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

Release 2011-08

http://nexusformat.org

September 11, 2011

CONTENTS

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

Part I

Preface

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

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

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

CHAPTER

ONE

REPRESENTATION OF DATA EXAMPLES

Most of the examples of data files have been written in a format intended to show the structure of the file rather than the data content. In some cases, where it is useful, some of the data is shown. Consider this prototype example:

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

Some words on the notation:

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

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

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

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

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

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

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

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

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

1.1 Class path specification

In some places in this documentation, a path may be shown using the class types rather than names. For example: `/NXentry/NXinstrument/NXcrystal/wavelength` identifies a dataset called `wavelength` that is inside a group of type `NXcrystal` inside a group of type `NXinstrument` inside a group of type `NSentry`. This nomenclature is used when the exact name of each group is either unimportant or not specified. Often, this will be used in a NXDL specification to indicate the connections of a link.

This manual is also available in a PDF version.

Part II

Contents

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)* (*cheatsheet:history).

quote

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

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

2.1.1 What is NeXus?

The NeXus data format has four components:

1. *A Set of Design Principles* (*volume1/introduction:introduction-designprinciples) to help people understand what is in the data files.
2. *A Set of Data Storage Objects* (*volume1/introduction:introduction-datastorageobjects) (base classes and application definitions) to allow the development of more portable analysis software.
3. *A Set of Subroutines* (*volume1/introduction:introduction-setofsubroutines) (utilities) to make it easy to read and write NeXus data files.

4. *NeXus Scientific Community* (*volume1/introduction: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.

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)* (*cheatsheet:fileformat).

The NeXus Application-Programmer Interface (NAPI), which provides the set of subroutines for reading and writing NeXus data files, is described briefly in *NAPI: The NeXus Application Programming Interface* (*volume1/introduction:introduction-napi). (Further details are provided in the NAPI chapter of Volume II of this documentation.)

The principles guiding the design and implementation of the NeXus standard are described in *design (not converted yet)* (*cheatsheet:design).

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

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

A Set of Design Principles

NeXus data files contain four types of entity: data groups, data fields, attributes, and links. See *design-groups (not converted yet)* (*cheatsheet:design-groups) for more details.

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

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

Example of a NeXus File

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

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

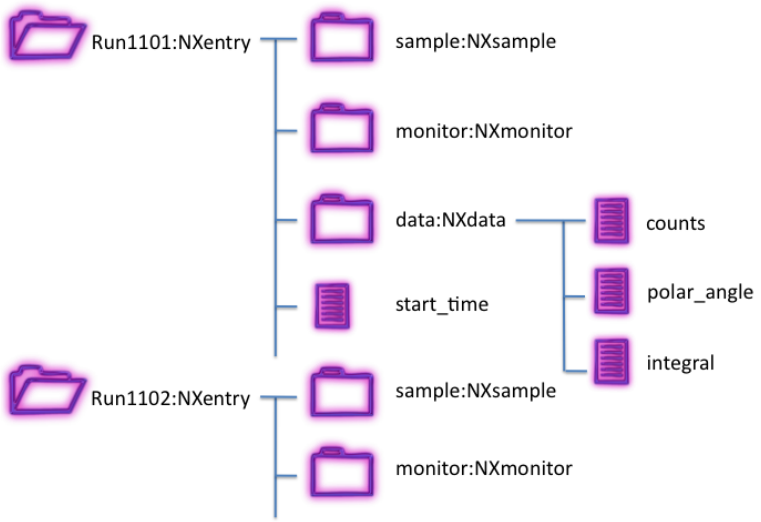


Figure 2.1: Example of a NeXus data file

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 (*design (not converted yet)* (*cheatsheet:design)).

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

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

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

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

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

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

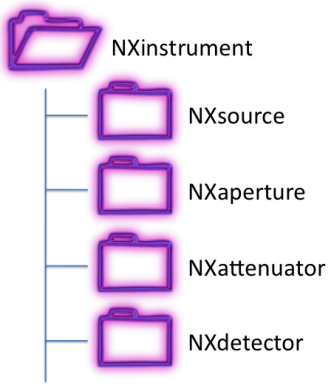


Figure 2.2: NXinstrument excerpt

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: introduction.rst 875 2011-09-11 18:34:50Z 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)* (*cheatsheet:nxdl-tutorial-creatingnxdlspec)).

A Set of Subroutines

NeXus data files are high-level so the user only needs to know how the data are referenced in the file but does not need to be concerned where the data are stored in the file. Thus, the data are most easily accessed using a subroutine library tuned to the specifics of the data format.

In the past, a data format was defined by a document describing the precise location of every item in the data file, either as row and column numbers in an ASCII file, or as record and byte numbers in a binary file. It is the job of the subroutine library to retrieve the data. This subroutine library is commonly called an application-programmer interface or API.

