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

Release 2011-09

http://nexusformat.org

September 25, 2011

CONTENTS

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

NeXus home: http://nexusformat.org

This manual is also available in a PDF version.

Part I

Contents

CHAPTER

ONE

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.

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:

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

```
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 separated by comma. The `@` symbol only indicates this is an attribute. The `@` symbol is not part of the attribute name.
- Links are shown with a text arrow `-->` indicating the source of the link (using HDF5 notation listing the sequence of names).

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

[Lines 2, 10, and 12] The `/entry` group contains three subgroups: `instrument`, `sample`, and `data`. These groups are of type `NXinstrument`, `NXsample`, and `NXdata`, respectively.

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

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

[Line 8] (reading `bins:[0, 1, 2, ... 1023]`) shows that `bins` is a 1-D array of length presumably 1024. A small, representative selection of values are shown.

[Line 9] an attribute that shows a descriptive name of `/entry/instrument/detector/bins`. This attribute might be used by a NeXus client while plotting the data.

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

[Lines 13-14] The `/entry/data` group has two datasets that are actually linked as shown. (As you will see later, the `NXdata` group is required and enables NeXus clients to easily determine what to offer for display on a default plot.)

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

CHAPTER

TWO

NEXUS: USER MANUAL

Contents:

2.1 NeXus Introduction

In recent years, a community of scientists and computer programmers working in neutron and synchrotron facilities around the world came to the conclusion that a common data format would fulfill a valuable function in the scattering community. As instrumentation becomes more complex and data visualization become more challenging, individual scientists, or even institutions, have found it difficult to keep up with new developments. A common data format makes it easier, both to exchange experimental results and to exchange ideas about how to analyze them. It promotes greater cooperation in software development and stimulates the design of more sophisticated visualization tools. For additional background information see *history (not converted yet)* (page 65).

quote

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

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

2.1.1 What is NeXus?

The NeXus data format has four components:

1. *A Set of Design Principles* (page 10) to help people understand what is in the data files.
2. *A Set of Data Storage Objects* (page 14) (base classes and application definitions) to allow the development of more portable analysis software.
3. *A Set of Subroutines* (page 15) (utilities) to make it easy to read and write NeXus data files.

4. *NeXus Scientific Community* (page 15) provides the scientific data, advice, and continued involvement with the NeXus standard. NeXus provides a forum for the scientific community to exchange ideas in data storage.

The NeXus 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 19). (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 *NeXus Design* (page 22).

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

Additionally, a brief list describing the set of NeXus Utilities available to browse, validate, translate, and visualise NeXus data files is provided in *utilities (not converted yet)* (page 65).

A Set of Design Principles

NeXus data files contain four types of entity: data groups, data fields, attributes, and links. See *Data Groups* (page 23) for more details.

1. ***Data Groups*** **(page 23)** *Data groups* are like folders that can contain a number of fields and/or other groups.
2. ***Data Fields*** **(page 23)** *Data fields* can be scalar values or multidimensional arrays of a variety of sizes (1-byte, 2-byte, 4-byte, 8-byte) and types (characters, integers, floats). In HDF, fields are represented as HDF *Scientific Data Sets* (also known as SDS).
3. ***Data Attributes*** **(page 24)** Extra information required to describe a particular group or field, such as the data units, can be stored as a data attribute.
4. ***Data Links*** **(page 25)** Links are used to reference the plottable data from `NXdata` when the data is provided in other groups such as `NXmonitor` or `NXdetector`.

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

Example of a NeXus File

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

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

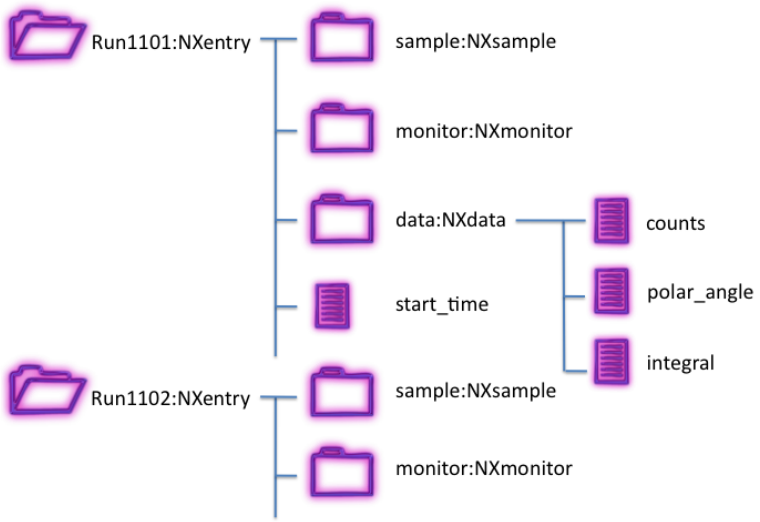


Figure 2.1: Example of a NeXus data file

Important Classes

Here are some of the important classes found in nearly all NeXus files. A complete list can be found in the NeXus Design section (*NeXus Design* (page 22)).

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

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

`NXdata` (**Required:**) Each `NXentry` group contains one or more groups with class `NXdata`. These groups contain the experimental results in a self-contained way, i.e., it should be possible to generate a sensible plot of the data from the information contained in each `NXdata` group. That means it should contain the axis labels and titles as well as the data.

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

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

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

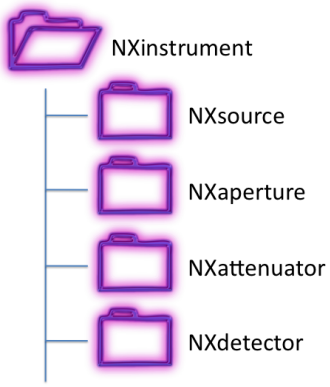


Figure 2.2: NXinstrument excerpt

Simple Data File Example

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

Figure 2.3: Simple Data File Example

This illustrates the fact that the structure of NeXus files is extremely flexible. It can accommodate very complex instrumental information, if required, but it can also be used to store very simple data sets. In the next example, a NeXus data file is shown as XML:

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

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

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

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

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

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

from nexpy.api import nexus

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

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

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

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

A Set of Data Storage Objects

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

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

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

`verysimple.nxdl.xml`: A very simple NeXus Definition Language (NXDL) file

```
<?xml version="1.0" ?>
<definition
  xmlns="http://definition.nexusformat.org/nxdl/3.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://definition.nexusformat.org/nxdl/3.1 ../nxdl.xsd"
  category="base"
  name="verysimple"
  version="1.0"
  svnid="$Id: objects.rst 885 2011-09-13 05:23:07Z Pete Jemian $"
  type="group" extends="NXobject">

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

This chapter has several examples of writing and reading NeXus data files. If you want to define the format of a particular type of NeXus file for your own use, e.g. as the standard output from a program, you are encouraged to *publish* the format using this XML format. An example of how to do this is shown in the section titled Creating a NXDL Specification (*nxdl_tutorial-creatingnxdlspec (not converted yet)* (page 65)).

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

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

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

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

NeXus Scientific Community

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. [1]

The NeXus International Advisory Committee (see *NIAC description* (page 65)) 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.

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

2.1.2 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 Koennecke (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 at two workshops, SoftNeSS'95 (NIST Sept. 1995) and SoftNeSS'96 (Argonne Oct. 1996), attended by representatives of a range of neutron and x-ray facilities.

Basic motivations for the NeXus standard

The NeXus API was released in late 1997. Basic motivations for this standard were:

Simple plotting

An important motivation for the design of NeXus was to simplify the creation of a default plot view. While the best representation of a set of observations will vary, depending on various conditions, a good suggestion is often known *a priori*. This suggestion is described in the `NXdata` element so that any program that is used to browse NeXus data files can provide a *best representation* without request for user input.

