



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

*Release 2011-08*

**<http://nexusformat.org>**

September 11, 2011



# CONTENTS

<b>I</b>	<b>Preface</b>	<b>3</b>
<b>1</b>	<b>Representation of data examples</b>	<b>7</b>
1.1	Class path specification . . . . .	8
<b>II</b>	<b>Contents</b>	<b>9</b>
<b>2</b>	<b>NeXus: User Manual</b>	<b>11</b>
2.1	NeXus Introduction . . . . .	11
2.2	NeXus Design . . . . .	21
<b>3</b>	<b>NeXus: Reference Documentation</b>	<b>31</b>
<b>4</b>	<b>Cheatsheet</b>	<b>33</b>
4.1	Symbols to mark Sections . . . . .	33
4.2	Other Links . . . . .	34
4.3	Demo list-table . . . . .	34
4.4	Numbered Lists . . . . .	35
4.5	About Linking . . . . .	35
4.6	Missing Links . . . . .	36
<b>5</b>	<b>reStructuredText Markup Specification</b>	<b>37</b>
5.1	Quick Syntax Overview . . . . .	37
5.2	Syntax Details . . . . .	39
<b>III</b>	<b>Indices and tables</b>	<b>55</b>
<b>IV</b>	<b>Documentation Authors</b>	<b>59</b>
	<b>Index</b>	<b>63</b>



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



# **Part I**

## **Preface**





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

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

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



---

# 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:

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

Some words on the notation:

- Hierarchy is represented by indentation. Objects on the same indentation level are in the same group
- The combination `name:NXclass` denotes a NeXus group with name `name` and class `NXclass`.
- A simple name (no following class) denotes a data field. An equal sign is used to show the value, where this is important to the example.
- Sometimes, a data type is specified and possibly a set of dimensions. For example, `energy:NX_NUMBER[NE]` says `energy` is a 1-D array of numbers (either integer or floating point) of length `NE`.
- Attributes are noted as `@name=value` pairs separated by comma. The `@` symbol only indicates this is an attribute. The `@` symbol is not part of the attribute name.
- Links are shown with a text arrow `-->` indicating the source of the link (using HDF5 notation listing the sequence of names).

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

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

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

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

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

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

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

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

## 1.1 Class path specification

In some places in this documentation, a path may be shown using the class types rather than names. For example: `/NXentry/NXinstrument/NXcrystal/wavelength` identifies a dataset called `wavelength` that is inside a group of type `NXcrystal` inside a group of type `NXinstrument` inside a group of type `NXentry`. 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.

This manual is also available in a PDF version.

# **Part II**

## **Contents**



# 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 36).

### 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 12) to help people understand what is in the data files.
2. *A Set of Data Storage Objects* (page 16) (base classes and application definitions) to allow the development of more portable analysis software.
3. *A Set of Subroutines* (page 17) (utilities) to make it easy to read and write NeXus data files.

4. *NeXus Scientific Community* (page 17) 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.

In addition, NeXus relies on a set of low-level file formats to actually store NeXus files on physical media. Each of these components are described in more detail in *fileformat (not converted yet)* (page 36).

The NeXus Application-Programmer Interface (NAPI), which provides the set of subroutines for reading and writing NeXus data files, is described briefly in *NAPI: The NeXus Application Programming Interface* (page 17). (Further details are provided in the NAPI chapter of Volume II of this documentation.)

The principles guiding the design and implementation of the NeXus standard are described in *NeXus Design* (page 21).

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 36).

## A Set of Design Principles

NeXus data files contain four types of entity: data groups, data fields, attributes, and links. See *Data Groups* (page 22) for more details.

