

libpniio **Users Guide**

Eugen Wintersberger

March 13, 2017

Contents

1. Introduction	9
2. Installation	11
2.1. Install precompiled packages	11
2.2. Install from sources	11
2.2.1. Running tests	11
3. Using the library	13
3.1. From the command line	13
3.2. From within a Makefile	13
3.3. With CMake	13
3.4. Dealing with updates and bug fixes	14
4. Legacy format support	15
4.1. ASCII data	15
4.1.1. Lowlevel parser interface	15
4.2. Binary data	17
5. Getting started with NeXus	19
5.1. The Nexus layer model	20
5.1.1. Layer 1 objects	20
5.1.2. Layer 2 objects	21
5.1.3. Layer 3 objects	21
6. Addressing NeXus object - the NeXus-path	23
6.1. Introduction	23
6.1.1. The structure of a NeXus-path	23
6.1.2. Some general path properties	24
6.1.3. Equality of two NeXus-paths	24
6.1.4. Matching paths	24
6.1.5. Examples	26
6.2. The <code>nxpath</code> type	27
6.2.1. Path construction	28
6.2.2. Path iteration	28
6.2.3. Push and pop on object	28
6.3. Utility functions	29
6.3.1. Element utilities	29
6.3.2. <code>nxpath</code> utilities	30
6.4. The grammar of a Nexus path	31

7. Basic usage of libpniio	33
7.1. Working with files	33
7.1.1. Creating single files	33
7.1.2. Create distributed files	33
7.1.3. Opening and closing files	34
7.1.4. Other file related functions	35
7.2. Working with groups	35
7.2.1. Creating groups	35
7.2.2. Accessing children	36
7.2.3. Other group related member functions	36
7.3. Working with fields	37
7.3.1. Creating fields	37
7.3.2. Reading and writing data	37
7.3.3. Growing fields	38
7.3.4. Partial reading and writing	39
7.3.5. Field inquiry	40
7.4. Working with attributes	40
7.4.1. Creating attributes	40
7.4.2. Attribute inquiry	41
7.4.3. Accessing an objects attributes	41
7.4.4. Reading and writing data from and to attributes	42
7.4.5. Attribute management	43
7.5. Working with links	43
7.5.1. Create internal links	44
7.5.2. Create external links	45
7.5.3. Link inquiry	46
8. libpniio in more detail	47
8.1. The mysterious <code>nxobject</code>	47
8.2. Using algorithms	48
8.2.1. Basic inquiry and conversion	48
8.2.2. Common algorithms for fields, groups, and attributes	49
8.2.3. Group related algorithms	51
8.2.4. Field and attribute related algorithms	54
8.3. Iterating groups	58
8.3.1. Simple iteration	58
8.3.2. Recursive iteration	58
8.4. Deleting items	59
8.5. Custom field chunks	59
8.5.1. What are chunks?	59
8.5.2. Setting the chunk shape	60
9. NeXus ASCII representation	63
9.1. Use cases	63
9.1.1. Generating NeXus structures from XML	63
9.1.2. Metadata ingestion of a file	63
9.2. Necessary limitations of an ASCII representation	63

9.3. NeXus and XML	64
9.3.1. Basic XML handling	64
9.3.2. High level XML interface	64
9.3.3. The XML low level interface	66
A. Parsing ASCII data	71
A.1. ASCII representations of primitive data types	71
A.1.1. Integers and floating point numbers	71
A.1.2. Complex numbers	71
A.1.3. Boolean values	72
A.2. Parser rules	72
B. NeXus-XML protocol	73
B.1. Common deviations from NXDL	73
B.1.1. Data types	73
B.1.2. Enumerations	73
B.1.3. Data in XML content	73
B.2. XML attribute representation	74
B.2.1. Scalar attributes	74
B.2.2. Multidimensional attributes	74
B.3. XML field representation	75
B.3.1. Scalar fields	75
B.3.2. Multidimensional fields	75
B.3.3. Adding chunking	75
B.3.4. Adding compression	75

Todo list

fix quote	20
maybe we should present some example here	24
To be implemented	53
To be implemented!	54

1. Introduction

`libpniio` is the IO library within the PNI library stack. More precisely it is responsible for file IO. Although several legacy formats are supported the library mainly deals with Nexus files providing functions and objects to deal with these kind of files.

Chapter 4 gives an overview over the supported legacy formats and how to access data. It is important to note that the legacy support is mainly for reading data. The reason for this is that newly created data should be written as Nexus files.

Chapter 5 gives a quick overview over Nexus and provides a kind of quick start tutorial how to use the library. In the ongoing chapters present more detailed information about how to work with Nexus files with `libpniio`.

2. Installation

2.1. Install precompiled packages

2.2. Install from sources

2.2.1. Running tests

In order to run the tests use

```
> make cleanup test
```

The `cleanup` target removes the test artifacts from previous test runs. This is important as some of the tests may fail if the old artifacts are still present.

3. Using the library

In this section we will have a short look on how to make your code working the `libpniio`. The key to make using `libpniio` simple is the usage of `pkg-config`. The rational behind the design decision to focus on `pkg-config` as the central element for build systems is simple: it works for virtually all build systems. It is even available for Windows (though not very often used).

3.1. From the command line

```
$> g++ -std=c++11 -otest test.cpp $(pkg-config --cflags --libs pniio)
```

There are two important remarks we have to make here. The first is the `-std=c++11` after `g++`. This tells the compiler to use the new C++11 standard. This option is absolutely required for the code to build. the `pkg-config` command at the end of the command line includes all the necessary compiler and linker flags to build and link the code.

3.2. From within a Makefile

`pkg-config` can be used in a Makefile by putting the following at the top of your Makefile

```
CPPFLAGS=-O2 -g -std=c++11 $(shell pkg-config --cflags pniio)
LDFLAGS=$(shell pkg-config --libs pniio)
```

3.3. With CMake

For `cmake` the `FindPkgConfig` module provides access to the functionality of `pkg-config`. The following snippet from a `CMakeLists.txt` file shows how to use it for `libpniio`

```
#load pkg-config package
include(FindPkgConfig)

#search for the pniio library
pkg_search_module(PNIIO REQUIRED pniio)
link_directories(${PNIIO_LIBRARY_DIRS})
include_directories(${PNIIO_INCLUDE_DIRS})
add_definitions(${PNIIO_CFLAGS})

set(SOURCE ...)

add_executable(myprog ${SOURCE})
target_link_libraries(myprog ${PNIIO_LIBRARIES})
```

3. *Using the library*

3.4. Dealing with updates and bug fixes

Like `libpnicore`, `libpniio` is for most of its parts a template library. This ensures good performance but comes at a price. Unlike binary only libraries programs have to be rebuilt when a new version of `libpniio` shall be used.

4. Legacy format support

The term legacy data refers to all non-Nexus file formats. `libpnicore` distinguishes between two families of legacy formats

- ASCII file where the content is entirely stored in human readable ASCII characters
- and binary data where the raw binary information is stored in a file.

4.1. ASCII data

4.1.1. Lowlevel parser interface

`libpniio` provides a low level parser interface based on the `boost::spirit` spirit framework. The major job of this interface is to provide save number parsing. It supports all primitive data types provided by `libpnicore` along with `std::vector` begin a container of a primitive type. For a detailed explanation about the low level parsers see Appendix A.

At the heart of the parser API is the `parser` class template. It takes one template parameter which is the primitive or container type to parse. To use the parser API just include `pni/io/parsers.hpp` in your source file.

Parsing primitive scalars

A very simple example would be something like this

```
#include <iostream>
#include <pni/core/types.hpp>
#include <pni/io/parsers.hpp>

using namespace pni::core;
using namespace pni::io;

typedef parser<float64> float64_parser_type;

int main(int argc, char **argv)
{
    float64_parser_type p;

    float64 data = p("1.234");
    std::cout<<data<<std::endl;

    return 0;
}
```

This example should be rather self explaining. When used with scalar values the parser template provides only a default constructor. No additional information is required to configure the parser code.

4. Legacy format support

Besides primitive types the `parser` template can also be used with the `value` type erasure. In this case the resulting parser matches either a `int64`, a `float64`, or a `complex64` type. Again no additional configuration at parser instantiation is required. For the ASCII representation of complex numbers see Appendix A.

Parsing a vector of primitives

Besides single scalars the `parser` template can also be used with `std::vector` based containers where the element type should be one of the primitive types or a value. For this purpose a specialization of the `parser` template of the form

```
template<typename T> class parser<std::vector<T>> {...};
```

is provided. A particularly interesting choice as an element is the `value` type erasure as it allows to parse a series of inhomogeneous types. The following program

```
#include <iostream>
#include <vector>
#include <pni/core/types.hpp>
#include <pni/io/parsers.hpp>

using namespace pni::core;
using namespace pni::io;

typedef std::vector<value> record_type;
typedef parser<record_type> record_parser;

int main(int argc, char **argv)
{
    record_parser p;
    record_type data = p("1.234 12 1+I3.4");
    for(auto v: data)
        std::cout<<v.type_id()<<std::endl;

    return 0;
}
```

would produce this output

```
FLOAT64
INT64
COMPLEX64
```

When using the default constructor of the `parser` template with a container type the individual elements are considered to be separated by at least one blank. However the vector parser specialization of the `parser` template provides three more additional constructors. The first, `parser(char del)` allows to use a custom delimiter symbol. In the next example the `' , '` is used as a delimiter for the individual elements

```
record_parser p(',',');
record_type data = p("1.234,12 , 1+I3.4");
```

It is important to not that the delimiter symbol can be surrounded by an arbitrary number of blanks. The second constructor provides the constructor with additional start and stop symbols.

```
record_parser p(['[', ']']);
record_type data = p("[1.234 12 1+I3.4]");
```

However, the elements in the string are now again separated only by blanks. Full customization of the parser is provided by the third constructor which allows the user to provide not only start and stop symbols but also a custom delimiter symbol

```
record_parser p(['[', ']'], ';');
record_type data = p("[1.234;12 ; 1+I3.4]");
```

4.2. Binary data

5. Getting started with NeXus

Today, data recorded during synchrotron experiments is typically stored in individual binary image files and/or as flat ASCII files. Figure 5.1 shows the typical directory structure of such a setup. The ASCII file stores scalar data while the detector data is stored in a separate directory as image files (here TIFF). Such an approach leads to technical and organizational problems

1. when the number of image files grows large the performance of most file systems degenerate
2. to access data in an individual image file a new file handler has to be created
3. image and scalar data is stored in different files which increases the managements efforts to keep related information together.

