works in
formats which do not have any title page as such, "Title Page" means
the text near the most prominent appearance of the work's title,
preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of
the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose
title either is precisely XYZ or contains XYZ in parentheses following
text that translates XYZ in another language.  (Here XYZ stands for a
specific section name mentioned below, such as "Acknowledgements",
"Dedications", "Endorsements", or "History".)  To "Preserve the Title"
of such a section when you modify the Document means that it remains a
section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which
states that this License applies to the Document.  These Warranty
Disclaimers are considered to be included by reference in this
License, but only as regards disclaiming warranties: any other
implication that these Warranty Disclaimers may have is void and has
no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either
commercially or noncommercially, provided that this License, the
copyright notices, and the license notice saying this License applies
to the Document are reproduced in all copies, and that you add no
other conditions whatsoever to those of this License.  You may not use
technical measures to obstruct or control the reading or further
copying of the copies you make or distribute.  However, you may accept
compensation in exchange for copies.  If you distribute a large enough
number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and
you may publicly display copies.
```

```
3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have
printed covers) of the Document, numbering more than 100, and the
Document's license notice requires Cover Texts, you must enclose the
copies in covers that carry, clearly and legibly, all these Cover
Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on
the back cover.  Both covers must also clearly and legibly identify
you as the publisher of these copies.  The front cover must present
the full title with all words of the title equally prominent and
visible.  You may add other material on the covers in addition.
Copying with changes limited to the covers, as long as they preserve
the title of the Document and satisfy these conditions, can be treated
as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit
legibly, you should put the first ones listed (as many as fit
reasonably) on the actual cover, and continue the rest onto adjacent
pages.

If you publish or distribute Opaque copies of the Document numbering
more than 100, you must either include a machine-readable Transparent
copy along with each Opaque copy, or state in or with each Opaque copy
a computer-network location from which the general network-using
public has access to download using public-standard network protocols
a complete Transparent copy of the Document, free of added material.
If you use the latter option, you must take reasonably prudent steps,
when you begin distribution of Opaque copies in quantity, to ensure
that this Transparent copy will remain thus accessible at the stated
location until at least one year after the last time you distribute an
Opaque copy (directly or through your agents or retailers) of that
edition to the public.

It is requested, but not required, that you contact the authors of the
Document well before redistributing any large number of copies, to
give them a chance to provide you with an updated version of the
Document.


4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under
the conditions of sections 2 and 3 above, provided that you release
the Modified Version under precisely this License, with the Modified
Version filling the role of the Document, thus licensing distribution
and modification of the Modified Version to whoever possesses a copy
of it.  In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct
   from that of the Document, and from those of previous versions
   (which should, if there were any, be listed in the History section
   of the Document).  You may use the same title as a previous version
   if the original publisher of that version gives permission.
```

```
B. List on the Title Page, as authors, one or more persons or entities
   responsible for authorship of the modifications in the Modified
   Version, together with at least five of the principal authors of the
   Document (all of its principal authors, if it has fewer than five),
   unless they release you from this requirement.
C. State on the Title page the name of the publisher of the
   Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
   adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice
   giving the public permission to use the Modified Version under the
   terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections
   and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add
   to it an item stating at least the title, year, new authors, and
   publisher of the Modified Version as given on the Title Page.  If
   there is no section Entitled "History" in the Document, create one
   stating the title, year, authors, and publisher of the Document as
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on.  These may be placed in the "History" section.
   You may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",
   Preserve the Title of the section, and preserve in the section all
   the substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles.  Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements".  Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
   or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant.  To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
```

```
been approved by an organization as the authoritative definition of a
standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list
of Cover Texts in the Modified Version.  Only one passage of
Front-Cover Text and one of Back-Cover Text may be added by (or
through arrangements made by) any one entity.  If the Document already
includes a cover text for the same cover, previously added by you or
by arrangement made by the same entity you are acting on behalf of,
you may not add another; but you may replace the old one, on explicit
permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License
give permission to use their names for publicity for or to assert or
imply endorsement of any Modified Version.


5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this
License, under the terms defined in section 4 above for modified
versions, provided that you include in the combination all of the
Invariant Sections of all of the original documents, unmodified, and
list them all as Invariant Sections of your combined work in its
license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and
multiple identical Invariant Sections may be replaced with a single
copy.  If there are multiple Invariant Sections with the same name but
different contents, make the title of each such section unique by
adding at the end of it, in parentheses, the name of the original
author or publisher of that section if known, or else a unique number.
Make the same adjustment to the section titles in the list of
Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History"
in the various original documents, forming one section Entitled
"History"; likewise combine any sections Entitled "Acknowledgements",
and any sections Entitled "Dedications".  You must delete all sections
Entitled "Endorsements".


6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other
documents released under this License, and replace the individual
copies of this License in the various documents with a single copy
that is included in the collection, provided that you follow the rules
of this License for verbatim copying of each of the documents in all
other respects.

You may extract a single document from such a collection, and
```

```
distribute it individually under this License, provided you insert a
copy of this License into the extracted document, and follow this
License in all other respects regarding verbatim copying of that
document.