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

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

## Example of a NeXus File

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

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



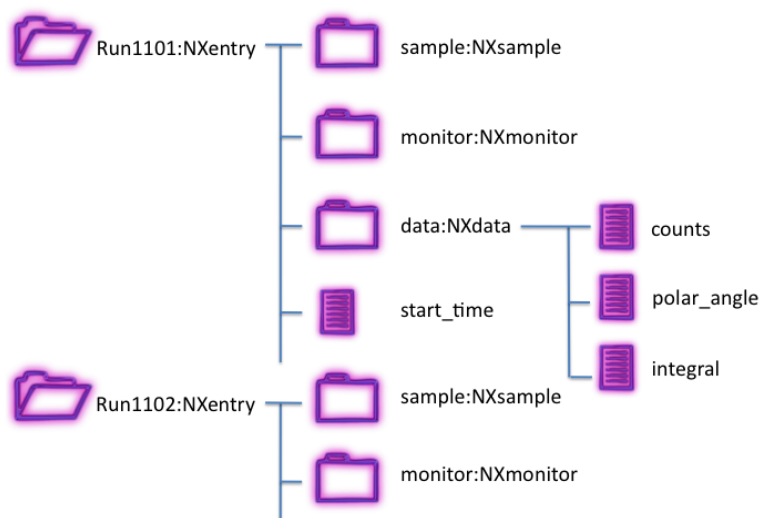


Figure 2.1: Example of a NeXus data file

## 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 21)).

---

**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.

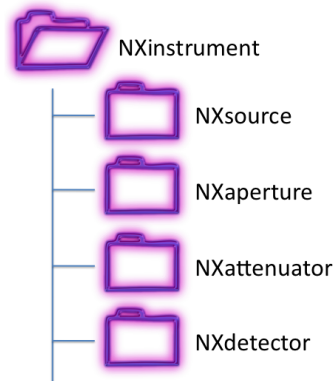


Figure 2.2: NXinstrument excerpt

### 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.)

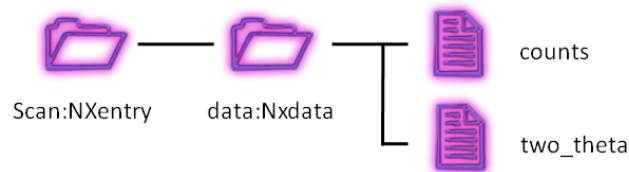


Figure 2.3: Simple Data File Example

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)

```

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

```

```

12         NAPIttype="NX_INT64[15] "
13         long_name="photodiode counts"
14         signal="NX_INT32:1"
15         axes="two_theta">
16             1193      4474
17             53220     274310
18             515430     827880
19             1227100    1434640
20             1330280    1037070
21             598720     316460
22             56677      1000
23             1000
24         </counts>
25         <two_theta
26             NAPIttype="NX_FLOAT64[15] "
27             units="degrees"
28             long_name="two_theta (degrees)">
29             18.90940    18.90960    18.90980    18.91000
30             18.91020    18.91040    18.91060    18.91080
31             18.91100    18.91120    18.91140    18.91160
32             18.91180    18.91200    18.91220
33         </two_theta>
34     </NXdata>
35 </NXentry>
36 </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)**

```

1  #
2  # This example uses NeXpy to build the verysimple.nx5 data file.
3
4  from nexpy.api import nexus
5
6  angle = [18.9094, 18.9096, 18.9098, 18.91, 18.9102,
7           18.9104, 18.9106, 18.9108, 18.911, 18.9112,
8           18.9114, 18.9116, 18.9118, 18.912, 18.9122]
9  diode = [1193, 4474, 53220, 274310, 515430, 827880,
10           1227100, 1434640, 1330280, 1037070, 598720,
11           316460, 56677, 1000, 1000]
12
13  two_theta = nexus.SDS(angle, name="two_theta",
14                        units="degrees",
15                        long_name="two_theta (degrees)")
16  counts = nexus.SDS(diode, name="counts", long_name="photodiode counts")
17  data = nexus.NXdata(counts, [two_theta])
18  data.nxsave("verysimple.nx5")
19
20  # The verysimple.xml file was built with this command:
21  # nxconvert -x verysimple.nx5 verysimple.xml

```

22 *# 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**

```
1  <?xml version="1.0" ?>
2  <definition
3    xmlns="http://definition.nexusformat.org/nxdl/3.1"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://definition.nexusformat.org/nxdl/3.1 ../nxdl.xsd"
6    category="base"
7    name="verysimple"
8    version="1.0"
9    svnId="$Id: introduction.rst 878 2011-09-11 22:34:35Z Pete Jemian $"
10   type="group" extends="NXobject">
11
12   <doc>
13     A very simple NeXus NXDL file
14   </doc>
15   <group type="NXentry">
16     <group type="NXdata">
17       <field name="counts" type="NX_INT" units="NX_UNITLESS">
18         <doc>counts recorded by detector</doc>
19       </field>
20       <field name="two_theta" type="NX_FLOAT" units="NX_ANGLE">
21         <doc>rotation angle of detector arm</doc>
22       </field>
23     </group>
24   </group>
25 </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 36)).