NeXus is a binary file format which attempts to solve all of these problems. NeXus can keep scalar and multidimensional data within a single file and allows to organize the data within the file in a tree like mannger. Additional attributes can be attached to each object in a file, storing metadata which might be required for later analysis. It must be noted that NeXus is not a physical file format itself. It is rather a set of rules how data must be organized within a particular format in order to become a valid Nexus file. Currently the following physical file formats are supported by the original Nexus API

- XML – only used for file structure validation
- HDF4 – for historical reasons, should not be used for new data
- HDF5 – the current standard storage backend for Nexus files.

One of the aims of `libpniio` is to provide an abstraction layer between the user and the storage backend. As `libpniio` currently supports only HDF5 this is rather artificial. However,

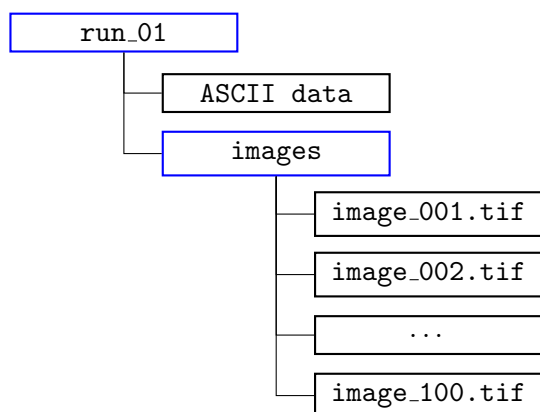


Figure 5.1.: A typical directory structure used at todays synchrotron experiments. Scalar data is stored in a single ASCII file while detector data is stored as individual image files in a separate directory.

5. Getting started with NeXus

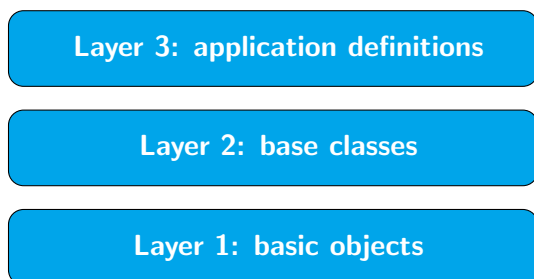


Figure 5.2.: Nexus can be considered to consist of three layers where each layer represents a particular level of abstraction.

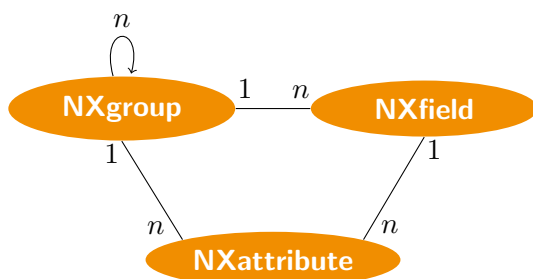


Figure 5.3.: The basic objects of the first layer in the Nexus object model and their relation to each other.

`libpniio` provides the architecture to include other file formats to be used with NeXus too. There has been a lot of confusion what physical file format NeXus files are. Many users think that NeXus has its own physical file format. This is in fact not true. Just to avoid any further confusion for the reader let me make this clear once and for all

Every NeXus file written by `libpniio` is also a valid HDF5 file!

This quote needs better formatting. Maybe the text should go into a box and the margins to the surrounding text must be bigger

5.1. The Nexus layer model

As already mentioned in the previous section, NeXus can be considered as a set of rules describing the logical organization of data within a file. NeXus, for this purpose, utilizes types defined in three different layers as shown in Fig. 5.2. Each group of types defined in a layer is used for a particular purpose. Layer 1 provides the basic brick-stones from which the NeXus standard is build of. Layer 2 provides collections of Layer 1 instances representing concrete beamline components as well as abstract concepts. Layer 3 finally can be used to define file formats for particular experimental techniques.

5.1.1. Layer 1 objects

The first layer in the NeXus layer model provides the fundamental types used to define the organizational structure of data in a file. Figure 5.3 shows an overview of relation between the basic Layer 1 objects types.

NXgroup	is a container which can hold instances of fields and other groups.
NXfield	stores numerical and other data.
NXattribute	instances of NXgroup and NXfield can be enhanced with attributes which can store additional metadata about an object.

Attributes behave a little like fields as will be shown later. These three types form the basement for all other objects in the above layers. It should be mentioned that attributes are heavily used by the NIAC to add metadata to a group or field. It is thus not wise to make too extensive use of attributes as a Nexus user as it may cause name clashes with future attributes defined by the NIAC.

The focus of `libpniio` is this first layer of the NeXus hierarchy.

5.1.2. Layer 2 objects

Objects from the second layer are composites of the types provided by the first layer. They describe logical or physical entities at a beamline. These range from the very concrete objects like undulators or detectors to abstract concepts like *entries* and *subentries*. The available objects from Layer 2 are defined in the Nexus Base class catalogue.

5.1.3. Layer 3 objects

The third layer finally provides concepts to standardize the structure of a Nexus data tree for particular scientific applications and methods. These so called *Application Definitions* are defined by the NIAC in collaboration with the scientific community.

6. Addressing NeXus object - the NeXus-path

Objects within a NeXus-file can be referenced by a path. Though being very similar to a Unix file system path, a NeXus-path provides much more flexibility. It reflects one of the key features of NeXus: types. A NeXus-path can reference an object not only via its name (as HDF5 does) but also by its type. Hence, under certain conditions, it is possible to construct a path which is independent of the names chosen within a file.

6.1. Introduction

6.1.1. The structure of a NeXus-path

Figure 6.1 shows the principal structure of a NeXus-path as used by `libpnio`. Such a path comprises three major sections

<i>file section</i>	which references the NeXus-file on the file system. It must thus be a valid file system path on the operating system platform in use.
<i>object section</i>	describing the location of an object within the file
<i>attribute section</i>	referencing an attribute attached to the object pointed to by the residual path. The attribute is identified by its name.

As shown in Fig. 6.1 the file and the object sections are separated by `://` while the object and attribute sections use `@` as a delimiter. Both, the file and the attribute section, are optional. The individual elements in the object section are separated by a single `/`. Every element in the *object section* is composed of two parts (see Fig. 6.2): the *name section* and the *class section* separated by a `..`. Whether or not the *name section* and/or the *class section* must be present in order to reference an element depends on the circumstances. There are three possible situations

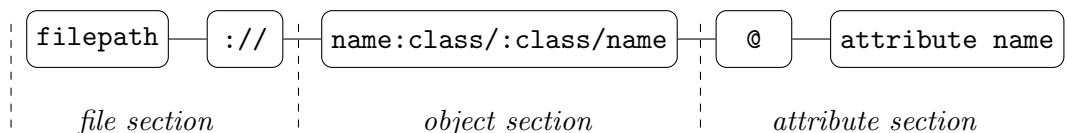


Figure 6.1.: The basic structure of a NeXus path as used by `libpnio`. The *file section* stores the Unix path to the data file. The *object section* the path to the field or group within the file and the *attribute section* holds the name of an attribute attached to the object referenced by the previous path.

6. Addressing NeXus object - the NeXus-path

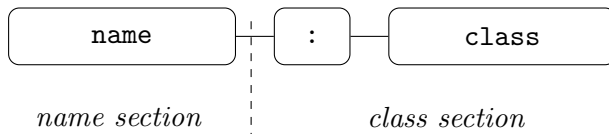


Figure 6.2.: Structure of the elements in the *object section* of a NeXus-path.

<code>name:class</code>	this is a full identifier for a group. It determines its name as well as its type. As fields have no type they cannot be referenced by such an expression.
<code>name</code>	if only the name is given the referenced object can be either a group or a field. However, in the case of a field, this must be the last element in the object path (as fields cannot have additional children).
<code>:class</code>	if only the class is given the referenced object must be a group. Denote the leading colon in this expression. It is necessary to distinguish such an expression from a mere name.

6.1.2. Some general path properties

A path is considered as *absolute* if its *object section* starts at the root group of the file. This is always the case if

- the *file section* of the path is not empty
- or, if no *file section* is given, the *object section* starts with a leading `/`.

The latter condition is equivalent to the convention used for Unix file system paths while the former requires some explanation. If the *file section* is not empty the *object section* has to be considered absolute otherwise we would not know where to start searching for objects. If no *file section* is provided the path can also refer to an object relative to a particular parent object.

The NeXus-path implementation provided by `libpnio` also understands `.` and `..` where the former one refers to the current group while the latter one to the parent group of the current group.

6.1.3. Equality of two NeXus-paths

Two NeXus-paths are considered to be equal if all of their elements are equal.

6.1.4. Matching paths

Two paths are considered as *matching* if one can deduce from their structure that they reference the same object within a NeXus-file. It is important to realize that this question must be answered independent of a particular file. Only the path object itself is of relevance. This leads to some surprising effects. Consider the following three paths

```
a = /entry/instrument/detector/data
b = /entry:NXentry/instrument:NXinstrument/detector:NXdetector/data
c = /:NXentry/:NXinstrument/:NXdetector/data
```

maybe
we
should
present
some ex-
ample
here

It is obvious for path **a** and **b** that they reference the same object. The same is true for the paths **b** and **c**. Surprisingly, **a** and **c** do not match. As **a** does not provide any type information for each of its nodes we cannot be sure that it references the same object as **c**. Thus, *matching* is not the same as equality. If a **match** would be the same as equality we would get

$$a = b \wedge b = c \text{ but } a \neq c \quad (6.1)$$

which, from a mathematical point of view, makes no sense. and reason about which of those paths are matching (according to the above definition). Before diving deeper in the matching paths problem lets first discuss what is it good for.