A 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 has been extended by the addition of the eXtensible Markup Language (XML) [2] 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.

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 the next figure, circles represent different data file formats while arrows represent conversion routines. Note that the red circle only maps to one other format.

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 shown in the next figure.

A 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`, ...).

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

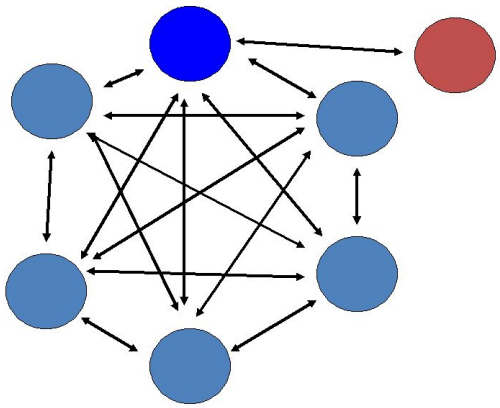


Figure 2.4: *N* separate file formats

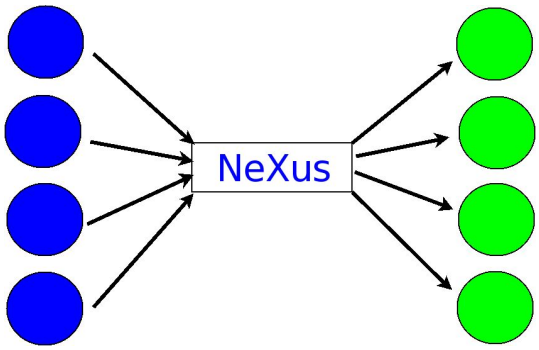


Figure 2.5: *N* separate file formats joined by a common NeXus converter

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.

2.1.3 NAPI: The NeXus Application Programming Interface

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

Further documentation of the NeXus Application Programming Interface (NAPI) for bindings to specific programming language can be obtained from the NeXus development site. [3]

For a more detailed description of the internal workings of NAPI that is maintained (mostly) concurrent with code revisions, see the NAPI chapter in Volume II of this documentation and also NeXusIntern.pdf in the NeXus code repository. [4] Likely this is only interesting for experienced programmers who wish to hack the NAPI.

How do I write a NeXus file?

The NeXus Application Program Interface (API) provides a set of subroutines that make it easy to read and write NeXus files. These subroutines are available in C, Fortran 77, Fortran 90, Java, Python, C++, and IDL. Access from other languages, such as Python, is anticipated in the near future. It is also possible to read NeXus HDF files in a number of data analysis tools, such as LAMP, ISAW, IgorPro, and Open GENIE. NeXus XML files can be read by any program or library that supports XML.

The API uses a very simple *state* model to navigate through a NeXus file. When you open a file, the API provides a file *handle*, which then stores the current location, i.e. which group and/or field is currently open. Read and write operations then act on the currently open entity. Following the *Simple Data File Example* (page 12), we walk through some parts of a typical NeXus program written in C.

Writing a simple NeXus file

```
#include "napi.h"

 int main()
 {
        NXhandle fileID;
        NXopen ('NXfile.nxs', NXACC_CREATE, &fileID);
          NXmakegroup (fileID, "Scan", "NXentry");
          NXopengroup (fileID, "Scan", "NXentry");
                NXmakegroup (fileID, "data", "NXdata");
                NXopengroup (fileID, "data", "NXdata");
                /* somehow, we already have arrays tth and counts, each length n*/
                  NXmakedata (fileID, "two_theta", NX_FLOAT32, 1, &n);
                  NXopendata (fileID, "two_theta");
                        NXputdata (fileID, tth);
```

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

```
15                          NXputattr (fileID, "units", "degrees", 7, NX_CHAR);
16                    NXclosedata (fileID);  /* two_theta */, NX_INT32, 1, &n);
17                    NXopendata (fileID, "counts");
18                          NXputdata (fileID, counts);
19                    NXclosedata (fileID);  /* counts */
20                  NXclosegroup (fileID);  /* data */
21            NXclosegroup (fileID);  /* Scan */
22          NXclose (&fileID);
23          return;
24  }
```

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

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

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

[line 12] To create a field, call `NXmakedata()`, specifying the data name, type (`NX_FLOAT32`), rank (in this case, `1`), and length of the array (`n`). Then, it can be opened for writing.

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

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

[line 16] Then we close the field before opening another. In fact, the API will do this automatically if you attempt to open another field, but it is better style to close it yourself.

[line 17] The remaining fields in this group are added in a similar fashion. Note that the indentation whenever a new field or group are opened is just intended to make the structure of the NeXus file more transparent.

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

How do I read a NeXus file?

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

Here is part of a program to read the two-theta array from the file created by *Writing a simple NeXus file* (page 19) above.

Reading a simple NeXus file

```
1   NXopen ('NXfile.nxs', NXACC_READ, &fileID);
2     NXopengroup (fileID, "Scan", "NXentry");
3           NXopengroup (fileID, "data", "NXdata");
4             NXopendata (fileID, "two_theta");
5                   NXgetinfo (fileID, &rank, dims, &datatype);
```

```
                NXmalloc ((void **) &tth, rank, dims, datatype);
                NXgetdata (fileID, tth);
            NXclosedata (fileID);
          NXclosegroup (fileID);
    NXclosegroup (fileID);
 NXclose (fileID);
```

How do I browse a NeXus file?

NeXus files can also be viewed by a command-line browser, `NXbrowse`, which is included with the NeXus API (*NAPI: The NeXus Application Programming Interface* (page 19)). The following is an example session of using `nxbrowse` to view a data file from the LRMECS spectrometer at IPNS. The following commands are used in *Using NXbrowse* (page 21) in this session (see the `nxbrowse` web page):

Using `NXbrowse`

```
%> nxbrowse lrcs3701.nxs

NXBrowse 3.0.0. Copyright (C) 2000 R. Osborn, M. Koennecke, P. Klosowski
        NeXus_version = 1.3.3
        file_name = lrcs3701.nxs
        file_time = 2001-02-11 00:02:35-0600
        user = EAG/RO
NX> dir
  NX Group : Histogram1 (NXentry)
  NX Group : Histogram2 (NXentry)
NX> open Histogram1
NX/Histogram1> dir
  NX Data  : title[44] (NX_CHAR)
  NX Data  : analysis[7] (NX_CHAR)
  NX Data  : start_time[24] (NX_CHAR)
  NX Data  : end_time[24] (NX_CHAR)
  NX Data  : run_number (NX_INT32)
  NX Group : sample (NXsample)
  NX Group : LRMECS (NXinstrument)
  NX Group : monitor1 (NXmonitor)
  NX Group : monitor2 (NXmonitor)
  NX Group : data (NXdata)
NX/Histogram1> read title
  title[44] (NX_CHAR) = MgB2 PDOS 43.37g 8K 120meV E0@240Hz T0@120Hz
NX/Histogram1> open data
NX/Histogram1/data> dir
  NX Data  : title[44] (NX_CHAR)
  NX Data  : data[148,750] (NX_INT32)
  NX Data  : time_of_flight[751] (NX_FLOAT32)
  NX Data  : polar_angle[148] (NX_FLOAT32)
NX/Histogram1/data> read time_of_flight
  time_of_flight[751] (NX_FLOAT32) = [ 1900.000000 1902.000000 1904.000000 ...]
        units = microseconds
        long_name = Time-of-Flight [microseconds]
```

```
35 NX/Histogram1/data> read data
36   data[148,750] (NX_INT32) = [ 1 1 0 ...]
37         units = counts
38         signal = 1
39         long_name = Neutron Counts
40         axes = polar_angle:time_of_flight
41 NX/Histogram1/data> close
42 NX/Histogram1> close
43 NX> quit
```

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

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

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

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

[line 41] Close the current group.