## 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.

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

### 2.1.2 NAPI: The NeXus Application Programming Interface

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

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

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](http://svn.nexusformat.org/code/trunk/doc/api/NeXusIntern.pdf) in the NeXus code repository.<sup>2</sup> 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 14), we walk through some parts of a typical NeXus program written in C.

## Writing a simple NeXus file

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

---

<sup>1</sup> <http://download.nexusformat.org>

<sup>2</sup> <http://svn.nexusformat.org/code/trunk/doc/api/NeXusIntern.pdf>

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

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

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

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

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

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

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

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

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

## How do I read a NeXus file?

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

Here is part of a program to read the two-theta array from the file created by [Writing a simple NeXus file](#) (page 18) 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);
6                  NXmalloc ((void **) &tth, rank, dims, datatype);
7                  NXgetdata (fileID, tth);
8              NXclosedata (fileID);
9          NXclosegroup (fileID);
10 NXclosegroup (fileID);
11 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 17)). 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 20) in this session (see the `nxbrowse` web page):

### Using NXbrowse

```
1  %> nxbrowse lracs3701.nxs
2
3  NXBrowse 3.0.0. Copyright (C) 2000 R. Osborn, M. Koennecke, P. Klosowski
4      NeXus_version = 1.3.3
5      file_name = lracs3701.nxs
6      file_time = 2001-02-11 00:02:35-0600
7      user = EAG/RO
8  NX> dir
9      NX Group : Histogram1 (NXentry)
10     NX Group : Histogram2 (NXentry)
11  NX> open Histogram1
12  NX/Histogram1> dir
13     NX Data  : title[44] (NX_CHAR)
14     NX Data  : analysis[7] (NX_CHAR)
15     NX Data  : start_time[24] (NX_CHAR)
16     NX Data  : end_time[24] (NX_CHAR)
17     NX Data  : run_number (NX_INT32)
18     NX Group : sample (NXsample)
19     NX Group : LRMECS (NXinstrument)
20     NX Group : monitor1 (NXmonitor)
21     NX Group : monitor2 (NXmonitor)
22     NX Group : data (NXdata)
23  NX/Histogram1> read title
24     title[44] (NX_CHAR) = MgB2 PDOS 43.37g 8K 120meV E0@240Hz T0@120Hz
25  NX/Histogram1> open data
26  NX/Histogram1/data> dir
27     NX Data  : title[44] (NX_CHAR)
28     NX Data  : data[148,750] (NX_INT32)
29     NX Data  : time_of_flight[751] (NX_FLOAT32)
30     NX Data  : polar_angle[148] (NX_FLOAT32)
31  NX/Histogram1/data> read time_of_flight
32     time_of_flight[751] (NX_FLOAT32) = [ 1900.000000 1902.000000 1904.000000 ...]
33         units = microseconds
34         long_name = Time-of-Flight [microseconds]
35  NX/Histogram1/data> read data
36     data[148,750] (NX_INT32) = [ 1 1 0 ...]
37         units = counts
38         signal = 1
39         long_name = Neutron Counts
40         axes = polar_angle:time_of_flight
41  NX/Histogram1/data> close
42  NX/Histogram1> close
```



43 NX> quit

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

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

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

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

[line 41] Close the current group.

[line 43] Quits `NXbrowse`.

The source code of `NXbrowse`<sup>3</sup> 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 17)) 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.

### 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:

**Groups** Group data fields and other groups together. Groups represent levels in the NeXus hierarchy

**Data fields** Multidimensional arrays and scalars representing the actual data to be stored

**Attributes** Additional metadata which can be assigned to groups or data fields

<sup>3</sup> <https://svn.nexusformat.org/code/trunk/applications/NXbrowse/NXbrowse.c>

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

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

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

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

## Data Groups

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

## Data Fields

Data fields contain the essential information stored in a NeXus file. They can be scalar values or multidimensional arrays of a variety of sizes (1-byte, 2-byte, 4-byte, 8-byte) and types (integers, floats, characters). The fields may store both experimental results (counts, detector angles, etc), and other information associated with the experiment (start and end times, user names, etc). Data fields are identified by their names, which must be unique within the group in which they are stored.

## 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 *NeXus Data Attributes* (page 22). Finally, NeXus files themselves have global attributes which are listed in the *NeXus File Global Attributes* (page 23) table that identify the NeXus version, file creation time, etc. Attributes are identified by their names, which must be unique in each field.