Applications for path matching

Consider a file which stores several instances of **NXdetector** within its **NXinstrument** group and that this file has also several entries (in other words, several measurements are stored in a single file). Furthermore we assume that we would have a hypothetical function **match(const nxpath &a, const nxpath &b)** which returns true if the two paths **a** and **b** match and false otherwise. Consider the case where we would like to obtain the detector groups for all entries in the file. This could easily be done with the following piece of code

```
typedef std::vector<h5::nxobject> detectors_type;
typedef std::back_inserter<detectors_type> detector_inserter;

detectors_type detectors;
detector_inserter inserter(detectors);

nxpath pref = nxpath::from_string("/:NXentry/:NXinstrument/:NXdetector");
h5::nxfile f = ....;

auto flat_root = make_flat(f.root());

std::copy_if(flat_root.begin(), flat_root.end(), inserter,
             [&pref](const h5::nxobject &o) { return match(pref, get_path(o)); });
```

Another situation would be that we would like to know how many entries (measurements) are stored in a particular file.

```
nxpath pref = nxpath::from_string("/:NXentry");
h5::nxfile f = ....;
h5::nxgroup root = f.root();
size_t nentries = std::count_if(root.begin(), root.end(),
                                [&pref](const h5::nxobject &o)
                                { return match(get_path(o), pref); });
```

All three paths could be used to address the data field in the detector group of the file. However, it would be difficult to prove only from the paths themselves that this is the case. While $a = b$ and $b = c$ is relatively simple, what about $a = c$? While **a** does not provide any type information, **c** has all the names removes (except for the name of the field). The only thing **a** and **c** have in common is the name of the field they refer to.

An easier approach might be to ask for the equality of two elements, **a** and **b**, of the *object section* of a path. The obvious case for equality is if

6. Addressing NeXus object - the NeXus-path

a and b are considered to be equal if their name and class strings are equal.

For instance, let $a = (\text{entry}, \text{NXentry})$ and $b = (\text{entry}, \text{NXentry})$. According to the previous rule $a = b$.

Furthermore, we can propose a second rule

a and b can be considered equal if their class component is equal and only one of them has the name attribute set.

This would be the case if $a = (, \text{NXentry})$ and $b = (\text{entry}, \text{NXentry})$. This is somehow logical if we consider that a is just a more general version of b . However, it is crucial that only one of them has a non empty name attribute. Otherwise this rule would violate rule one.

The third rule states

If a and b have both either their name *or* their class attribute set and those are equal.

For names a and b would be equal for instance of $a(\text{entry},)$ and $b = (\text{entry},)$. The same is true for the class attribute. a and b are equal if $a = (, \text{NXentry})$ and $b = (, \text{NXentry})$.

In all other cases a and b would be not equal. For instance $a \neq b$ if $a = (\text{entry},)$ and $b = (, \text{NXentry})$. It is also clear that for fields (which have only a name) the name must be equivalent to be considered as equal.

This rules also solve the above problem. Indeed $a = b$ and $b = c$ but $a \neq c$. This may sounds awkward from a mathematical point of view. But it has several advantages as will be shown later. The comparison operators for `nxpath::element_type` are implemented following the above rules.

6.1.5. Examples

Let's have a look on some examples. The following path addresses the data field in the detector group of a file

```
/data/run/detector.nxs://entry/instrument/detector/data
```

Here, the individual groups are referenced by their name in the object section of the path. Indeed, this path can be written in a more general way with

```
/data/run/detector.nxs://:NXentry/:NXinstrument/:NXdetector/data
```

where the parent groups of the `data` field are referenced implicitly via their type. This requires that only one instance of a particular type (`:NXentry`, `:NXinstrument`, etc.) exists in its parent group. In the case that we have two detectors and each of them is stored as an instance of `NXdetector` below the `NXinstrument` group, the name of the detector must be provided explicitly

```
/data/run/detector.nxs://:NXentry/:NXinstrument/det1:NXdetector/data
```

The last group reference `det1:NXdetector` is the most precise description of a group instance. Not only does it determined the name of the group but also its type. This example already shows one of the powers of NeXus. As long as only one instance of a particular type exists within a group it can be identified by its type rather than by its name. In many situations

it is thus possible to generate paths which are virtually independent of all object names (in fact only the fields must be named as they have no type).

All path examples until now represented an absolute path (a path with a leading *file section*). In many situations no file must be specified. A typical application for paths without *file section* would be program where an object should be referenced by a path relative to a given parent object. The path in the next example references the data field of the detector relative to the top level instance of `NXentry`

```
:NXinstrument/detector/data
```

In order to make a path without a *file section absolute*, it must start with a leading `/` as in the next example

```
/:NXentry/:NXinstrument/pilatus/data
```

In order to reference the root group of a file one can either use

```
/
```

a single `/` or, in case of a file section

```
/data/run/detector.nxs://
```

where the trailing `://` denotes the root group. In case of an absolute path the root group is always included in the path object (as will be shown later).

Finally an application for `..` should be discussed. Lets assume that the current parent is the detector group and we want to address the diameter field of an instance of `NXpnihole` located one level above. We could do this with

```
../:NXpnihole/diameter
```

where `..` indicates that one should move to the parent group of the current one.

6.2. The *nxpath* type

In C++ a NeXus-path is represented by an instance of `nxpath`. To use `nxpath` and its utility functions the appropriate header file must be included

```
#include <pni/io/nx/nxpath.hpp>
```

`nxpath` is an iterable over the elements of the *object section* of a NeXus-path. The optional *file-* and *attribute section* can be accessed via getter and setter methods like this

```
nxpath path = ...;
path.filename("/data/run/detector.nxs"); //set file section
std::cout<<path.filename()<<std::endl;  //retrieve file section
```

and analogously for the *attribute section*

```
nxpath path = ...;
path.attribute("units"); //set attribute section
std::cout<<path.units()<<std::endl;  //retrieve attribute section
```

6. Addressing NeXus object - the NeXus-path

The elements of the *object section* are stored as instances of `std::pair<string,string>` where the first element of the pair holds the name of the element and the second the class (if available). `nxpath` provides an alias to the element type via the public member type `nxpath::element_type`. Technically, `nxpath` is a thin wrapper around a list of such `element_type` (although not all the list functionality is exported). Consult the API documentation for a detailed description of `nxpath`'s interface.

6.2.1. Path construction

Though the `nxpath` type has a constructor one would typically construct a path from a string using the `from_string` static member method

```
nxpath path = nxpath::from_string("/:NXentry:NXinstrument/pilatus");
```

`from_string` has also a static counterpart method `to_string` which converts a path instance to its string representation.

```
nxpath path = ....;
std::cout<<nxpath::to_string(path)<<std::endl;
```

6.2.2. Path iteration

`nxpath` provides an STL compliant iterator interface which allows easy iteration over all elements in the *object section* of the path. Consider the following example

```
nxpath p = nxpath::from_string("/:NXentry:NXinstrument/pilatus/data");

for(auto e:p)
    std::cout<<"name: "<<e.first<<"\t type:"<<e.second<<std::endl;
```

which would yield the output

```
name: /          type: NXroot
name:           type: NXentry
name:           type: NXinstrument
name: pilatus type:
name: data      type:
```

As we can see from the above example: the first member of the `std::pair<string,string>` stored in the object section list is the name of an object while the second is its type. In the case of a field only the first (name) element will be set (a field does not have a particular type). The number of elements in the *object section* of `nxpath` can be obtained via the `size` member function (which is the same as for any other STL container).

6.2.3. Push and pop on object

Elements of the *object section* of the path can be added using the `push_back` and `push_front` member functions.

```
nxpath p = nxpath::from_string(":NXinstrument");
std::cout<<p<<std::endl; // output: :NXinstrument
```

```

p.push_back(object_element("", "NXdetector"));
std::cout<<p<<std::endl; // output: :NXinstrument/:NXdetector

p.push_front(object_element("", "NXentry"));
std::cout<<p<<std::endl; // output: :NXentry/:NXinstrument/:NXdetector

Like other STL containers nxpath also provides the front, back, pop_front, and pop_back
member functions which have the standard STL behavior.

nxpath p = nxpath::from_string(":NXentry/:NXinstrument/:NXdetector");

//get front and back elements from the object section
nxpath::element_type entry = p.front();
nxpath::element_type detector = p.back();

std::cout<<p<<std::endl; // output: :NXentry/:NXinstrument/:NXdetector

//remove front and back objects from the object section
p.pop_front();
p.pop_back();

std::cout<<p<<std::endl; // output: :NXinstrument

```

6.3. Utility functions

6.3.1. Element utilities

There are a couple of utility functions available to work with the elements stored in the *object section* of the path. One important function is the `object_element` function which creates a single element for the *object section* of a path. This is particularly useful in connection with the `push_back` and `push_front` member functions of `nxpath`. If for instance one wants to append a detector group to the object section we could use

```

nxpath p = ...;
p.push_back(object_element("detector", "NXdetector"));

```

`object_element` takes two arguments: the first is the name of the object while the second its type (only relevant for groups). If both are empty strings and exception will be thrown.

Furthermore there are some functions for querying the basic properties of an element instance. Each of these functions returns a boolean value and takes an instance of `nxpath::element_type` as its only argument.