For example, in NeXus, a program to read in the wavelength of an experiment would contain lines similar to the following:

Simple example of reading data using the NeXus API

```
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

Note: **TODO:** Show how these work together.

- NIAC
- NeXus Wiki
- ...

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. [1]

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

How do I write a NeXus file?

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

The API uses a very simple *state* model to navigate through a NeXus file. When you open a file, the API provides a file *handle*, which then stores the current location, i.e. which group and/or field is currently open. Read and write operations then act on the currently open entity. Following the simple example of *fig.simple-example (not converted yet)* (*cheatsheet:fig-simple-example), we walk through some parts of a typical NeXus program written in C.

Writing a simple NeXus file

```
#include "napi.h"

 int main()
 {
        NXhandle fileID;
        NXopen ('NXfile.nxs', NXACC_CREATE, &fileID);
          NXmakegroup (fileID, "Scan", "NXentry");
          NXopengroup (fileID, "Scan", "NXentry");
                NXmakegroup (fileID, "data", "NXdata");
                NXopengroup (fileID, "data", "NXdata");
                /* somehow, we already have arrays tth and counts, each length n*/
                  NXmakedata (fileID, "two_theta", NX_FLOAT32, 1, &n);
                  NXopendata (fileID, "two_theta");
                        NXputdata (fileID, tth);
                        NXputattr (fileID, "units", "degrees", 7, NX_CHAR);
                  NXclosedata (fileID);  /* two_theta */, NX_INT32, 1, &n);
                  NXopendata (fileID, "counts");
                        NXputdata (fileID, counts);
                  NXclosedata (fileID);  /* counts */
                NXclosegroup (fileID);  /* data */
          NXclosegroup (fileID);  /* Scan */
        NXclose (&fileID);
```

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

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

Using `NXbrowse`

```
%> nxbrowse lrcs3701.nxs

NXBrowse 3.0.0. Copyright (C) 2000 R. Osborn, M. Koennecke, P. Klosowski
        NeXus_version = 1.3.3
        file_name = lrcs3701.nxs
        file_time = 2001-02-11 00:02:35-0600
        user = EAG/RO
NX> dir
  NX Group : Histogram1 (NXentry)
  NX Group : Histogram2 (NXentry)
NX> open Histogram1
NX/Histogram1> dir
  NX Data  : title[44] (NX_CHAR)
  NX Data  : analysis[7] (NX_CHAR)
  NX Data  : start_time[24] (NX_CHAR)
  NX Data  : end_time[24] (NX_CHAR)
  NX Data  : run_number (NX_INT32)
  NX Group : sample (NXsample)
  NX Group : LRMECS (NXinstrument)
  NX Group : monitor1 (NXmonitor)
  NX Group : monitor2 (NXmonitor)
  NX Group : data (NXdata)
NX/Histogram1> read title
  title[44] (NX_CHAR) = MgB2 PDOS 43.37g 8K 120meV E0@240Hz T0@120Hz
NX/Histogram1> open data
NX/Histogram1/data> dir
  NX Data  : title[44] (NX_CHAR)
  NX Data  : data[148,750] (NX_INT32)
  NX Data  : time_of_flight[751] (NX_FLOAT32)
  NX Data  : polar_angle[148] (NX_FLOAT32)
NX/Histogram1/data> read time_of_flight
  time_of_flight[751] (NX_FLOAT32) = [ 1900.000000 1902.000000 1904.000000 ...]
        units = microseconds
        long_name = Time-of-Flight [microseconds]
NX/Histogram1/data> read data
  data[148,750] (NX_INT32) = [ 1 1 0 ...]
        units = counts
        signal = 1
        long_name = Neutron Counts
        axes = polar_angle:time_of_flight
NX/Histogram1/data> close
```

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

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

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

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

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

[line 41] Close the current group.

[line 43] Quits `NXbrowse`.

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

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

CHAPTER

THREE

NEXUS: REFERENCE DOCUMENTATION

Contents:

CHAPTER

FOUR

CHEATSHEET

This is a cheat sheet and will be removed later.

Section headings automatically get labels assigned. For example, see this: Demo list-table (*cheatsheet:demo-list-table)

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) = {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' )
```

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* (*markup-spec:indirect-hyperlinks) 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* (*volume1/index:nexus-user-manual).

4.6 Missing Links

These sections show up as missing links in *NeXus Introduction* (*volume1/introduction:nexus-introduction).

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

4.6.1 history (not converted yet)

4.6.2 fileformat (not converted yet)

4.6.3 design (not converted yet)

4.6.4 utilities (not converted yet)

4.6.5 design-groups (not converted yet)

4.6.6 nxdl_tutorial-creatingnxdlspec (not converted yet)

4.6.7 fig.simple-example (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* (*cheatsheet:my-reference-label). What about a section on another page, such as *Footnote References* (*markup-spec:footnote-references)?