## NeXus Data Attributes

For the full specification of attributes, see *volume2.NXDL.section (not converted yet)* (page 36).

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 <i>signal</i> =”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 <i>data[j][i]</i> (C) or <i>data(i,j)</i> (Fortran), with dimension scales <i>time_of_flight[i]</i> and <i>polar_angle[j]</i> , then data would have an attribute <i>axes</i> =” <i>polar_angle:time_of_flight</i> ” in addition to an attribute <i>signal</i> =”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 <i>nxdata-structure (not converted yet)</i> (page 36) structure), i.e. if the array being stored is data, with elements <i>data[j][i]</i> in C and <i>data(i,j)</i> in Fortran, axis would have the following values: ith dimension ( <i>axis</i> =”1”), jth dimension ( <i>axis</i> =”2”), etc.
primary	NX_INT	Defines the order of preference for dimension scales which apply to the same rank of signal data. Use <i>primary</i> =”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	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 than replicating the data, NeXus uses links in such situations. See the figure *Linking in a NeXus file*. (page 24) for a more descriptive representation of the concept of linking.

### Linking in a NeXus file.

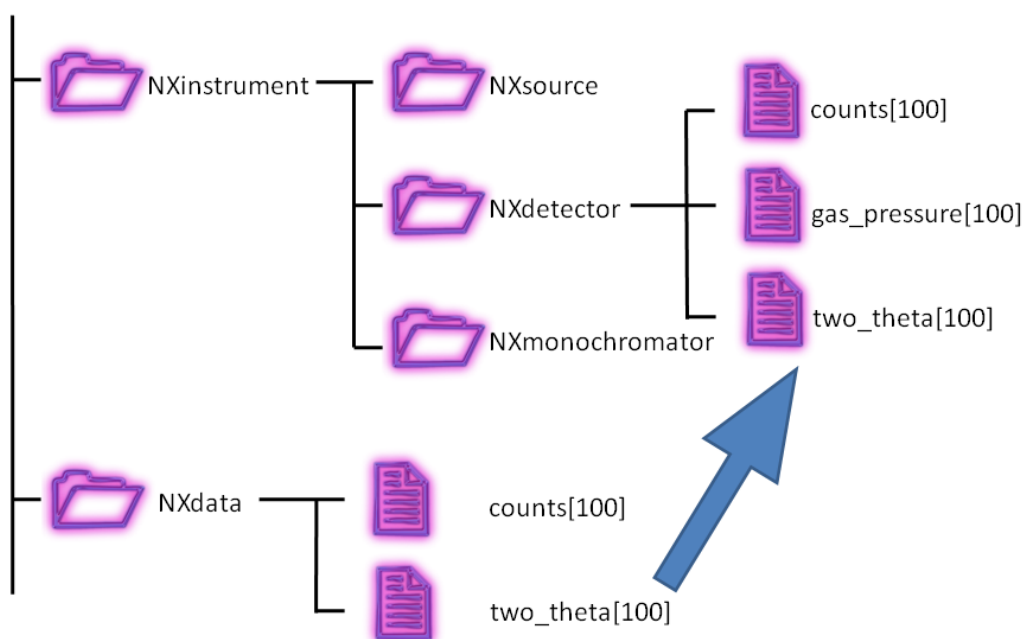


Figure 2.4: Linking in a NeXus file. See example code below: *example.data-linking (not converted yet)* (page 36)

## NeXus Base Classes

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

Not all classes define physical objects. Some refer to logical groupings of experimental information, such as

plottable data, sample environment logs, beam profiles, etc. There can be multiple instances of each class. On the other hand, a typical NeXus file will only contain a small subset of the possible classes.

NeXus base classes are not proper classes in the same sense as used in object oriented programming languages. In fact the use of the term classes is actually misleading but has established itself during the development of NeXus. NeXus base classes are rather dictionaries of field names and their meanings which are permitted in a particular NeXus group implementing the NeXus class. This sounds complicated but becomes easy if you consider that most NeXus groups describe instrument components. Then for example, a *NXmonochromator* base class describes all the possible field names which NeXus allows to be used to describe a monochromator.

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

***NXdata*** *NXdata* is used to identify the default plottable data. The notion of a default plot of data is a basic motivation of NeXus.

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

***NXnote*** *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.

## **NeXus 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
  1. 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.
1. 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.