<code>is_root_element</code>	returns true if the element references the root group (with name / and type NXroot)
<code>is_complete</code>	return true if the element has a non-empty name and type
<code>has_name</code>	return true if the element has a non-empty name
<code>has_class</code>	return true if the element has a non-empty type

6. Addressing NeXus object - the NeXus-path

6.3.2. nxpath utilities

Three inquiry functions exist for `nxpath`. Each of them returns a boolean and takes as their single argument a reference to an instance of `nxpath`

<code>is_absolute</code>	returns <code>true</code> if the path is an absolute path
<code>has_file_section</code>	returns <code>true</code> if the path has a non-empty file section
<code>has_attribute_section</code>	returns <code>true</code> if the path has a non-empty attribute section
<code>is_empty</code>	returns <code>true</code> if a path has neither a <i>file section</i> , an <i>attribute section</i> , and an <i>object section</i> . This situation would be equivalent to a default constructed path object.

The `split_path` function divides an `nxpath` into two partial paths at a user defined position.

```
string s = "test.nxs://:NXentry/:NXinstrument/detector@NX_class";
nxpath p = nxpath::from_string(s);
nxpath instrument_path, detector_path;
split_path(p, 3, instrument_path, detector_path);

// output: test.nxs://:NXentry/:NXinstrument
std::cout<<instrument_path<<std::endl;
// output: detector@NX_class
std::cout<<detector_path<<std::endl;
```

The second argument to `split_path` is the position where to perform the split. It is the index of the first element for the second path. To chop of the *file section* from a path one could use the following code

```
string s = "test.nxs://:NXentry/:NXinstrument/detector@NX_class";
nxpath p = nxpath::from_string(s);
nxpath instrument_path, detector_path;
split_path(p, 0, instrument_path, detector_path);

// output: test.nxs
std::cout<<instrument_path<<std::endl;
// output: /:NXentry/:NXinstrument/detector@NX_class
std::cout<<detector_path<<std::endl;
```

Two paths can be joined using the `join()` function.

```
nxpath a = nxpath::from_string("file.nxs://:NXentry/:NXinstrument");
nxpath b = nxpath::from_string("pilatus300k:NXdetector/data");
nxpath c = join(a,b);
std::cout<<c<<std::endl;

//would output
//file.nxs://:NXentry/:NXinstrument/pilatus300k:NXdetector/data"
```

There are several restrictions to the two path arguments `a` and `b` passed to the `join()` function

- a must not have an *attribute section*
- b must not have a *file section*
- b must not be an absolute path.

If any of these restrictions are violated `join()` throws `value.error`. There are additional special conditions which should be taken into account and where the above rules do not apply

a empty, b not	→	return b unchanged
b empty, a not	→	return a unchanged
a and b empty	→	return an empty path object

6.4. The grammar of a Nexus path

Lets first have a look on the grammar of a Nexus path in EBNF¹

```
file_path = {all characters allowed by the plattform to describe a path}
```

```
(* definition of character sets*)
```

```
valid_char = "_" | "a-z" | "A-Z" | "0-9";
```

```
whitespace = " " | "\n" | "\r";
```

```
(*definition of required terminal symbols*)
```

```
class_seperator = ":";
```

```
object_seperator = "/";
```

```
current_group = ".";
```

```
parent_group = "..";
```

```
(*a nexus ID must not be empty*)
```

```
nexus_id = valid_char,{valid_char};
```

```
nexus_name = nexus_id,(class_seperator|group_separtor|whitespace);
```

```
nexus_group = group_seperator,nexus_id,[group_seperator|whitespace];
```

```
(*
```

```
the first part is the object name, the second the group class if the
object is a group
```

```
*)
```

```
object_id = nexus_name
           | nexus_name,nexus_group
           | nexus_group
           | current_group
           | parent_Gruop
```

```
object_path ::= ["/"],object_id,{"/",object_id};
```

```
nexus_path ::= [file_path,"://"],object_path,["@ ",nexus_attr];
```

¹EBNF=Extended Backus Naur Form

6. Addressing NeXus object - the NeXus-path

The `file_path` is platform dependent which makes it difficult to determine which characters would be allowed in a path. Thus we leave this open to and separate the file path from everything else by a `://` string germinal. `nexus_id` describes a repetition of a set of characters allowed in Nexus names (for groups, fields, attributes, and classes). It is much more restrictive as for the filename.

7. Basic usage of libpniio

This chapter deals with the basic interface provided by the layer 1 types implemented in `libpniio`. All types concerning Nexus reside in one of the namespaces embedded in `pni::io::nx`. The namespaces below this one indicate either a particular storage backend (currently only HDF5 is implemented).

To use the Nexus part of the library just add

```
#include <pni/io/nx/nx.hpp>
```

to your source file.

7.1. Working with files

7.1.1. Creating single files

The simplest approach towards handling NeXus-files is to create a single file to store data. This can be achieved with the `create_file` static member function of `nxfile`.

```
#include <pni/io/nx/nx.hpp>
```

```
using namespace pni::io::nx;
```

```
int main(int argc, char **argv)
{
    h5::nxfile file = h5::nxfile::create_file("test.nxs");
    //... code omitted ...
    file.close();

    return 0;
}
```

The code should be rather self explaining. If the file already exists a `object_error` exception is thrown. In order to overwrite an existing file one can use

```
h5::nxfile file = h5::nxfile::create_file("test.nxs", true);
```

where the second argument to `create_file` enables overwriting an existing file of same name. This option should be used with care as all data stored in the original file will be lost forever.

7.1.2. Create distributed files

In cases where a single data file would grow rather large (more than 40 GByte for instance) creating a single large file is not a good solution. One problem is the transfer of the file via the network. It would require a quite sophisticated down- or upload software which must be able to recover a transfer from a broken network connection, for instance. The other problem

7. Basic usage of `libpniio`

comes from archives. Data which should be archived goes typically to a tape library. However, such libraries typically want to have files in a particular size in order to operate with optimal performance.

`libpniio` allows the content of a single file to be distributed over several files each having the same size. Such a set of files can be created using the `create_files` static member function as shown below

```
h5::nxfile file = h5::nxfile::create_files("test.%04i.nxs",1024);
```

Aside from its name the arguments of the `create_files` function have a slightly different meaning. If a set of files should be produced the file name is not a simple string but a `printf` like format string. This allows the storage backend of `libpniio` to number each new file as it is created. The second argument to this function is the size in MByte an individual file can attain before a new one will be created. The above call to `create_files` would yield the following files

```
test.0001.nxs
test.0002.nxs
test.0003.nxs
...
```

As for the simple `create_file`, `create_files` will throw an `object_error` exception if a file already exists. In order to overwrite an existing file append `true` to the above call

```
h5::nxfile file = h5::nxfile::create_files("test.%04i.nxs",1024,true);
```

However, in this case already existing members of the family will not be removed but just truncated (their size becomes 0). So do not wonder that you still find all the member files of a set even after overwriting it. Their size will be set to zero.

7.1.3. Opening and closing files

If a file already exist the `open_file` static member function of the `nxfile` should be used. Its signature is rather simple

```
open_file(const string &n,bool ro=true)
```

where the first argument is the name of the file and the second determines whether or not the file will be opened in read-only mode. By default files are opened read-only in order to avoid accidental changes in the file.

`open_file` can be used with a single file as well as with a file family. For a single file use

```
h5::nxfile f = h5::nxfile::open_file("test.nxs");
```

In order to open a file split into several parts only a different file name must be used

```
h5::nxfile f = h5::nxfile::open_file("test.%05i.nxs");
```

Like for file creation, the `printf`-like format string has to be used for the filename.

Like all objects in `libpniio` a file object is destroyed automatically if it loses scope. However, in some cases one may want to explicitly close the file. This can be done with the `close` member function

```
h5::nxfile f = ...;
.... code omitted ...
f.close();
```

7.1.4. Other file related functions

Like virtually all level 1 objects in `libpniio nxfile` possesses an `is_valid` inquiry method. It can be used to check whether or not an object is a valid instance or not. This is necessary as a default constructed file is not a valid instance.

```
h5::nxfile f = ...;
...code omitted ...
if(!f.is_valid())
    std::cerr<<"Something went wrong!"<<std::endl;
```

You can also check whether a file is read-only or not by means of the `is_readonly` member function

```
h5::nxfile f = ...;
...code omitted ...
if(f.is_readonly())
    std::cerr<<"File is in read-only mode!"<<std::endl;
```

As one can see from the API documentation, the interface of `nxfile` is rather simple. In order to do anything useful (like creating groups and fields) one has to obtain the root group of the file. This can be done with the `root` member function

```
h5::nxfile f = ...;
h5::nxgroup root = f.root();
```

Finally there is an important member function named `flush`. Whenever possible use this function to explicitly hand over data from the underlying storage library to the operating system for writing.

```
h5::nxfile f =....;

while(measurement_running())
{
    //record data

    //flush the file
    f.flush();
}
```

7.2. Working with groups

NeXus groups are instances of the `nxgroup` template. They can be considered as containers for fields and other groups and expose an STL compliant interface. To start working with groups in a file one has to first obtain the root group with

```
h5::nxfile file = h5::nxfile::open_file("test.nxs");
h5::nxgroup root = file.root();
```

7.2.1. Creating groups

New groups are created by means of the `create_group` member function of `nxgroup`

7. Basic usage of *libpniio*

```
h5:nxgroup entry = root.create_group("scan_1", "NXentry");
```

This method takes two arguments where the first one is mandatory and denotes the name of the group while the second one is optional and determines the NeXus-class of the group. If the last argument is omitted a simple HDF5 group is created (without an `NX_class` attribute).

Like files, groups are automatically destroyed when an instance loses scope, but they can also be deliberately closed using their `close()` method.