[line 43] Quits `NXbrowse`.

The source code of `NXbrowse` [5] provides an example of how to write a NeXus reader. The test programs included in the NeXus API (*NAPI: The NeXus Application Programming Interface* (page 19)) may also be useful to study.

2.2 NeXus Design

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

> **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.

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.

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

2.2.1 NeXus Objects and Terms

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

Data Groups **(page 23)** Group data fields and other groups together. Groups represent levels in the NeXus hierarchy

Data Fields **(page 23)** Multidimensional arrays and scalars representing the actual data to be stored

Data Attributes **(page 24)** Additional metadata which can be assigned to groups or data fields

Data Links **(page 25)** Elements which point to data stored in another place in the file hierarchy

NeXus Classes **(page 25)** Dictionaries of names possible in the various types of NeXus groups

NeXus Application Definitions **(page 27)** Describe the content of a NeXus file for a particular usage case

NeXus Coordinate Systems **(page 28)** Coordinate systems are used to describe the positions and orientations of objects.

Rules for Structuring Information in NeXus Files **(page 31)**

This section describes *where to place data* within a NeXus data file.

Rules for Storing Data Items in NeXus Files **(page 32)** This section describes *how to store data* in a NeXus data file.

Physical File format **(page 38)** This section describes how NeXus structures are mapped to features of the underlying physical file format.

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.

Data Attributes

Attributes are extra (meta-)information that are associated with particular fields. They are used to annotate the data, for example with physical units or calibration offsets, and may be scalar numbers or character strings. NeXus also uses attributes to identify plottable data and their axes, etc. A description of possible attributes can be found in the table titled *Example NeXus Data Attributes* (page 24). Finally, NeXus files themselves have global attributes which are listed in the *NeXus File Global Attributes* (page 25) table that identify the NeXus version, file creation time, etc. Attributes are identified by their names, which must be unique in each field.

Example NeXus Data Attributes

For the full specification of attributes, see *volume2.NXDL.section (not converted yet)* (page 65).

Name	Type	Description
units	NX_CHAR	Data units, given as character strings, must conform to the NeXus units standard. See the "NeXus units" 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	Defines the names of the dimension scales for this data set as a colon-delimited list. 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 *nxdata-structure (not converted yet)* (page 65) 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
transformation_type	NX_CHAR	translation or totation
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), or vertex (3-D data).

NeXus File Global Attributes

Name	Type	Description
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 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 access to the same data in different groups within 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 *Linking in a NeXus file* (page 25) for a more descriptive representation of the concept of linking.

Linking in a NeXus file

NeXus 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*.

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

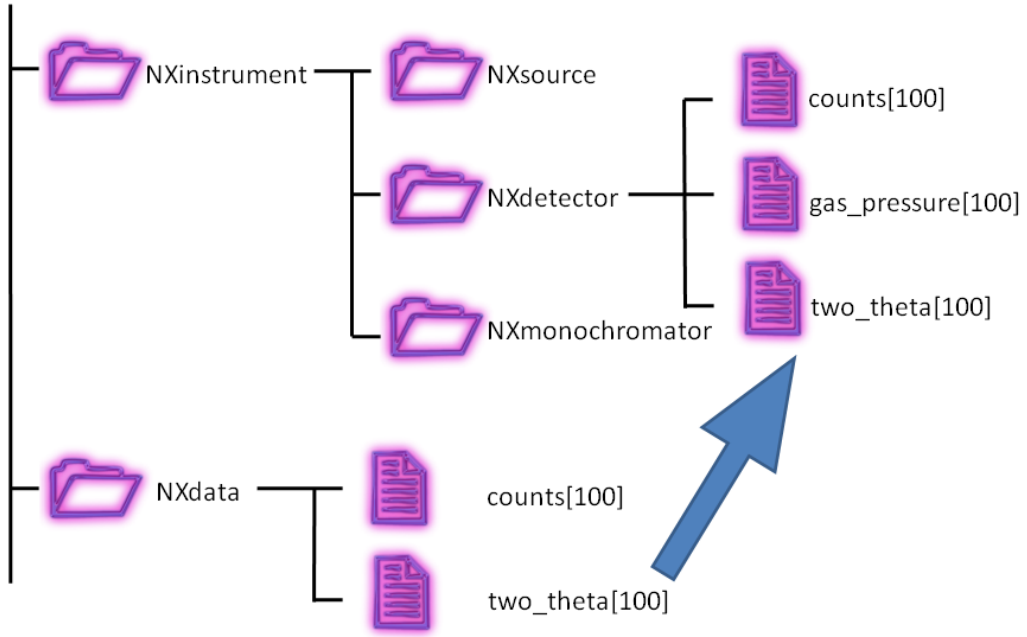


Figure 2.6: Linking in a NeXus file. See example code below: *example.data-linking (not converted yet)* (page 65)

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 *NXnote* 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 (also known as *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.
    1. Search for a dataset with attribute `signal=1`. This is your main dataset. (There should be only one dataset that matches.)
    2. Try to read the axes attribute of the main dataset, if it exists.
    1. 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"`).
    2. Parse axes and open the datasets to describe your dimension scales
    3. If axes does not exist:
    1. 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.
    2. 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 to validate a NeXus file against a given application definition.

NeXus application definition is a contract

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

NeXus Coordinate Systems

Coordinate systems in NeXus underwent quite some development. Initially, just positions of relevant motors were stored without further standardization. This soon proved to be too little and the *NeXus polar coordinate* system was developed. This system still is very close to angles 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* [6] system. We soon learned that all the things we do can be expressed through the McStas coordinate system. So the McStas coordinate system 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 [7] 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

NeXus uses the *McStas coordinate system* [1] 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
- 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 *The McStas Coordinate System* (page 29).

Note: The NeXus definition of $+z$ is opposite to that in the International Tables for Crystallography, volume G, [8] and consequently, $+x$ is also reversed.

[6] McStas, http://www.mcstas.org, also http://mcstas.risoe.dk

[7] CIF (Crystallographic Information Framework), http://www.iucr.org/resources/cif

[8] **International Tables for Crystallography** *Volume G: Definition and exchange of crystallographic data*. Sydney Hall and Brian McMahon, Editors. Published for the IUCr by Springer, 2005 ISBN 1-4020-3138-6, 594 + xii pages