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*<sup>4</sup> 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<sup>5</sup> 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*<sup>1</sup> 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 27).

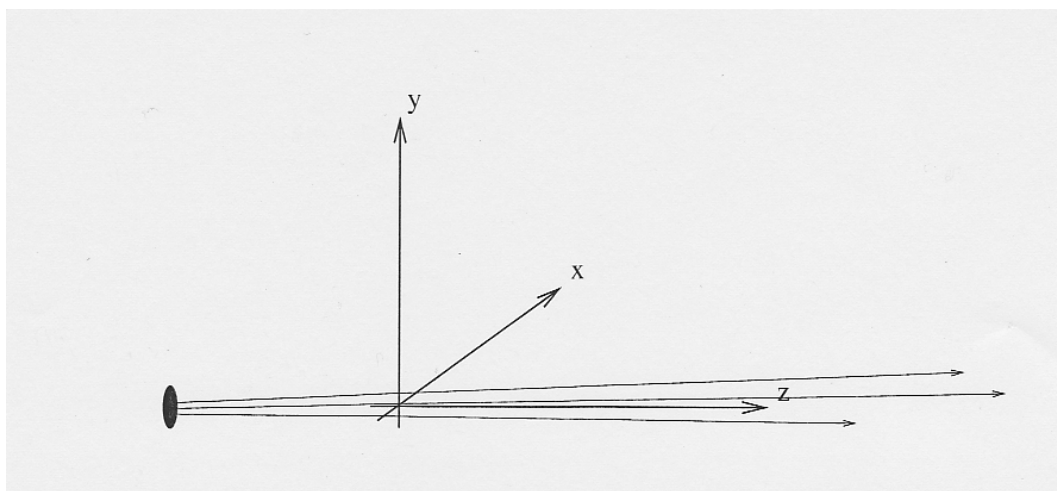


Figure 2.5: The McStas Coordinate System

**Note:** The NeXus definition of  $+z$  is opposite to that in the International Tables for Crystallography, volume G,<sup>6</sup> and consequently,  $+x$  is also reversed.

<sup>4</sup> McStas, <http://www.mcstas.org>, also <http://mcstas.risoe.dk>

<sup>5</sup> CIF (Crystallographic Information Framework), <http://www.iucr.org/resources/cif>

<sup>6</sup> **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



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 28).

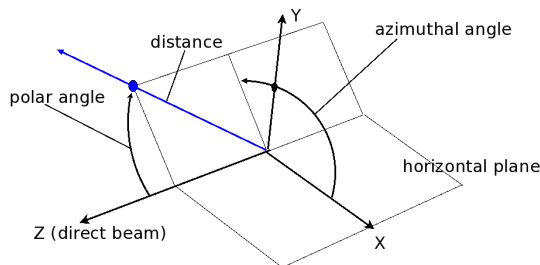


Figure 2.6: NeXus Simple (Spherical Polar) Coordinate System

### Coordinate Transformations

Another way to look at coordinates is through the use of transformation matrices. In this world view, the absolute position of a component or a detector pixel with respect to the laboratory coordinate system is calculated by applying a series of translations and rotations. These operations are commonly expressed as transformation matrices and their combination as matrix multiplication. A very important aspect is that the order of application of the individual operations *does* matter. Another important aspect is that any operation transforms the whole coordinate system and gives rise to a new local coordinate system. The mathematics behind this is well known and used in such applications such as industrial robot control, space flight and



computer games. The beauty in this comes from the fact that the operations to apply map easily to instrument settings and constants. It is also easy to analyze the contribution of each individual operation: this can be studied under the condition that all other operations are at a zero setting.

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

**Type** The type of operation: rotation or translation

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

**Value** The angle of rotation or the length of the translation

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

#### 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 29). 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 *transform* field to get the detector as a whole into its position and a *transform\_pixel* field which describes how the absolute position of a detector pixel can be calculated.

For the NeXus spherical coordinate system, the order is implicit and is given 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*.