7.2.2. Accessing children

Access to the direct children of a group instance is given via the `at()` method or the `[]` operator. Both accept either a numeric index of a child or its name as an argument. To loop over all children of the root group the following code could be used

```
h5:nxfile f = ....;
h5:nxgroup root = f.root();

for(size_t i=0; i<root.size(); ++i) std::cout<<root[i].name()<<std::endl;
```

As for STL containers, the `size()` method returns the number of children of a group. To access a particular group via its name one can use

```
h5:nxfile f = ....;
h5:nxgroup root = f.root();

h5:nxgroup entry = root["entry"]; //alternatively root.at("entry");
```

Unlike for STL containers both access variants (`at()` or `[]`) will throw an exception if a particular child could not be found or the index passed exceeds the total number of children of the group. In addition to this simple access interface `nxgroup` also exposes a fully STL compliant iterator interface. However, in order to use it some more deeper knowledge about *libpniio* is required and thus this topic will be dealt with in Section 8.3.

7.2.3. Other group related member functions

Like files, groups possess an `is_valid()` method which allows checking the state of a group. Similar to files, default constructed instances of `nxgroup` are not valid.

```
h5:nxgroup entry;

if(!entry.is_valid()) std::cerr<<"The entry group is not valid!"<<std::endl;
```

The getter methods `name()` and `filename()` return the name of the group and the name of the file the group is stored in respectively. Finally the `parent()` function returns the parent group of the a group. In order to use the `parent()` member function a bit more extra care is used. When using the method in a simple way like

```
h5:nxgroup p = other_group.parent();

everything will be fine. However, when we want to use the return value of parent() as a
temporary we have to do an explicit conversion to nxgroup like this

std::cout<<h5:nxgroup(entry_group.parent())<<std::endl;
```

The reason for this is that `parent()` does not really return an instance of `nxgroup` but rather of `nxobject`. But `nxobject` can be converted to `nxgroup` safely. The reason for this behavior will be explained in detail in Section 8.1.

7.3. Working with fields

Fields are the basic data holding facilities in `libpniio` and are represented by instances of `nxfield`. One can imagine a field as a multidimensional array stored on disk. Thus, it has quite similar properties than instances of `mdarray` in `libpnicore`. It is impossible to create a purely scalar field with `libpniio` as every field should be extensible if required.

7.3.1. Creating fields

Fields are created as children of a particular group instance. Creating fields is a rather complex task as there are many options available so let's start with the simplest possible example

```
h5::nxgroup entry = root["entry"];
h5::nxfield field = entry.create_field<float32>("temperature");
```

This creates a 1D field with a single element. This is as close as one can get to store a scalar value. The template parameter of the `create_field` method can be any type supported by `libpnicore`. For multidimensional fields use

```
h5::nxgroup entry = root["entry"];
h5::nxfield field = entry.create_field<float32>("temperature", shape_t{3,4});
```

which will create a 2-dimensional field with a shape of (3,4) and a total size of 12 elements. When using HDF5 as a storage format a compression algorithm can be associated with a field. This algorithm will later on be used to compress the data stored in a field and thus reduce disk utilization of the file. Currently only the standard deflate filter is supported

```
h5::nxgroup entry = root["entry"];
h5::nxdeflate_filter deflate(4, false);
h5::nxfield field = entry.create_field<float32>("temperature", shape_t{3,4}, deflate);
```

In this particular case the filter uses a compression level of 4 and no Fletcher pre-sorting of the data.

7.3.2. Reading and writing data

Fields provide two basic methods for reading and writing data: `read()` and `write()`. Both member functions accept a single argument which can be an instance of the following types

type	description
<code>mdarray<...></code>	an instance of the <code>mdarray</code> template
<code>array</code>	an instance of the array type erasure
<code>T&</code>	a single scalar value of the field's element type or a convertible type

In addition there is a special version of `read()` and `write()` available for legacy code with raw pointers. The two functions have the signatures

```
template<typename T> void read(size_t n, T *ptr);
template<typename T> void write(size_t n, const T *ptr);
```

7. Basic usage of `libpniio`

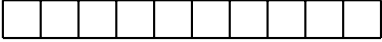
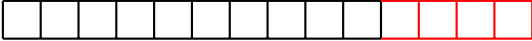
```
h5::nxfield f = ...;          before growth 
//..... code omitted .....
f.grow(0,4);                   after growth 
```

Figure 7.1.: Growing a one dimensional field by 4 elements.

The additional first argument `n` is the number of elements of type `T` referenced by the pointer `*ptr`. This number ensures that the functions can check if the size of the field matches the number of elements which should be read from or written to memory. A scalar can be read from a field simply with

```
float32 temperature;
h5::nxfield field = ...;
field.read(temperature);
```

and writing runs exactly as one would expect

```
float32 temperatur = ...;
h5::nxfield field = ...;
field.write(temperatur);
```

The same simple concept applies to all other types. For an instance of `mdarray` the code would look like this

```
auto data = dynamic_array<uint32>::create(shape_t{1024,1024});
h5::nxfield background = ....;

background.write(data); //writing

background.read(data);  //reading
```

The `read()` and `write()` member functions perform a size check on their arguments. The size of the argument must match the size of the field. In the case of scalar data a field-size of 1 is assumed. If argument and field size do not match a `size_mismatch_exception` is thrown.

7.3.3. Growing fields

The reason why there are no purely scalar fields is that during an experiment one would append data to a field as the measurement progresses. For this purpose `nxfield` provides a `grow()` method which allows to extend the field along a particular dimension. The member function has the signature

```
void grow(size_t e, size_t n=1)
```

where the first (mandatory) argument is the index of the dimension along which the field should grow and the second (optional) argument contains the number of elements by which to grow. Figures 7.1 and 7.2 show examples of growing a one and a two dimensional field respectively.

The canonical application for this feature would be to add content to a field as an experiment progresses. Using such a pattern we can start with a field of 0 size and then add points if required. The principal code would look like this

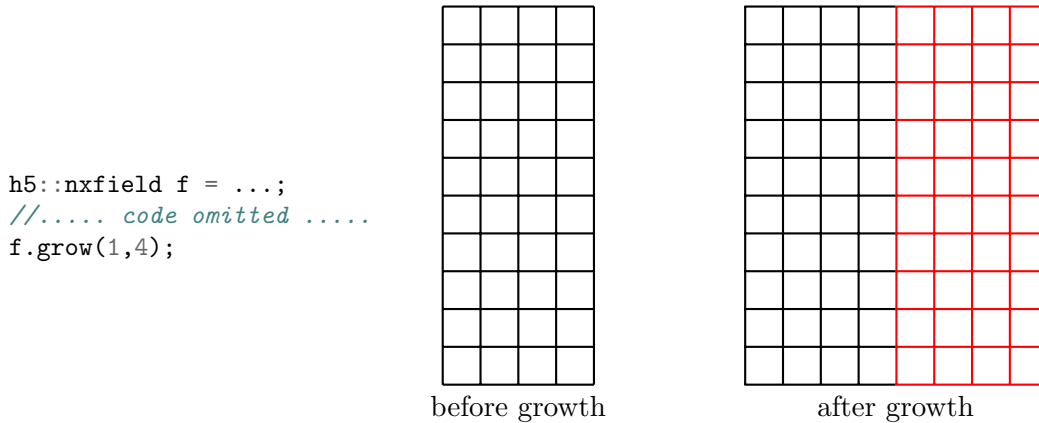


Figure 7.2.: Growing a two dimensional field by 4 elements along its second dimension.

```

h5:nxfile f = ...;
h5:nxfield data = detector_group.create_field<uint16>("spectra",
                                                    shape_t{0,1024});

size_t index = 0;
while(true)
{
    data.grow(0,1); //grow by one element along first dimension
    //..... code omitted ....
}

```

Note here that the initial shape of the field is (0, 1024). Such a pattern removes the burden of determining the number of points, recorded during an experiment, before the experiment starts. If the measurement is stopped somewhere in between the number of points written to the file matches exactly the number of points recorded. It is important to note that one cannot extend the size of a field only along *existing* dimensions. It is not possible to change the rank of a field via the growth method. Every attempt to do so will raise an `index_error` exception. How to write the data will be explained in the next section.

7.3.4. Partial reading and writing

The previous section explained how to adjust the size of fields dynamically. What is still missing is how to write data to such a growing field. The `read()` and `write()` member functions used so far are always writing the entire content of the field. This is not really what we want. Thus, `libpniios nxfield` type provides partial IO quite similar to the `mdarray` template of `libpnicore`.

The best way to understand partial IO is to have a look on an example. We thus will complete the previous one

```

h5:nxfile f = ...;
h5:nxfield data = detector_group.create_field<uint16>("spectra",
                                                    shape_t{0,1024});
auto buffer = dynamic_array<uint16>::create_array(shape_t{1024});

size_t index = 0;

```

7. Basic usage of `libpniio`

```
while(true)
{
    data.grow(0,1); //grow by one element along first dimension
    //..... code omitted ....

    //write data
    data(index++,slice(0,1024)).write(buffer);
}
```

The important line of code here is the last one in the `for`-loop. To obtain a selection of a field we can use the `()` operator of `nxfield`. The selection works the same as for `mdarray`. However, the return value is a new field instance with the selection set. One currently cannot apply more selections successively.

7.3.5. Field inquiry

Fields share a set of inquiry functions with groups. These are `name()`, `filename()`, and `is_valid()`. In addition to these functions there are also some member functions which are special for fields. The `size()` member function returns the total number of elements stored in a field. If a selection has been applied to the field `size()` returns the total number of elements selected. `type_id()` returns the Id of the field elements data type. The `shape()` template function returns a container, which can be passed as a template parameter, with the number of elements along each dimension.

```
h5:nxfield field = ...;
auto s = field.shape<shape_t>();
```

7.4. Working with attributes

Attributes are quite similar to fields. They can be attached to either a group or a field to provide additional information (metadata) for a particular field or group. Unlike fields attributes cannot

- use compression
- grow.