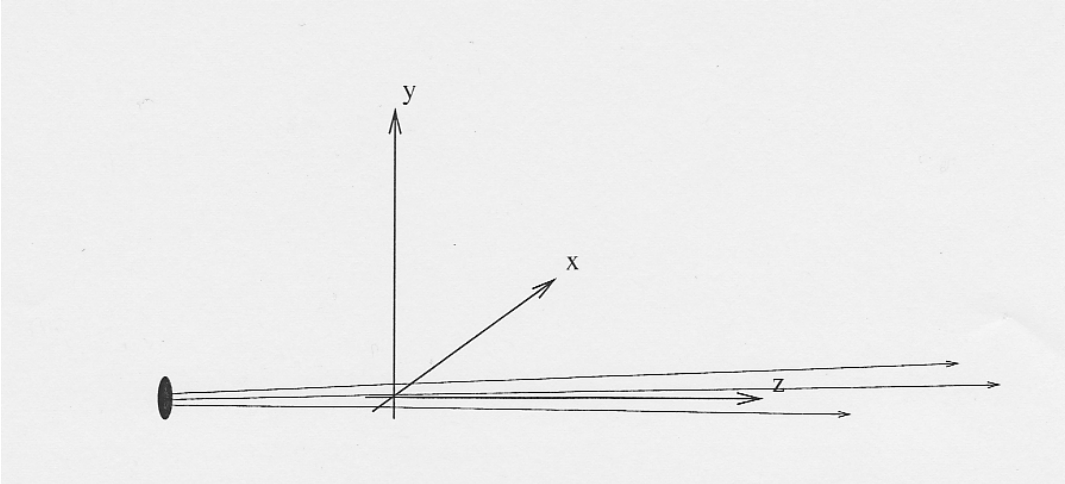


Figure 2.7: The McStas Coordinate System

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

Simple (Spherical Polar) Coordinate System

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

In order to understand this spherical polar coordinate system it is helpful to look initially at the common condition that azimuthal_angle is zero. This corresponds to working directly in the horizontal scattering plane of the instrument. In this case polar_angle maps directly to the setting commonly known as two theta. Now, there are instruments where components live outside of the scattering plane. Most notably detectors. In order to describe such components we first apply the tilt out of the horizontal scattering plane as the azimuthal_angle. Then, in this tilted plane, we rotate to the component. The beauty of this is that polar_angle is always two theta. Which, in the case of a component out of the horizontal scattering plane, is not identical to the value read from the motor responsible for rotating the component. This situation is shown in figure *NeXus Simple (Spherical Polar) Coordinate System* (page 30).

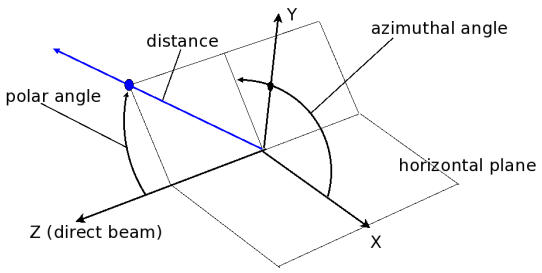


Figure 2.8: NeXus Simple (Spherical Polar) Coordinate System

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 corrdinate 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.

Actions of standard NeXus fields

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

The type and direction of the NeXus standard operations is documented in table *Actions of standard NeXus fields* (page 30). NeXus can now also allow non standard operations to be stored in data files. In such cases additional data attributes are required which describe the operation. These are *transformation_type* which can be either translation or rotation. The other is *vector* which is 3 float values describing the direction of translation or rotation. The value is of course always the value of the data field in the data file.

How NeXus describes the order of operations to apply has not yet been decided upon. The authors favorite 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 transfrom 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 by:

```
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.

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 <link xlink:href="#table.RawData">Raw Data</link> are provided.

Content of a Raw Data `NXentry` Group An example raw data hierarchy is shown in the next figure (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.

```
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.

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 adheres to these rules. The names of NeXus *group* and *field* objects must contain only a restricted set of characters. This set may be described by this regular expression syntax.

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

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 are to be used. [9] 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.

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

One good reason to deviate from this rule is the situation where data must be streamed to disk as fast as possible and a conversion to NeXus storage order is not possible. In such cases, exceptions can be made. It is possible to store data in other storage orders in NeXus as well as to specify that the data needs to be converted first before being useful.

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.

[9] The NeXus base classes provide a comprehensive dictionary of terms than can be used for each class.

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 2-D 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 3-D 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
```

Data Value Transformations It is possible to store raw values in NeXus data files. Such data has to be stored in special *NXformula* [10] groups together with the data and information required to transform it into physical values.

NeXus Data Types Matching regular expressions for 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 35).

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 [11] format, such as:

```
1996-07-31T21:15:22+0600
```

[10] NeXus has not yet defined the *NXformula* group (or base class) for use in NeXus data files. The exact content of the *NXformula* group is still under discussion.

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

Note:

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*. While human-readable, compatibility with the ISO 8601 standard is not assured with this substitution.

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.

NeXus 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. [12] The UDunits specification also includes instructions for derived units. At present, the contents of NeXus *units* attributes are not validated in data files. 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 ...
```

[12] Unidata UDunits, http://www.unidata.ucar.edu/software/udunits/udunits-2-units.html

```
  some_other_angle = 0.0 0.0 2.0 ...
  data = 5 7 14 ...
    @axes = polar_angle:time_of_flight
    @signal = 1
```

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:

```
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.

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.

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.

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.

Physical File format This section describes how NeXus structures are mapped to features of the underlying physical file format. This can also be considered 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) [13], as a multi-platform data storage format with capacity for conveying large data payloads and a substantial user community. HDF (now HDF5) was 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.

[13] HDF: http://www.hdfgroup.org, initially from 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).

Mapping NeXus into HDF NeXus data structures map directly to HDF structures.

NeXus *groups* HDF4 *vgroups* or HDF5 *groups*

NeXus data sets (or *fields*) HDF4 *SDS (scientific data sets)* or HDF5 *datasets*

Attributes 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 a group attribute with the name `NX_class`.

NeXus links directly map to the HDF linking mechanisms.

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.

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 of a 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` [14] in the element. An example of two NeXus data elements in XML:

```
<mode NAPItype="NX_CHAR[7]">
        monitor
</mode>
<counts NAPItype="NX_INT32[4]">
        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 [15] 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.

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

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 The *target* 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`.

2.3 Frequently Asked Questions

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

2.3.1 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 :ref:`Facilities Wiki page`. Please have a look at this list. Keep in mind that it is not complete.

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

NeXus files are not binary *per se*. 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.

2.3.3 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.

2.3.4 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.

2.3.5 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 main stream 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.

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

Read the NXDL Tutorial in *nxdl_tutorial-creatingnxdlspec (not converted yet)* (page 65). The procedures for acceptance are defined in the NIAC constitution. Refer to the most recent version of the NIAC constitution on the NIAC wiki [16].

2.3.7 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 16)) for the NeXus standard. The choice of the name `NXdata` is historic and does not really reflect its function.

2.3.8 How do I identify the plottable data?

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

[16] NIAC wiki: http://www.nexusformat.org/NIAC

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

2.3.9 How can I specify reasonable axes for my data?

See the section: *NXdata Facilitates Automatic Plotting* (page 26).

2.3.10 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.

2.3.11 Specifications are boring. Where can I find some good example data files?

There are a few checked into the *definitions repository*. At the moment the selection is quite limited and not very representative.

2.3.12 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.

2.3.13 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.

2.3.14 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.

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

NeXus uses HDF5 hard links. 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. 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.

CHAPTER

THREE

NEXUS: REFERENCE DOCUMENTATION

Contents:

CHAPTER

FOUR

DOCUMENTATION AUTHORS

These people have made substantial contributions to the NeXus manual:

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

CHAPTER

FIVE

TODO ITEMS

- fix column width problems in PDF
- fix math source formatting between html and pdf
- tables, examples, and figures: treat them consistently with titles, captions, and cross-references
- stop the section numbering for very deep subsections (2.1.4.1.2.1.3.1.4.5.1.4.1... is just ridiculous)
- try to get the contents to look like:
  - Preface
  - Volume 1
  - Volume 2
- OR, should we produce two or more separate books?

Part II

Cheatsheet

This is a cheat sheet and will be removed later.

Section headings automatically get labels assigned. For example, see this: Demo list-table (page 59)

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

CHAPTER

SIX

TYPESETTING MATH AND EQUATIONS

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

TODO:

adjust *conf.py*?