# NEXUS: REFERENCE DOCUMENTATION

Contents:



# 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 34)

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

## 4.1 Symbols to mark Sections

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) = \frac{2}{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')
```

## 4.2 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/>

## 4.3 Demo list-table

Does this work?

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

Table 4.1: 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!

## 4.4 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.

## 4.5 About Linking

What about a link to *Indirect Hyperlinks* (page 52) 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 11).

## 4.6 Missing Links

These sections show up as missing links in *NeXus Introduction* (page 11).

Can you find the [history](#) (page 36) link below? What about the [history](#) (page 36) link below? This works: *history (not converted yet)* (page 36) (or *history (not converted yet)* (page 36)).

### 4.6.1 history (not converted yet)

### 4.6.2 fileformat (not converted yet)

### 4.6.3 utilities (not converted yet)

### 4.6.4 nxdl\_tutorial-creatingnxdlspec (not converted yet)

### 4.6.5 nxdata-structure (not converted yet)

### 4.6.6 volume2.NXDL.section (not converted yet)

### 4.6.7 example.data-linking (not converted yet)

### Section to cross-reference

This is the text of the section.

It refers to the section itself, see *Section to cross-reference* (page 36). What about a section on another page, such as *Footnote References* (page 53)?



---

# RESTRUCTUREDTEXT MARKUP SPECIFICATION

---

Author: David Goodger Contact: [dgoodger@bigfoot.com](mailto:dgoodger@bigfoot.com) Version: 0.2 Date: 2001-05-29

`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 39) section.

[Literal blocks](#) (page 45) are used for examples throughout this document.

## 5.1 Quick Syntax Overview

A `reStructuredText` document is made up of body elements, and may be structured into sections. [Section Structure](#) (page 40) is indicated through title style (underlines & optional overlines). Sections contain body elements and/or subsections.

Here are examples of body elements:

- [Paragraphs](#) (page 41) (and [inline markup](#) (page 50)):

```
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 42):

- This is a bullet list.
- Bullets can be '-', '\*', or '+'.

2. [Enumerated lists](#) (page 42):

1. This is an enumerated list.
2. Enumerators may be arabic numbers, letters, or roman numerals.

3. [Definition lists](#) (page 43):

```
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 45):

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 46):

Block quotes consist of indented body elements:

```
This theory, that is mine, is mine.

Anne Elk (Miss)
```

• [Tables](#) (page 47):

```
+-----+-----+-----+
| 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 48):

```
.. 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 48):

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

- [Footnotes](#) (page 49):

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

```
    It is a form of hyperlink target.
```

- [Hyperlink targets](#) (page 49):

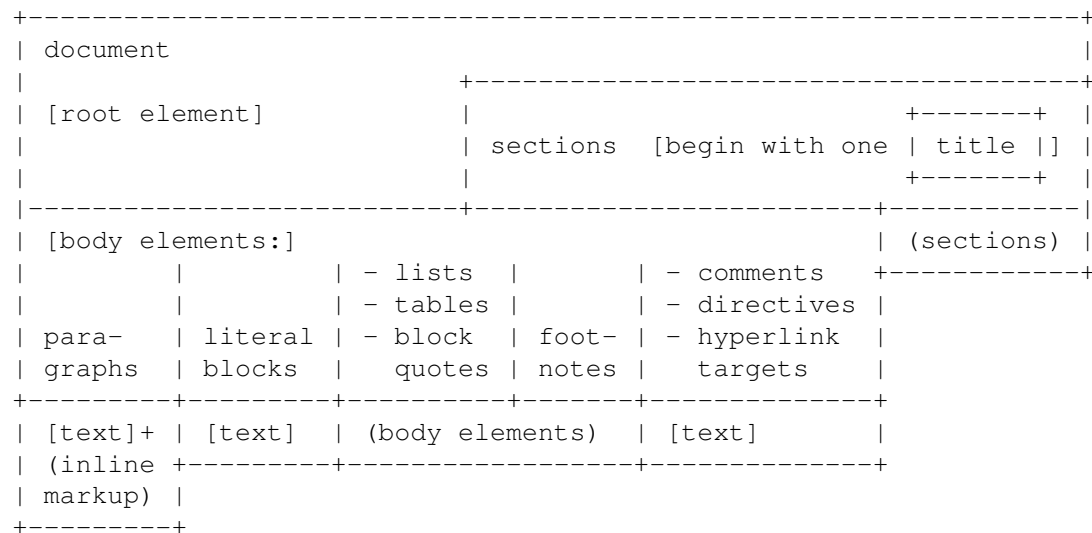
```
.. _Python: http://www.python.org
```