CHAPTER

FIVE

RESTRUCTUREDTEXT MARKUP SPECIFICATION

Author: David Goodger Contact: 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 (*markup-spec:syntax-details) section.

Literal blocks (*markup-spec:literal-blocks) 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 (*markup-spec:section-structure) is indicated through title style (underlines & optional overlines). Sections contain body elements and/or subsections.

Here are examples of body elements:

- Paragraphs (*markup-spec:paragraphs) (and inline markup (*markup-spec:inline-markup)):

```
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 (*markup-spec:bullet-lists):

```
- This is a bullet list.

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

2. Enumerated lists (*markup-spec:enumerated-lists):

```
1. This is an enumerated list.

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

3. Definition lists (*markup-spec:definition-lists):

```
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 (*markup-spec:literal-blocks):

```
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 (*markup-spec:block-quotes):

```
Block quotes consist of indented body elements:

    This theory, that is mine, is mine.

    Anne Elk (Miss)
```

- Tables (*markup-spec:tables):

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

- Comments (*markup-spec:comments):

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

- Directives (*markup-spec:directives):

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

- Footnotes (*markup-spec:footnotes):

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

   It is a form of hyperlink target.
```

- Hyperlink targets (*markup-spec:hyperlink-targets):

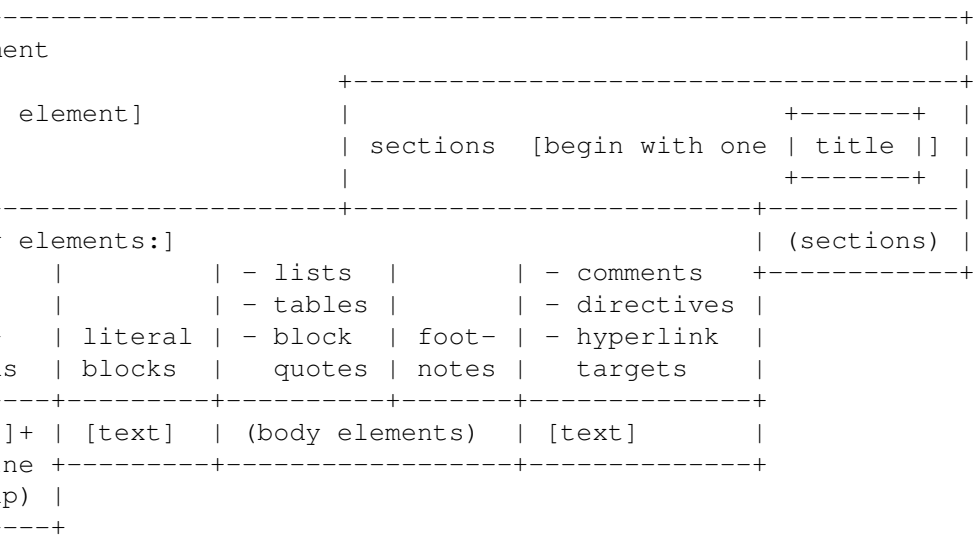
```
.. _Python: http://www.python.org

.. _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, gpdi.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 (*markup-spec:inline-markup).

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 contination, 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 (*markup-spec:block-quotes)).

```
term 1
    Definition 1.

term 2
    Definition 2, paragraph 1.

    Definition 2, paragraph 2.
```

Syntax diagram:

```
+------+
| term |
+--+---+----------------------+
   | defninition               |
```

```
| (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 '-', '=', '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.

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, hhere'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 (*markup-spec:indirect-hyperlinks) and internal hyperlinks (*markup-spec:internal-hyperlinks). 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 (*markup-spec:footnote-references):

```
.. _[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 (*markup-spec:emphasis), strong emphasis (*markup-spec:strong-emphasis), interpreted text (*markup-spec:interpreted-text), and inline literals (*markup-spec:inline-literals)) use start-strings and end-strings to indicate the markup. The indirect hyperlinks (*markup-spec:indirect-hyperlinks) construct (shared by internal hyperlinks (*markup-spec:internal-hyperlinks)) uses an end-string only. Standalone hyperlinks (*markup-spec:standalone-hyperlinks) 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 (*markup-spec:inline-literals), an unescaped backslash preceeding a start-string or end-string will disable markup recognition. See escaping mechanism (*markup-spec:escaping-mechanism) 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 (*markup-spec:standalone-hyperlinks) which are recognized independently.

Standalone Hyperlinks

DTD element: link.

An absolute URI (*markup-spec:uri) 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 (*markup-spec:hyperlink-targets) 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 (*markup-spec:hyperlink-targets) 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 (*markup-spec:target) 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 (*markup-spec:footnotes) 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, <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>