Sphinx has some inconsistency with this expression:

```
.. ! this is a candidate for conditional compilation
   make html      needs two backslashes while
   make latexpdf  needs one backslash

.. math::

        \tilde I(Q) = {2 \over l_o} \ \int_0^\infty I(\sqrt{(q^2+l^2)}) \ dl
```

Tip: Perhaps some modification of *conf.py* would help?

The Sphinx HTML renderer handles simple math this way but not all LaTeX markup. The HTML renderer needs two backslashes while the LaTeX renderer only needs one.

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

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

CHAPTER

SEVEN

OTHER LINKS

Here are some links to more help about reStructuredText formatting.

reST home page http://docutils.sourceforge.net/rst.html

Docutils http://docutils.sourceforge.net/

Very useful! http://docutils.sourceforge.net/docs/ref/rst/directives.html

Independent Overview http://www.siafoo.net/help/reST

Wikipedia http://en.wikipedia.org/wiki/ReStructuredText

reST Quick Reference http://docutils.sourceforge.net/docs/user/rst/quickref.html

Comparison: text v. reST v. DocBook http://www.ibm.com/developerworks/library/x-matters24/

Curious http://rst2a.com/

CHAPTER

EIGHT

DEMO LIST-TABLE

Does this work?

It was found on this page http://docutils.sourceforge.net/docs/ref/rst/directives.html

Table 8.1: Frozen Delights!

Treat	Quantity	Description
Albatross	2.99	On a stick!
Crunchy Frog	1.49	If we took the bones out, it wouldn't be crunchy, now would it?
Gannet Ripple	1.99	On a stick!

CHAPTER

NINE

NUMBERED LISTS

What about automatically numbering a list?

1. How will the numbering look?
2. Will it look great?
3. Even more great? What about more than one line of text in the source code?
8. Made a jump in the numbering. But that started a new list and produced a compile error. What about more than one line of text in the source code? Cannot use multiple paragraphs in a list, it seems. Maybe there is a way.
9. And another ...
6. Perhaps we can switch to lettering? Only if we start a new list. But we needed a blank line at the switch.
7. Another lettered item.

CHAPTER

TEN

ABOUT LINKING

What about a link to *Indirect Hyperlinks* (page 86) on another page?

The reSt documentation says that links can be written as:

```
`NeXus: User Manual`_
```

This works for sphinx, as long as the link target is in the same `.rst` document. **But**, when the link is in a different document, sphinx requires the citation to use:

```
:ref:`NeXus User Manual`
```

and the target must be a section with an explicit hyperlink definition, such as on the top page of these docs:

```
.. _NeXus User Manual:

######################
NeXus: User Manual
######################
```

This is the correct link: *NeXus: User Manual* (page 9).

CHAPTER

ELEVEN

MISSING LINKS

These sections show up as missing links.

Can you find the history (page 65) link below? What about the history (page 65) link below? This works: *history (not converted yet)* (page 65) (or *history (not converted yet)* (page 65)).

11.1 history (not converted yet)

11.2 utilities (not converted yet)

11.3 nxdl_tutorial-creatingnxdlspec (not converted yet)

11.4 nxdata-structure (not converted yet)

11.5 volume2.NXDL.section (not converted yet)

11.6 NIAC description

11.7 example.data-linking (not converted yet)

11.7.1 Section to cross-reference

This is the text of the section.

It refers to the section itself, see *Section to cross-reference* (page 65). What about a section on another page, such as *Footnote References* (page 87)?

Part III

reStructuredText Markup Specification

Author: David Goodger Contact: dgoodger@bigfoot.com Version: 0.2 Date: 2001-05-29

Help for reST authors

This document, while old, presents information useful to authors of reST documents, such as this manual.
This will be removed in the released version of the NeXus documentation.

reStructuredText is plain text that uses simple and intuitive constructs to indicate the structure of a document. These constructs are equally easy to read in raw and processed forms. This document is itself an example of reStructuredText (raw, if you are reading the text file, or processed, if you are reading an HTML document, for example). reStructuredText is a candidate markup syntax for the Python Docstring Processing System.

Simple, implicit markup is used to indicate special constructs, such as section headings, bullet lists, and emphasis. The markup used is as minimal and unobtrusive as possible. Less often-used constructs and extensions to the basic reStructuredText syntax may have more elaborate or explicit markup.

The first section gives a quick overview of the syntax of the reStructuredText markup by example. More details are given in the Syntax Details (page 73) section.

Literal blocks (page 79) are used for examples throughout this document.

CHAPTER

TWELVE

QUICK SYNTAX OVERVIEW

A reStructuredText document is made up of body elements, and may be structured into sections. Section Structure (page 74) is indicated through title style (underlines & optional overlines). Sections contain body elements and/or subsections.

Here are examples of body elements:

- Paragraphs (page 75) (and inline markup (page 84)):

```
Paragraphs contain text and may contain inline markup: *emphasis*,
**strong emphasis**, `interpreted text`, ``inline literals``,
standalone hyperlinks (http://www.python.org), indirect hyperlinks
(Python_), internal cross-references (example_), footnote references
([1]_).

Paragraphs are separated by blank lines and are flush left.
```

- Three types of lists:

  1. Bullet lists (page 75):

```
- This is a bullet list.

- Bullets can be '-', '*', or '+'.
```

  2. Enumerated lists (page 76):

```
1. This is an enumerated list.

2. Enumerators may be arabic numbers, letters, or roman numerals.
```

  3. Definition lists (page 77):

```
what
    Definition lists associate a term with a definition.

how
    The term is a one-line phrase, and the definition is one or
    more paragraphs or body elements, indented relative to the
    term.
```

- Literal blocks (page 79):

```
Literal blocks are indented, and indicated with a double-colon ('::')
at the end of the preceeding paragraph::

    if literal_block:
        text = 'is left as-is'
        spaces_and_linebreaks = 'are preserved'
        markup_processing = None
```

- Block quotes (page 80):

```
Block quotes consist of indented body elements:

    This theory, that is mine, is mine.

    Anne Elk (Miss)
```

- Tables (page 81):

```
+------------------------+------------+----------+
| Header row, column 1   | Header 2   | Header 3 |
+========================+============+==========+
| body row 1, column 1   | column 2   | column 3 |
+------------------------+------------+----------+
| body row 2             | Cells may span        |
+------------------------+------------+----------+
```

- Comments (page 81):

```
.. Comments begin with two dots and a space. Anything may follow,
   except for the syntax of directives, footnotes, and hyperlink
   targets, described below.
```

- Directives (page 82):

```
.. graphic:: mylogo.png
```

- Footnotes (page 83):

```
.. _[1] A footnote contains indented body elements.

   It is a form of hyperlink target.
```

- Hyperlink targets (page 83):