HDF5 attributes also do not support partial IO. However, `libpniio` provides partial IO for attributes in its implementation. Attributes can be accessed from their parent object (field or group instance) via the public attribute `attributes`. `attributes` is an instance of the `nxattribute_manager` template class. The details about `nxattribute_manager` is only of interest for developers working on `libpniio` and hence will not be discussed here. Only the interface `nxattribute_manager` exposes is of interest and will be discussed in this section.

7.4.1. Creating attributes

Attributes are created via the `create()` template member functions provided by `nxattribute_manager`. These functions are quite similar to those used to create fields below groups.

```
h5:nxfield f = ....;
auto units = f.attributes.create<string>("units");
```

This short snippet creates an instance of `nxattribute` the template type used to represent attributes in memory. The newly created attribute is scalar and can store a single string. Multidimensional attributes can be constructed by adding a container with shape information to the argument list of the `create` template.

```
h5::nxfield f = ...;
auto matrix = f.attributes.create<float32>("transformation", shape_t{3,3});

std::cout<<matrix.rank()<<std::endl; //output is 2
```

It is important here to note that even a scalar attribute has a rank of 1. This might not be the obvious choice but makes fields and attributes more consistent. To check if an attribute is a scalar one could use the `size()` member function of an attribute. If the `size()` returns 1 the attribute can be considered as scalar.

When one tries to create an attribute on an object which already has an attribute of the same name an `object_error` exception will be thrown.

```
h5::nxfield f = ....;
f.attributes.create<string>("units"); //create the original attribute
f.attributes.create<float32>("units"); //would throw object_error
```

However, an attribute can be overwritten with

```
h5::nxfield f = ....;

//create the original attribute
f.attributes.create<string>("units");

//overwrite original attribute
f.attributes.create<float32>("units", true);
```

where the last argument of the second call to `create()` allows to overwrite an existing attribute. A similar call exists for multidimensional attributes

```
h5::nxfield f = ....;

//create the original attribute
f.attributes.create<string>("units", shape_t{3});

//replace the original "units" attribute
f.attributes.create<float32>("units", shape_t{3,4}, true);
```

7.4.2. Attribute inquiry

Attributes and fields share the same set of inquiry methods. Thus, see Section 7.3.5 for details.

7.4.3. Accessing an objects attributes

The attribute manager instance associated with each field or group exposes a very minimalistic but STL compliant container interface. Its `size()` returns the number of attributes attached to an object. One can access each attribute either by its index

7. Basic usage of libpniio

```
h5::nxgroup g = ....;

for(size_t i=0;i<g.attributes.size();++i)
    std::cout<<g.attributes[i].name()<<std::endl;
```

or by its name

```
h5::nxgroup g = ....;
auto attr = g.attributes["NX_class"];
```

When using the `[]` operator with a numeric index an `index_error` exception will be thrown if the index exceeds the total number of attributes. Similarly, `[]`, when used with an attributes name, will throw a `key_error` exception. One can also iterate over all attributes, either by using the standard `begin()` and `end()` functions to retrieve iterators, or by using the more modern for-each construction

```
h5::nxfield f = ....;
for(auto attr: f.attributes)
    std::cout<<attr.name()<<std::endl;
```

7.4.4. Reading and writing data from and to attributes

Data IO for attributes works exactly the same as for fields with the exception that attributes cannot be changed in size. A simple example for writing and reading a string attribute would look like this

```
string unit = "nm";
h5::nxfield f = ....;
auto attr = f.attributes.create<string>("units");
attr.write(unit);
//..... code omitted .....
attr.read(unit);
```

The read/write member functions also accept instances of the `mdarray` template as well as of `array`. Like for fields a pointer version exists to interact safely with legacy C libraries

```
size_t size = 9;
double *matrix = get_matrix_from_c_code();

h5::nxfield f = ....;
auto attr = f.attributes.create<float64>("matrix",shape_t{3,3});
attr.write(n,matrix);
```

Reading works pretty much the same, however, you have to allocate memory before reading the data from the attribute

```
h5::nxfield f = ....;
auto attr = f.attributes["matrix"]; //obtain the attribute from the field
float64 *matrix = allocate_memory(attr.size());

attr.write(attr.size(),matrix);
```

Unlike standard HDF5 attributes support libpniio's NeXus-attributes partial IO. Partial IO with attributes works exactly the same way as with field

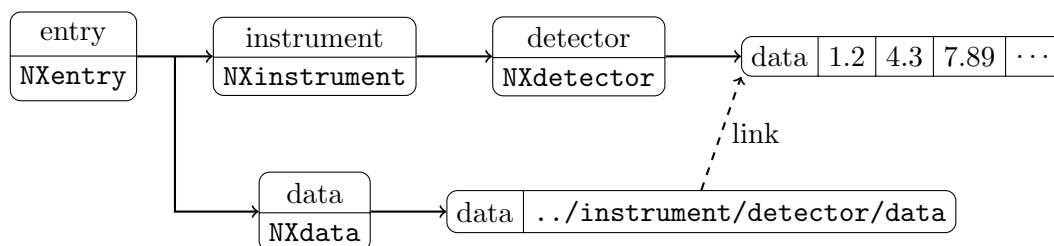


Figure 7.3.: Links can be used to make data available in two different group without duplicating the data. Here the data stored in the detector is also available below the data group in the entry group of the tree.

```
h5:nxfield f = ...;
auto attr = f.attributes.create<float64>("matrix",shape_t{3,3});
auto row = dynamic_array<T>::create(shape_t{3});
//..... code omitted .....
for(size_t i=0;i<3;++i)
{
    row = .....; //fill row buffer with data
    attr(i,slice(0,3)).write(row);
}
```

For reading data just replace the `write` with `read`. It should be mentioned that using partial IO on attributes, though it works, can be significantly slower than writing the entire attribute in a single step.

7.4.5. Attribute management

There are two functions left of the `nxattribute_manager` interface. One is the `exists()` member function which can be used to check for the existence of a particular attribute.

```
if(!field.attributes.exists("units"))
    std::cerr<<"Field does not have a units attribute!"<<std::endl;
```

An attribute can be removed from an object using the `remove()` method.

```
if(field.attributes.exists("units"))
    field.attributes.remove("units");
```

The `remove` method throws `key_error` if the attribute to delete does not exist.

7.5. Working with links

One of the key features of NeXus is its support for links. Like on a file system linking allows for making data available at different positions in the NeXus group hierarchy without data duplication. A typical application for a link would be the data stored in `NXdata` instance of a NeXus-file. NeXus distinguishes two kinds of links

- internal links - where the link and its target are located in the same file
- and external links - where the target resides in a different file than the link.

7. Basic usage of *libpniio*

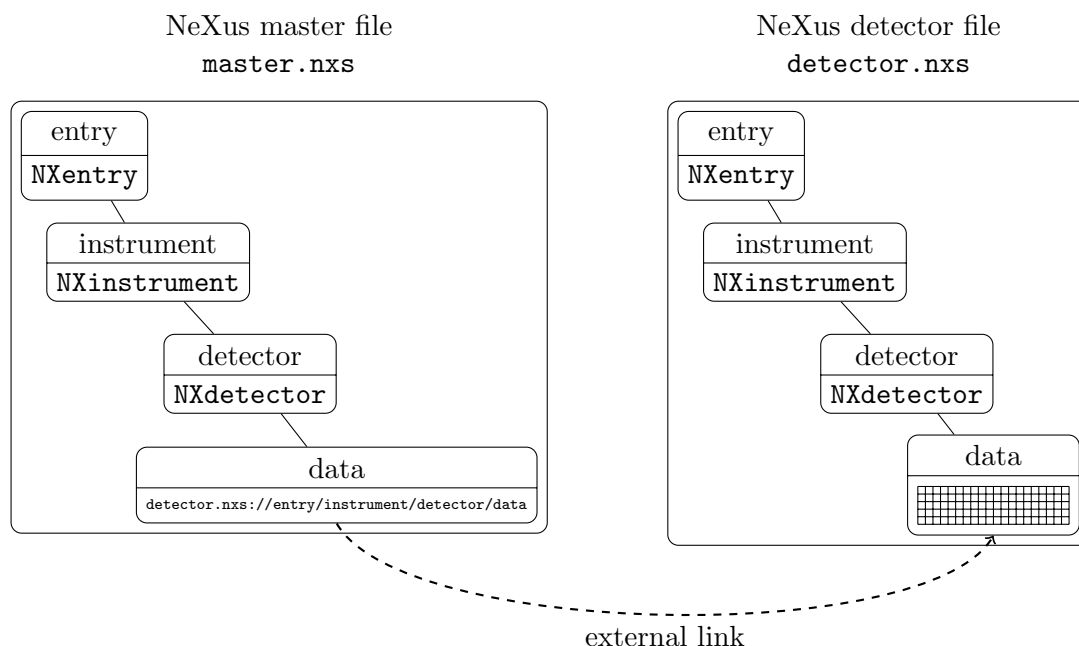


Figure 7.4.: An external link used to reference the data stored in a separate detector file via the data field in the master file. The master file is what the user typically uses to access the data.

For both kinds of links there are canonical use cases. The standard use case for internal links is depicted in Fig. 7.3. Here the link is used to reference the data stored in the detector group of the file in its data group¹. By using links we only have to store the data once (in the detector group) and can make it available at any other position in the file.

The default use case for external links is shown in Fig. 7.4. In many cases detector write to a network storage while entirely bypassing the control system of the experiment. This is typically done when the detector has to record images at a very high rate which could not have been achieved if all the data has to be pumped through the control systems software stack. In the best case, as shown in Fig. 7.4, the detector already writes its data in a NeXus file which contains nothing else than the detector images. While the detector is recording images the control system writes all the metadata and other relevant data to a master file (this data is usually rather small and not recorded at low frequency). As a result we end up with two files and the user would have to access metadata through the master file and the detector images via the detector file. By using external links it is possible to make data within a NeXus-file available that is originally stored in a different NeXus-file. Now the user can access the detector data via the master file as it would be stored directly within this file. The only thing the user has to remember is that he or she needs to copy two files when moving the data to a different storage.