```
.. _example:
```

The ‘\_example’ target above points to this paragraph.

## 5.2 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, [gpdtd.dtd](#). Descriptions below list ‘DTD elements’ (XML ‘generic identifiers’) corresponding to syntax constructs.

### 5.2.1 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.

### 5.2.2 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.

### 5.2.3 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.

## 5.2.4 Body Elements

### 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 50).

Syntax diagram:

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

## 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 continuation, not a sublist (no blank line).
  - Warnings may be issued by the implementation.

Syntax diagram:

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

## 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.
```

## 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 46)).

```
term 1
    Definition 1.

term 2
    Definition 2, paragraph 1.

    Definition 2, paragraph 2.
```

Syntax diagram:

```
+-----+
| term |
+---+---+-----+
| definition |
```

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

## 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.

## **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 |
+-----+
```

## 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)+  |
+-----+

```

## Tables

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

Tables are described with a visual outline made up of the characters '-', '=', '|', 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 ('|') 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 preceding text blocks; otherwise, the table is considered to be part of a block quote.

## 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 52) and [internal hyperlinks](#) (page 53). Internal hyperlink targets have empty link blocks; they point to the next element. Indirect hyperlink targets 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 53):

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

Syntax diagram:

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

### 5.2.5 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 50), [strong emphasis](#) (page 51), [interpreted text](#) (page 51), and [inline literals](#) (page 51)) use start-strings and end-strings to indicate the markup. The [indirect hyperlinks](#) (page 52) construct (shared by [internal hyperlinks](#) (page 53)) uses an end-string only. [Standalone hyperlinks](#) (page 52) 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 51), an unescaped backslash preceeding a start-string or end-string will disable markup recognition. See [escaping mechanism](#) (page 40) above for details.

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

### 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.

## 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.

## 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`
```

## 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.

## Hyperlinks

Hyperlinks are indicated by a trailing underscore, ‘\_’, except for [standalone hyperlinks](#) (page 52) which are recognized independently.

### Standalone Hyperlinks

DTD element: link.

An absolute [URI](#) (page 52) 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 49) 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 49) 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 53) 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 49) for the footnote itself.



## **Part III**

# **Indices and tables**



- *genindex*
- *search*



## **Part IV**

# **Documentation Authors**





These people have made substantial contributions to the NeXus manual:

- Ray Osborn, Argonne National Laboratory, <rosborn@anl.gov>
- Mark Koennecke, Paul Scherrer Institut, <Mark.Koennecke@psi.ch>
- Przemek Klosowski, U. of Maryland and NIST, <przemek.klosowski@nist.gov>
- Frederick Akeroyd, Rutherford Appleton Laboratory, <freddie.akeroyd@stfc.ac.uk>
- Peter F. Peterson, Spallation Neutron Source, <peterpsonpf@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>



# INDEX

## A

- attributes
  - data, [12](#), [19](#)
- authors, [61](#)
- automatic plotting, [25](#)

## C

- CIF, [26](#)
- classes
  - base class: NXdata, [13](#)
  - base class: NXentry, [13](#)
  - base class: NXinstrument, [13](#)
  - base class: NXsample, [13](#)
- coordinates
  - transformations, [28](#)

## D

- data
  - examples, [7](#)
- data objects
  - attributes, [12](#), [22](#)
  - global, [22](#)
  - data items, [22](#)
  - fields, [12](#), [21](#), [22](#)
  - groups, [12](#), [22](#)
- default plot, [25](#)
- dimension scale, [25](#)

## E

- example
  - simple, [17–19](#), [21](#)
  - very simple, [15](#), [17](#)

## F

- file
  - browse, [20](#)
  - read, [19](#)

- write, [18](#)

## G

- geometry, [26–28](#)

## H

- HDF
  - Scientific Data Sets, [12](#)
- hierarchy, [12](#), [21](#), [22](#)
  - example NeXus data file, [12](#)

## I

- instrument definitions, [16](#)

## L

- link, [12](#), [24](#)

## M

- McStas, [26](#), [27](#)

## N

- NAPI, [12](#), [17–19](#)
- NeXus, [11](#)
  - Design Principles, [12](#)
- NeXus basic motivation, [25](#)
  - default plot, [12](#), [13](#), [19](#), [22](#), [24](#), [25](#)
  - unified format, [11](#)
- NXDL, [24](#)

## R

- rank, [19](#)
- rules, [11](#)
  - HDF, [22](#)
  - naming, [24](#)

## S

- Scientific Data Sets, [12](#)

## T

target, [24](#)

## U

units, [12](#), [19](#), [22](#)

utility

    nxbrowse, [20](#)