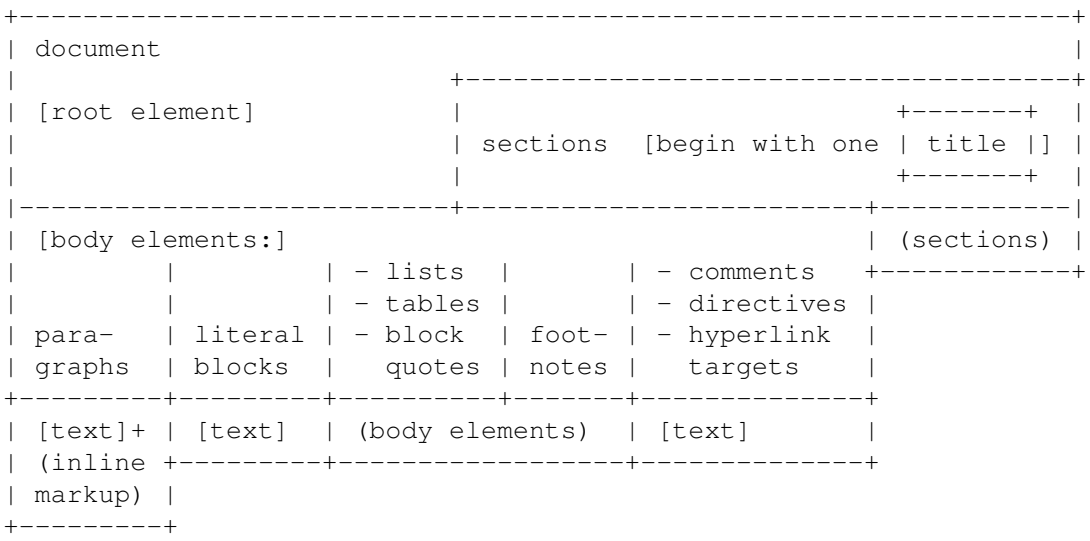
```
.. _Python: http://www.python.org

.. _example:

The '_example' target above points to this paragraph.
```

CHAPTER

THIRTEEN

SYNTAX DETAILS

Below is a diagram of the hierarchy of element types in reStructuredText. Elements may contain other elements below them. Element types in parentheses indicate recursive or one-to-many relationships: sections may contain (sub)sections, tables contain further body elements, etc.



For definitive element hierarchy details, see the "Generic Plaintext Document Interface DTD" XML document type definition, gpdi.dtd. Descriptions below list 'DTD elements' (XML 'generic identifiers') corresponding to syntax constructs.

13.1 Whitespace

Blank lines are used to separate paragraphs and other elements. Blank lines may be omitted when the markup makes element separation unambiguous.

Indentation is used to indicate, and is only significant in indicating:

- multiple body elements within a list item (including nested lists),
- the definition part of a definition list item,
- block quotes, and

- the extent of literal blocks.

Although spaces are recommended for indentation, tabs may also be used. Tabs will be converted to spaces. Tab stops are at every 8th column.

13.2 Escaping Mechanism

The character set available in plain text documents, 7-bit ASCII, is limited. No matter what characters are used for markup, they will already have multiple meanings in written text. Therefore markup characters *will* sometimes appear in text **without being intended as markup**.

Any serious markup system requires an escaping mechanism to override the default meaning of the characters used for the markup. In reStructuredText we use the backslash, commonly used as an escaping character in other domains.

A backslash followed by any character escapes the character. The escaped character represents the character itself, and is prevented from playing a role in any markup interpretation. The backslash is removed from the output. A literal backslash is represented by two backslashes in a row.

There are two contexts in which backslashes have no special meaning: literal blocks and inline literals. In these contexts, a single backslash represents a literal backslash.

13.3 Section Structure

DTD elements: section, title.

Sections are identified through their titles, which are marked up with 'underlines' below the title text (and, in some cases, 'overlines' above the title). An underline/overline is a line of non-alphanumeric characters that begins in column 1 and extends at least as far as the right edge of the title text. When there an overline is used, the length and character used must match the underline. There may be any number of levels of section titles.

Rather than imposing a fixed number and order of section title styles, the order enforced will be the order as encountered. The first style encountered will be an outermost title (like HTML H1), the second style will be a subtitle, the third will be a subsubtitle, and so on.

Below are examples of section title styles:

```
===============
 Section Title
===============

---------------
 Section Title
---------------

Section Title
=============

Section Title
```

```
-------------

Section Title
.............

Section Title
~~~~~~~~~~~~~

Section Title
*************

Section Title
+++++++++++++

Section Title
^^^^^^^^^^^^^
```

When a title has both an underline and an overline, the title text may be inset, as in the first two examples above. This is merely aesthetic and not significant. Underline-only title text may not be inset.

A blank line after a title is optional. All text blocks up to the next title of the same or higher level are included in a section (or subsection, etc.).

All section title styles need not be used, nor must any specific section title style be used. However, a document must be consistent in its use of section titles: once a hierarchy of title styles is established, sections must use that hierarchy.

13.4 Body Elements

13.4.1 Paragraphs

DTD element: paragraph.

Paragraphs consist of blocks of left-aligned text with no markup indicating any other body element. Blank lines separate paragraphs from each other and from other body elements. Paragraphs may contain inline markup (page 84).

Syntax diagram:

```
+------------------------------+
| paragraph                    |
|                              |
+------------------------------+
+------------------------------+
| paragraph                    |
|                              |
+------------------------------+
```

13.4.2 Bullet Lists

DTD elements: bullet_list, list_item.

A text block which begins with a '-', '*', or '+', followed by whitespace, is a bullet list item (a.k.a. 'unordered' list item). For example:

```
- This is the first bullet list item. The blank line above the first
  list item is required; blank lines between list items (such as below
  this paragraph) are optional. Text blocks must be left-aligned,
  indented relative to the bullet.

- This is the first paragraph in the second item in the list.

  This is the second paragraph in the second item in the list. The
  blank line above this paragraph is required. The left edge of this
  paragraph lines up with the paragraph above, both indented relative
  to the bullet.

  - This is a sublist. The bullet lines up with the left edge of the
    text blocks above. A sublist is a new list so requires a blank line
    above and below.

- This is the third item of the main list.

This paragraph is not part of the list.
```

Here are examples of **incorrectly** formatted bullet lists:

```
- This first line is fine.
A blank line is required between list items and paragraphs. (Warning)

- The following line appears to be a new sublist, but it is not:
  - This is a paragraph contination, not a sublist (no blank line).
  - Warnings may be issued by the implementation.
```

Syntax diagram:

```
+------+----------------------+
| '- ' | list item            |
+------| (body elements)+     |
       +----------------------+
```

13.4.3 Enumerated Lists

DTD elements: enumerated_list, list_item.

Enumerated lists (a.k.a. 'ordered' lists) are similar to bullet lists, but use enumerators instead of bullets. An enumerator consists of an enumeration sequence member and formatting, followed by whitespace. The following enumeration sequences are recognized:

- arabic numerals: 1, 2, 3, ... (no upper limit).
- uppercase alphabet characters: A, B, C, ..., Z.
- lower-case alphabet characters: a, b, c, ..., z.
- uppercase Roman numerals: I, II, III, IV, ... (no upper limit).

- lowercase Roman numerals: i, ii, iii, iv, ... (no upper limit).

The following formatting types are recognized:

- suffixed with a period: '1.', 'A.', 'a.', 'I.', 'i.'.
- surrounded by parentheses: '(1)', '(A)', '(a)', '(I)', '(i)'.
- suffixed with a right-parenthesis: '1)', 'A)', 'a)', 'I)', 'i)'.

For an enumerated list to be recognized, the following must hold true:

1. The list must consist of multiple adjacent list items (2 or more).
2. The enumerators must all have the same format and sequence type.
3. The enumerators must be in sequence (i.e., '1.', '3.' is not allowed).

It is recommended that the enumerator of the first list item be ordinal-1 ('1', 'A', 'a', 'I', or 'i'). Although other start-values will be recognized, they may not be supported by the output format.

Nested enumerated lists must be created with indentation. For example:

```
1. Item 1.

   a) item 1a.
   b) Item 1b.
```

13.4.4 Definition Lists

DTD elements: definition_list, definition_list_item, term, definition.

Each definition list item contains a term and a definition. A term is a simple one-line paragraph. A definition is a block indented relative to the term, and may contain multiple paragraphs and other body elements. Blank lines are required before the term and after the definition, but there may be no blank line between a term and a definition (this distinguishes definition lists from block quotes (page 80)).