7.5.1. Create internal links

To create an internal link the `pni::io::nx` namespace provides the `link()` function template which exists in several overloaded versions. All of them differ virtually only in the way how

¹This group is used for programs to quickly find plottable data.

the link target is referenced which is the first argument of `link()`. The second and third argument are the parent group for the new link and its name.

The most intuitive version of the `link()` function template references the target object directly via their group or field instance. The next example shows the canonical case where a new link to the detector data is generated below the data group.

```
h5::nxfile f = h5::nxfile::open_file(data,true);
h5::nxgroup entry = f.root()["entry"];
h5::nxgroup instrument = entry["instrument"];
h5::nxgroup detector = instrument["detector"];
h5::nxgroup datagroup = entry["data"];
h5::nxfield det_data = detector["data"];
```

```
link(det_data,data_group,"data");
```

With this approach only internal links can be created as the target and the parent object for the new link must be located in the same file. This approach is rather tedious as we have to open all the groups (for now we have no better way - see the advanced section). However, one can exploit a feature that NeXus-links have in common with file system links: the target object must be available when the link is created. Consequently we can describe the target also merely by its path

```
h5::nxgroup data_group = ...;
```

```
link("../instrument/detector/data",data_group,"data");
```

Here the first argument (the target) is described by a string with the path to the target. In this case the path is relative to the parent group of the link. Alternatively one can also use an instance of `nxpath` to reference the target object

```
h5::nxgroup data_group = ...;
nxpath target_path = nxpath::from_string("../instrument/detector/data");
```

```
link(target_path,data_group,"name");
```

There is one important restriction a path has to obey when it should be used as a reference to a link target: the path must not contain type only elements. For instance the path `"../:NXinstrument/:NXdetector/data"` would be a perfect valid NeXus-path. However, when using such a path `link` will throw a `value_error` exception. The reason for this is simple: `libpnio` currently uses the HDF5 linking mechanism which has no idea about NeXus-group types! Thus, as a rule of thumb, use only names within the path used to reference a link target.

There is some other subtle issue with paths to link targets. They must not comprise intermediate parent directory references (`..`). This cannot be handled by HDF5 correctly. `..` is only allowed at the beginning of a relative path. Absolute paths must not contain any `..` elements. Thus `../../entry/instrument` would be a valid target path for a link, but `../entry/../../entry` would not.

7.5.2. Create external links

To create external links within a NeXus-file use the `link` function as for internal links. However, only the string and path version of `link` can be used to create external links. The only

7. Basic usage of `libpniio`

thing one has to do in order to obtain a link add the filename to the path. Lets start again with the canonic example, the external detector file

```
h5::nxgroup detector_group = ....; //open the detector group in the master file  
link("detector.nxs://entry/instrument/detector/data",detector_group,"data");
```

7.5.3. Link inquiry

`libpniio` (or better HDF5) distinguishes three kinds of links

hard links	these are links to an object which are typically created from the parent object to a newly created field or group
soft links	this is what one typically creates with the <code>link</code> function template
external links	are links that reference an object in a different file.

The major difference between hard- and soft-links is the fact that the latter ones can dangle (the object they point to must not exist). `libpniio` provides utility functions to check what kind of link one deals with.

```
h5::nxgroup parent = ...;  
string name = "data";  
  
if(is_hard_link(parent,name))  
    std::cout<<"hard link";  
else if(is_soft_link(parent,name))  
    std::cout<<"soft link";  
else if(is_external_link(parent,name))  
    std::cout<<"external link";  
else  
    std::cout<<"unkown link type";
```

Each of the function takes the parent object of the link as its first and the name of the link as its second argument.

8. libpniio in more detail

8.1. The mysterious `nxobject`

In section 7 we have discussed the basic usage of `libpniio` and the careful reader may already have observed some missing concepts

- what is the mysterious return type of the `parent()` member function every field, group, or attribute possesses?
- if `nxgroup` is like a container – can we use iteration with it?

Indeed solving the first riddle leads to a good solution for the iteration problem as both are coupled by the fact that `nxgroup` is not just a simple container type as the STL containers, it is rather a heterogeneous container.

But let's start at the beginning – the return type of `parent()`. If we would have only fields and groups the situation would be easy: the return type of the `parent()` function for a field or group would always be `nxgroup` (as fields cannot have children for their own). However, what about attributes. They can be attached to fields or groups. Thus, also the return type would be either a field or a group. Unfortunately, a function (and thus a method too) can only have one return type. We would need a return type that could represent either a field or a group. The classical approach to solve such an issue in object oriented programming would be a class hierarchy as depicted in Fig. 8.1.

However, this approach has some flaws. What interface should be base class expose. In fact, the parts of their interface `nxfield` and `nxgroup` have in common also matches those of `nxattribute` too. However, all three classes are from a semantic point of view to different to be derived from a single base class. There is also a less philosophical problem. `nxfield` and `nxgroup` are in fact templates whose template parameter is the implementation type (the same is true for `nxattribute`). The basic idea was to implement a *pointer to implementation* pattern with template (so in fact no pointers) following an approach presented by [1]. This approach has severe consequences for a base class approach: as the base class would need

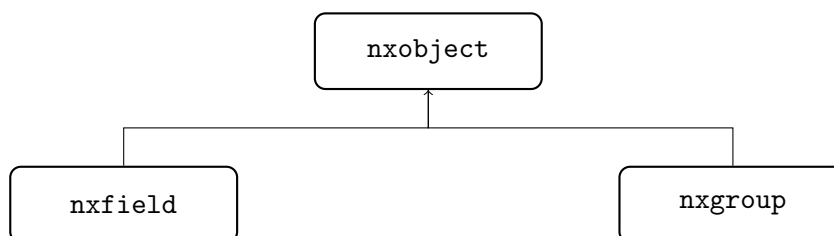


Figure 8.1.: The default procedure in OOP to obtain a master object which can either represent a field or a group would be inheritance. In this schema `nxfield` and `nxgroup` are both descendants of `nxobject`.

8. *libpniio* in more detail

access to the particular implementation and thus the implementation type, a class hierarchy must be assembled for each implementation type. During the development of *libpniio* such an approach was once used and has proven to be far too complex.

To circumvent all these issues and in general avoid design by inheritance (as suggested by [2]) a different approach was taken. You may have already recognized in the API documentation an instance of type `nxobject`. `nxobject` mimics a bit the behavior of a base class but in reality it is something entirely different. It is an instance of a *variant* template provided by the `boost::variant` library. `nxobject` thus can hold either an instance of `nxgroup`, `nxfield`, or `nxattribute`. It is thus the perfect return type for the `parent()` member function of `field`, `groups`, and `attributes`. Indeed `nxobject` is also what the `[]` operators and the `at()` member function of `nxgroup` returns. By the special design of the constructor as well as the assignment operator of `nxfield`, `nxgroup`, and `nxattribute`, these objects can be directly constructed from an instance of `nxobject` if the instance of `nxobject` holds an instance of the appropriate type. Thus, `nxobject` is rather transparent to the user. In the next sections 8.2 and 8.3 we will see how to work with instances of `nxobject` effectively.

It is important to understand that `nxobject` depends on the particular implementation in use. Thus it is part of the particular implementation namespace. As *libpniio* currently only uses HDF5 the appropriate type would be `h5::nxobject`.

8.2. Using algorithms

`nxobject` does not expose any interface functions. Thus, all operations on instances of `nxobject` are implemented as algorithms using the visitor pattern. Algorithms can be used by adding

```
#include <pni/io/nx/algorithms.hpp>
```

at the top of a source file. All the function templates provided reside within the `pni::io::nx` namespace. Some of the algorithms also work with the standard types like `nxfield`, `nxgroup`, and `nxattribute`. Which one will be explained in the following section.

8.2.1. Basic inquiry and conversion

When accessing the elements of a group or when retrieving the parent object of an attribute the first question one may raise is: what kind of object is stored in the `nxobject` instance. There are basically three simple functions: `is_attribute()`, `is_field()`, and `is_group()`. Each of these functions takes as their single argument an instance of `nxobject`.

```
h5::nxgroup g = file.root();

for(size_t i=0; i<g.size(); ++i)
{
    auto object = g[i]; //this returns an instance of nxobject

    if(is_field(object))
        std::cout<<"found a field"<<std::endl;
    else if(is_group(object))
        std::cout<<"found a group"<<std::endl;
    else
```

```

        std::cout<<"found unknown object type!"<<std::endl;
    }

```

The first line in this example, `h5::nxgroup g = file.root();` is a good example for implicit conversion. In fact `root()` returns an instance of `nxobject` rather than of `nxgroup`. However, `nxgroup` can be constructed from `nxobject` as long as `nxobject` stores an instance of `nxgroup`.

There are also three conversion function templates available: `as_field()`, `as_group()`, and `as_attribute()` each taking an instance of `nxobject` and returning a field, group, or attribute instance. If the `nxobject` argument does not store a value of appropriate type a `type_error` exception will be thrown. We could modify the `if` block in the above example with

```

if(is_field(object))
    do_something_with_field(as_field(object));
else if(is_group(object))
    do_something_else_with_groups(as_group(object));
else
    std::cout<<"found unknown object type!"<<std::endl;

```

None of these algorithms will work on either `nxfield`, `nxgroup`, or `nxattribute`. It would not make too much sense to ask an instance of `nxfield` whether or not it is a field. We know this already.