7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate
and independent documents or works, in or on a volume of a storage or
distribution medium, is called an "aggregate" if the copyright
resulting from the compilation is not used to limit the legal rights
of the compilation's users beyond what the individual works permit.
When the Document is included in an aggregate, this License does not
apply to the other works in the aggregate which are not themselves
derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these
copies of the Document, then if the Document is less than one half of
the entire aggregate, the Document's Cover Texts may be placed on
covers that bracket the Document within the aggregate, or the
electronic equivalent of covers if the Document is in electronic form.
Otherwise they must appear on printed covers that bracket the whole
aggregate.


8. TRANSLATION

Translation is considered a kind of modification, so you may
distribute translations of the Document under the terms of section 4.
Replacing Invariant Sections with translations requires special
permission from their copyright holders, but you may include
translations of some or all Invariant Sections in addition to the
original versions of these Invariant Sections.  You may include a
translation of this License, and all the license notices in the
Document, and any Warranty Disclaimers, provided that you also include
the original English version of this License and the original versions
of those notices and disclaimers.  In case of a disagreement between
the translation and the original version of this License or a notice
or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements",
"Dedications", or "History", the requirement (section 4) to Preserve
its Title (section 1) will typically require changing the actual
title.


9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense, or distribute it is void, and
will automatically terminate your rights under this License.
```

```

However, if you cease all violation of this License, then your license
from a particular copyright holder is reinstated (a) provisionally,
unless and until the copyright holder explicitly and finally
terminates your license, and (b) permanently, if the copyright holder
fails to notify you of the violation by some reasonable means prior to
60 days after the cessation.

Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, receipt of a copy of some or all of the same material does
not give you any rights to use it.


10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the
GNU Free Documentation License from time to time.  Such new versions
will be similar in spirit to the present version, but may differ in
detail to address new problems or concerns.  See
http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number.
If the Document specifies that a particular numbered version of this
License "or any later version" applies to it, you have the option of
following the terms and conditions either of that specified version or
of any later version that has been published (not as a draft) by the
Free Software Foundation.  If the Document does not specify a version
number of this License, you may choose any version ever published (not
as a draft) by the Free Software Foundation.  If the Document
specifies that a proxy can decide which future versions of this
License can be used, that proxy's public statement of acceptance of a
version permanently authorizes you to choose that version for the
Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any
World Wide Web server that publishes copyrightable works and also
provides prominent facilities for anybody to edit those works.  A
public wiki that anybody can edit is an example of such a server.  A
"Massive Multiauthor Collaboration" (or "MMC") contained in the site
means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0
```

```
license published by Creative Commons Corporation, a not-for-profit
corporation with a principal place of business in San Francisco,
California, as well as future copyleft versions of that license
published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in
part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this
License, and if all works that were first published under this License
somewhere other than this MMC, and subsequently incorporated in whole or
in part into the MMC, (1) had no cover texts or invariant sections, and
(2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site
under CC-BY-SA on the same site at any time before August 1, 2009,
provided the MMC is eligible for relicensing.


ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of
the License in the document and put the following copyright and
license notices just after the title page:

    Copyright (c)  YEAR  YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```

6.2 LGPL: GNU Lesser Gnu Public License

```
                   GNU LESSER GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


  This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

  0. Additional Definitions.

  As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

  "The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

  An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

  A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
```

```
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

   a) Give prominent notice with each copy of the Combined Work that
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
   document.

   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
   copies of the GNU GPL and this license document.

   d) Do one of the following:

       0) Convey the Minimal Corresponding Source under the terms of this
       License, and the Corresponding Application Code in a form
       suitable for, and under terms that permit, the user to
       recombine or relink the Application with a modified version of
```

```
       the Linked Version to produce a modified Combined Work, in the
       manner specified by section 6 of the GNU GPL for conveying
       Corresponding Source.

       1) Use a suitable shared library mechanism for linking with the
       Library.  A suitable mechanism is one that (a) uses at run time
       a copy of the Library already present on the user's computer
       system, and (b) will operate properly with a modified version
       of the Library that is interface-compatible with the Linked
       Version.

   e) Provide Installation Information, but only if you would otherwise
   be required to provide such information under section 6 of the
   GNU GPL, and only to the extent that such information is
   necessary to install and execute a modified version of the
   Combined Work produced by recombining or relinking the
   Application with a modified version of the Linked Version. (If
   you use option 4d0, the Installation Information must accompany
   the Minimal Corresponding Source and Corresponding Application
   Code. If you use option 4d1, you must provide the Installation
   Information in the manner specified by section 6 of the GNU GPL
   for conveying Corresponding Source.)

  5. Combined Libraries.

  You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based
   on the Library, uncombined with any other library facilities,
   conveyed under the terms of this License.

   b) Give prominent notice with the combined library that part of it
   is a work based on the Library, and explaining where to find the
   accompanying uncombined form of the same work.

  6. Revised Versions of the GNU Lesser General Public License.

  The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

  Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
```

```
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

  If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```

This manual built September 18, 2012

INDEX