```
term 1
    Definition 1.

term 2
    Definition 2, paragraph 1.

    Definition 2, paragraph 2.
```

Syntax diagram:

```
+------+
| term |
+--+---+----------------------+
   | defninition              |
   | (body elements)+         |
   +--------------------------+
```

13.4.5 Field Lists

DTD elements: field_list, field, field_name, field_argument, field_body.

Field lists are mappings from field names to field bodies, modeled on RFC822 headers. A field name is made up of one or more letters, numbers, and punctuation, except colons (':') and whitespace. A single colon and whitespace follows the field name, and this is followed by the field body. The field body may contain multiple body elements.

Applications of reStructuredText may recognize field names and transform fields or field bodies in certain contexts. Field names are case-insensitive. Any untransformed fields remain in the field list as the document's first body element.

The syntax for field lists has not been finalized. Syntax alternatives:

1. Unadorned RFC822 everywhere:

   ```
   Author: Me
   Version: 1
   ```

   Advantages: clean, precedent. Disadvantage: ambiguous (these paragraphs are a prime example).

   Conclusion: rejected.

2. Special case: use unadorned RFC822 for the very first or very last text block of a docstring:

   ```
   """
   Author: Me
   Version: 1

   The rest of the docstring...
   """
   ```

   Advantages: clean, precedent. Disadvantages: special case, flat (unnested) field lists only.

   Conclusion: accepted, see below.

3. Use a directive:

   ```
   .. fields::

      Author: Me
      Version: 1
   ```

   Advantages: explicit and unambiguous. Disadvantage: cumbersome.

4. Use Javadoc-style:

   ```
   @Author: Me
   @Version: 1
   @param a: integer
   ```

   Advantages: unambiguous, precedent, flexible. Disadvantages: non-intuitive, ugly.

One special context is defined for field lists. A field list as the very first non-comment block, or the second non-comment block immediately after a title, is interpreted as document bibliographic data. No special syntax is required, just unadorned RFC822. The first block ends with a blank line, therefore field bodies

must be single paragraphs only and there may be no blank lines between fields. The following field names are recognized and transformed to the corresponding DTD elements listed, child elements of the 'document' element. No ordering is imposed on these fields:

- Title: title
- Subtitle: subtitle
- Author/Authors: author
- Organization: organization
- Contact: contact
- Version: version
- Status: status
- Date: date
- Copyright: copyright

This field-name-to-element mapping can be extended, or replaced for other languages. See the implementation documentation for details.

13.4.6 Literal Blocks

DTD element: literal_block.

Two colons ('::') at the end of a paragraph signifies that all following **indented** text blocks comprise a literal block. No markup processing is done within a literal block. It is left as-is, and is typically rendered in a monospaced typeface:

```
This is a typical paragraph. A literal block follows::

    for a in [5,4,3,2,1]:   # this is program code, formatted as-is
        print a
    print "it's..."
    # a literal block continues until the indentation ends

This text has returned to the indentation of the first paragraph, is
outside of the literal block, and therefore treated as an ordinary
paragraph.
```

When '::' is immediately preceeded by whitespace, both colons will be removed from the output. When text immediately preceeds the '::', *one* colon will be removed from the output, leaving only one (i.e., '::' will be replaced by ':'). When '::' is alone on a line, it will be completely removed from the output; no empty paragraph will remain.

In other words, these are all equivalent:

1. Minimized:

   ```
   Paragraph::

       Literal block
   ```

2. Partly expanded:

```
Paragraph: ::

    Literal block
```

3. Fully expanded:

```
Paragraph:

::

    Literal block
```

The minimum leading whitespace will be removed from each line of the literal block. Other than that, all whitespace (including line breaks) is preserved. Blank lines are required before and after a literal block, but these blank lines are not included as part of the literal block.

Syntax diagram:

```
+------------------------------+
| paragraph                    |
| (ends with '::')             |
+------------------------------+
   +--------------------------+
   | literal block            |
   +--------------------------+
```

13.4.7 Block Quotes

DTD element: block_quote.

A text block that is indented relative to the preceeding text, without markup indicating it to be a literal block, is a block quote. All markup processing (for body elements and inline markup) continues within the block quote:

```
This is an ordinary paragraph, introducing a block quote:

    "It is my business to know things. That is my trade."

    --Sherlock Holmes
```

Blank lines are required before and after a block quote, but these blank lines are not included as part of the block quote.

Syntax diagram:

```
+------------------------------+
| (current level of            |
| indentation)                 |
+------------------------------+
   +--------------------------+
   | block quote              |
```

```
| (body elements)+          |
+---------------------------+
```

13.4.8 Tables

DTD elements: table, tgroup, colspec, thead, tbody, row, entry.

Tables are described with a visual outline made up of the characters '-', '=', 'l', and '+'. The hyphen ('-') is used for horizontal lines (row separators). The equals sign ('=') may be used to separate optional header rows from the table body. The vertical bar ('l') is used for vertical lines (column separators). The plus sign ('+') is used for intersections of horizontal and vertical lines.

Each cell contains zero or more body elements. Example:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | Cells may span columns.          |
+------------------------+------------+---------------------+
| body row 3             | Cells may  | - Table cells       |
+------------------------+ span rows. | - contain           |
| body row 4             |            | - body elements.    |
+------------------------+------------+---------------------+
```

As with other body elements, blank lines are required before and after tables. Tables' left edges should align with the left edge of preceeding text blocks; otherwise, the table is considered to be part of a block quote.

13.4.9 Comment Blocks

A comment block is a text block:

- whose first line begins with '.. ' (the 'comment start'),
- whose second and subsequent lines are indented relative to the first, and
- which ends with an unindented line.

Comments are analogous to bullet lists, with '..' as the bullet. Blank lines are required between comment blocks and other elements, but are optional between comment blocks where unambiguous.

The comment block syntax is used for comments, directives, footnotes, and hyperlink targets.

Comments

DTD element: comment.

Arbitrary text may follow the comment start and will be processed as a comment element, possibly being removed from the processed output. The only restriction on comments is that they not use the same syntax as directives, footnotes, or hyperlink targets.

Syntax diagram:

```
+-------+----------------------+
| '.. ' | comment block        |
+--+----+                      |
   |                           |
   +---------------------------+
```

Directives

DTD element: directive.

Directives are indicated by a comment start followed by a single word (the directive type, regular expression '*[a-zA-Z][a-zA-Z0-9_-]**'), two colons, and whitespace. Two colons are used for these reasons:

- To avoid clashes with common comment text like:

  ```
  .. Danger: modify at your own risk!
  ```

- If an implementation of reStructuredText does not recognize a directive (i.e., the directive-handler is not installed), the entire directive block (including the directive itself) will be treated as a literal block, and a warning generated. Thus '::' is a natural choice.

Directive names are case-insensitive. Actions taken in response to directives and the interpretation of data in the directive block or subsequent text block(s) are directive-dependent.

No directives have been defined by the core reStructuredText specification. The following are only examples of *possible uses* of directives.

Directives can be used as an extension mechanism for reStructuredText. For example, here's how a graphic could be placed:

```
.. graphic:: mylogo.png
```

A figure (a graphic with a caption) could be placed like this:

```
.. figure:: larch.png

   The larch.
```

Directives can also be used as pragmas, to modify the behavior of the parser, such as to experiment with alternate syntax.

Syntax diagram:

```
+----------------+------------+
| '.. ' type '::' | directive  |
+--+-------------+ block      |
   |                          |
   +--------------------------+
```

Hyperlink Targets

DTD element: target.

Hyperlink targets consist of a comment start ('.. '), an underscore, the hyperlink name (no trailing underscore), a colon, whitespace, and a link block. Hyperlink targets go together with indirect hyperlinks (page 86) and internal hyperlinks (page 87). Internal hyperlink targets have empty link blocks; they point to the next element. Indirect hyperlink targerts have an absolute or relative URI in their link blocks.

If a hyperlink name contains colons, either:

- the phrase must be enclosed in backquotes:

```
.. _`FAQTS: Computers: Programming: Languages: Python`:
   http://python.faqts.com/
```

- or the colon(s) must be backslash-escaped in the link target:

```
.. _Chapter One\: 'Tadpole Days':

It's not easy being green...
```

Whitespace is normalized within hyperlink names, which are case-insensitive.

Syntax diagram:

```
+-----------------+------------+
| '.. _' name ':' | link       |
+--+--------------+ block      |
   |                           |
   +---------------------------+
```

Footnotes

DTD elements: footnote, label.

Footnotes are similar to hyperlink targets: a comment start, an underscore, open square bracket, footnote label, close square bracket, and whitespace. To differentiate footnotes from hyperlink targets:

- the square brackets are used,
- the footnote label may not contain whitespace,
- no colon appears after the close bracket.

Footnotes may occur anywhere in the document, not necessarily at the end. Where or how they appear in the processed output depends on the output formatter. Here is a footnote, referred to in Footnote References (page 87):

```
.. _[GVR2001] Python Documentation, van Rossum, Drake, et al.,
   http://www.python.org/doc/
```

Syntax diagram:

```
+------------------+----------+
| '.. _[' label ']' | footnote |
+--+---------------+          |
   | (body elements)+          |
   +--------------------------+
```

13.5 Inline Markup

Inline markup is the markup of text within a text block. Inline markup cannot be nested.

There are six inline markup constructs. Four of the constructs (emphasis (page 84), strong emphasis (page 84), interpreted text (page 85), and inline literals (page 85)) use start-strings and end-strings to indicate the markup. The indirect hyperlinks (page 86) construct (shared by internal hyperlinks (page 87)) uses an end-string only. Standalone hyperlinks (page 86) are interpreted implicitly, and use no extra markup.

The inline markup start-string and end-string recognition rules are as follows:

1. Inline markup start-strings must be immediately preceeded by whitespace and zero or more of single or double quotes, '(', '[', or '{'.
2. Inline markup start-strings must be immediately followed by non-whitespace.
3. Inline markup end-strings must be immediately preceeded by non-whitespace.
4. Inline markup end-strings must be immediately followed by zero or more of single or double quotes, '.', ',', ':', ';', '!', '?', '-', ')', ']', or '}', followed by whitespace.
5. If an inline markup start-string is immediately preceeded by a single or double quote, '(', '[', or '{', it must not be immediately followed by the corresponding single or double quote, ')', ']', or '}'.
6. An inline markup end-string must be separated by at least one character from the start-string.
7. Except for the end-string of inline literals (page 85), an unescaped backslash preceeding a start-string or end-string will disable markup recognition. See escaping mechanism (page 74) above for details.

For example, none of the following are recognized as inline markup start-strings: ' * ', '"*"', "'*'", '(*)', '(* ', '[*]', '{*}', '*', ' ' ', etc.

13.5.1 Emphasis

DTD element: emphasis.

Text enclosed by single asterisk characters (start-string = end-string = '*') is emphasized:

```
This is *emphasized text*.
```

Emphasized text is typically displayed in italics.

13.5.2 Strong Emphasis

DTD element: strong.

Text enclosed by double-asterisks (start-string = end-string = '**') is emphasized strongly:

```
This is **strong text**.
```

Strongly emphasized text is typically displayed in boldface.

13.5.3 Interpreted Text

DTD element: interpreted.

Text enclosed by single backquote characters (start-string = end-string = '`') is interpreted:

```
This is `interpreted text`.
```

The semantics of interpreted text are domain-dependent. It can be used as implicit or explicit descriptive markup (such as for program identifiers, as in the Python Extensions to reStructuredText), for cross-reference interpretation (such as index entries), or for other applications where context can be inferred. The role of the interpreted text may be inferred implicitly. The role of the interpreted text may also be indicated explicitly, either a prefix (role + colon + space) or a suffix (space + colon + role), depending on which reads better:

```
`role: interpreted text`
```

```
`interpreted text :role`
```

13.5.4 Inline Literals

DTD element: literal.

Text enclosed by double-backquotes (start-string = end-string = '``') is treated as inline literals:

```
This text is an example of ``inline literals``.
```

Inline literals may contain any characters except two adjacent backquotes in an end-string context (according to the recognition rules above). No markup interpretation (including backslash-escape interpretation) is done within inline literals. Line breaks are *not* preserved; other whitespace is not guaranteed to be preserved.

Inline literals are useful for short code snippets. For example:

```
The regular expression ``[+-]?(\d+(\.\d*)?|\.\d+)`` matches
non-exponential floating-point numbers.
```

13.5.5 Hyperlinks

Hyperlinks are indicated by a trailing underscore, '_', except for standalone hyperlinks (page 86) which are recognized independently.

Standalone Hyperlinks

DTD element: link.

An absolute URI (page 86) within a text block is treated as a general external hyperlink with the URI itself as the link's text (start-string = end-string = '', the empty string). For example:

```
See http://www.python.org for info.
```

would be marked up in HTML as:

```
See <A HREF="http://www.python.org">http://www.python.org</A> for info.
```

Uniform Resource Identifier: URIs are a general form of URLs (Uniform Resource Locators). For the syntax of URIs see RFC2396.

Indirect Hyperlinks

DTD element: link.

Indirect hyperlinks consist of two parts. In the text body, there is a source link, a name with a trailing underscore (start-string = '', end-string = '_'; start-string = ''', end-string = '''_'):

```
See the Python_ home page for info.
```

Somewhere else in the document is a target link containing a URI (see Hyperlink Targets (page 83) for a full description):

```
.. _Python: http://www.python.org
```

After processing into HTML, this should be expressed as:

```
See the <A HREF="http://www.python.org">Python</A> home page for info.
```

See the Python home page for info.

Phrase-links (a hyperlink whose name is a phrase, two or more space-separated words) can be expressed by enclosing the phrase in backquotes and treating the backquoted text as a link name:

```
Want to learn about `my favorite programming language`_?

.. _my favorite programming language: http://www.python.org
```

Want to learn about my favorite programming language?

Whitespace is normalized within hyperlink names, which are case-insensitive.

Internal Hyperlinks

DTD element: link.

Internal hyperlinks connect one point to another within a document. They are identical to indirect hyperlinks (start-string = ‘’, end-string = ‘_’; start-string = ‘‘’, end-string = ‘‘_’) except that there is no URI in the target link. See Hyperlink Targets (page 83) for a full description. For example:

```
Clicking on this internal hyperlink will take us to the target_ below.

.. _target:

The hyperlink target above points to this paragraph.
```

Clicking on this internal hyperlink will take us to the target (page 87) below. The hyperlink target above points to this paragraph.

Footnote References

DTD element: footnote_reference.

Footnote references consist of a square-bracketed label (no whitespace), with a trailing underscore (start-string = ‘[’, end-string = ‘]_’):

```
Please refer to the fine manual [GVR2001]_.
```

See Footnotes (page 83) for the footnote itself.

INDEX

R

S

T

U

W