8.2.2. Common algorithms for fields, groups, and attributes

Fields, groups, and attributes share a set of common member functions which can be accessed by a set of algorithms acting on an instance of `nxobject` which stores either a field, a group, or an attribute.

is_valid()

One of the most important questions to ask from an object is whether or not it is valid. The `is_valid()` algorithms does exactly that. It behaves like the `is_valid()` member function of `nxgroup`, `nxfield`, or `nxattribute`. Indeed, this algorithms does not only work for `nxobject`, `nxgroup`, `nxfield`, and `nxattribute` but also for `nxfile`. Using it is fairly simple

```

auto object = .....;

if(!is_valid(object))
    std::cerr<<"Object not valid!"<<std::endl;

```

The return value is of type `bool` and has the value `true` when the object is valid, `false` otherwise.

close()

This template function works for `nxgroup`, `nxfield`, `nxattribute`, and `nxobject`. It does what the name suggests: it calls the `close()` method of the object passed as its only argument.

```

auto object = ...; //can be either a field,group, attribute, or object

close(object);

```

8. libpniio in more detail

This function template has no return value.

get_filename()

Calls the `get_filename()` method on the object passed as an argument. The function template works with `nxobject`, `nxfield`, `nxgroup`, and `nxattribute`.

```
auto object = ...; //can be either a field,group, attribute, or object
```

```
std::cout<<"File is: "<<get_filename(object)<<std::endl;
```

The return value is a string with the name of the file.

get_root()

Returns the root group of the file an object belongs to. I accepts `nxobject`, `nxattribute`, `nxfield`, and `nxgroup` instances as arguments.

```
auto object = ...; //can be either a field,group, attribute, or object
```

```
h5::nxgroup root = get_root(object);
```

The root group is returned as an instance of `nxobject`. As shown in the short code snippet above we do not have to care about this as `nxgroup` can be constructed from an `nxobject`.

get_name()

Returns the name of an object. The function templates takes an instance of `nxobject`, `nxfield`, `nxattribute`, `nxgroup` as its only argument

```
auto object = ...; //can be either a field,group, attribute, or object
```

```
std::cout<<"Name: "<<get_name(object)<<std::endl;
```

The name is returned as a string. If the object is not valid an `invalid_object_error` exception is thrown.

get_parent()

Return the parent of an object. Takes an instance of `nxobject`, `nxfield`, `nxattribute`, and `nxgroup` as its only argument

```
auto object = ...; //can be either a field,group, attribute, or object  
auto p = get_parent(object);
```

The parent object is returned as an instance of `nxobject`. If the object is not valid an `invalid_object_error` is thrown.

get_path()

The `get_path` function template takes either an `nxobject`, `nxattribute`, `nxfield`, and `nxgroup` as its single argument.

```
auto object = ...; //can be either a field, group, attribute, or object
std::cout<<get_path(object)<<std::endl;
```

The path is returned as a string. This is maybe not seem to be the best approach but strings are most probably more often printed than used for iteration. However is a path object is required one can always do

```
nxpath p = nxpath::from_string(get_path(object));
```

get_size()

Returns the size of an object. This function template accepts an instance of `nxobject`, `nxattribute`, `nxfield`, or `nxgroup` as its single input argument.

```
auto object = ...; //can be either a field, group, attribute, or object
size_t n = get_size(object);
```

The interpretation is slightly different whether or not the argument represents a group, a field, or an attribute.

- for fields and attributes the return value is the total number of elements stored in the field or attribute
- for groups it is the total number of children.

8.2.3. Group related algorithms

The set of algorithms described in this section make only sense in connection with a group object.

create_field()

The `create_field` function template works only with an instance of `nxobject` which stores an instance of `nxgroup`. In the simplest case the `create_field`

```
auto detector = instrument["detector"];

auto field = create_field<float64>(detector, "../temperature");
```

The template parameter of the `create_field` function template, in this example, determines the data type which should be used for the field. The first argument is the parent object for the field. In this example it is the group *detector* represented by an instance of `nxobject`. The second argument is the path to the new field. In this case this particular case the new field will not be created no below the *detector* but below the *instrument*. In fact, the parent object (the first argument) is just the starting point for the path to the new field. The path must exist except for the last element (which is the name of the newly created field). The path can either be passed to the function template as a string or as an instance of `nxpath`.

Multidimensional fields can be created by appending the shape of the field to the list of arguments of the `create_field` function template

8. *libpniio* in more detail

```
auto detector = instrument["detector"];  
  
auto field = create_field<float64>(detector, "background", shape_t{1024,1024});
```

A filter can simply be added by appending it to the argument list

```
h5::nxdeflate_filter comp = ...;  
auto detector = instrument["detector"];  
shape_t shape{1024,1024};  
  
auto field = create_field<float64>(detector, "background", shape, comp);
```

A description of custom chunking can be found in section 8.5. One of the disadvantages of these version of the `create_field` function template is the fact, that the data type can only be determined at compile time. To overcome this problem there is also a runtime version

```
h5::nxdeflate_filter comp = ...;  
auto detector = instrument["detector"];  
shape_t shape{1024,1024};  
  
auto field = create_field(detector, type_id::FLOAT64, "background", shape, comp);
```

where the second argument is the type ID of the desired data type. All other arguments are the same as for the previous versions of `create_field`.

create_group()

Creating groups using algorithms is much more powerful than the standard `create_group` method of `nxgroup`. The `create_group` algorithm requires two arguments where the first one is the parent object for the new group and the second one the path to the new group. The `creat_group` algorithm uses the NeXus-path syntax to determine the type of the newly created group. In it simplest form, in order to achieve the same result as with the `create_group` member function, one could use

```
auto root = ...;  
auto entry = create_group(root, "entry:NXentry");
```

which would create a new group of name `entry` and of type `NXentry`. A more elaborate example would be

```
auto detector = ...  
auto mono      = create_group(detector, "../monochromator:NXmonochromator");
```

Instead of its string representation, an instance of `nxpath` can be used to determined position, name, and class of the new group.

get_child()

Retrieves the child of a group object. The child can be identified either by its index or by its name (and class in case of a group).

```

auto entry = ....;

for(size_t i=0;i<get_size(entry);++i)
{
    auto child = get_child(entry,i);

    //do something with the child
}

```

The parent can either be an instance of `nxgroup` or `nxobject`. A `type_error` exception is thrown in the case that an `nxobject`-parent does not store an instance of `nxgroup`. More useful, however, is to search for a particular object by its type

```

auto instrument = ...;
auto detector   = get_child(instrument,"","NXdetector");

```

or by its name

```

auto instrument = ...;
auto monitor    = get_child(instrument,"monitor");

```

or even both

```

auto instrument = ....;
auto detector_1 = get_child(instrument,"det_1","NXdetector");

```

In all, of the last three examples, if there are more than one instance in the parent which would satisfy the requirement, the first one will be returned.

get_children()

To be implemented

get_class()

The `get_class` function template works for instances of `nxgroup` and `nxobject`. It returns the value of the `NX_class` attribute which determines the base class type of the group.

```

if(get_class(group) == "NXentry")
    //do something with the entry
else
    throw type_error(EXCEPTION_RECORD,"group is not an entry!");

```

If the attribute is not set or does not exist the function returns an empty string.

get_object()

Aside from the `get_child` function template, this is most probably one of the most important function templates presented in this section. The common methods to access children of a group usually only allow you to access the direct children of a group. The `get_child` function template allows us to go beyond this limitation. We can use a path relative to a parent object

8. libpniio in more detail

```
auto detector = ....;
auto monitor  = get_object(detector, "../NXmonitor");
```

Or even an absolute path

```
auto detector = ....;
auto monitor  = get_object(detector, "/:NXentry/:NXmonitor");
```

Quite similar to `get_child`, if several objects exist that would match the path, the first one will be returned. The path to the object can be passed either as a string or as an instance of `nxpath`.

get_objects()

To be implemented!

is_class()

The `is_class` function template checks if a group is an instance of a particular base class.

```
if(is_class(group, "NXentry"))
    //process NXentry
else
    throw type_error(EXCEPTION_RECORD, "group is not an instance of NXentry!");
```

This function works for instances of `nxgroup` as well as for `nxobject`. If `nxobject` does not hold an instance of `nxgroup` a `type_error` exception will be thrown.

set_class()

Set the base class type for a particular group.

```
set_class(group, "NXentry");
```

The first argument is the group for which to set the type. It must be either an instance of `nxgroup` or `nxobject`. In the latter case a `type_error` exception is thrown if the `nxobject` does not store a group instance.

If the group has already its `NX_class` attribute set, the old value will be overwritten.

8.2.4. Field and attribute related algorithms

Fields and attributes are rather similar from a developers point of view. Most algorithms working on attributes also work on field and vice versa.

get_unit() – fields only

This function template accepts either an `nxobject`-instance storing a field or an instance of `nxfield`. The `get_unit` function template reads the `units` attribute from a field and returns its value (as a string).

```
auto o = get_object(root, ":NXentry/:NXdata/:NXdetector/data");
string detector_unit = get_unit(o);
```

Alternatively one can use the function template with `nxfield`

```
h5:nxfield field = get_object(...);
string detector_unit = get_unit(field);
```

In principle it would be possible to retrieve an attribute also via the attribute manager interface. Thus, this function template is merely a convenience function. In the case of an `nxobject` argument a `type_error` exception is thrown. For arguments of type `nxattribute` and `nxgroup` a compile time assertion fails and breaks the build.

set_unit() – fields only

`set_unit()` is the counterpart of the `get_unit` function. It is used to set the value of the `units` attribute attached to a field. If the attribute does not exist it will be created, otherwise its content will be overwritten. Like `get_unit` this function template accepts either an instance of `nxfield` or an instance of `nxobject` storing a field instance.

```
//handle a field stored in nxobject
auto field = get_object(root, ":NXentry/:NXdata/:NXdetector/data");
set_unit(field, "cps");
```

or

```
h5:nxfield field = get_object(...);
set_unit(field, "cps");
```

If an `nxobject` argument does not store a field a `type_error` exception will be thrown. A failing compile time assertion will break the build in the case of an `nxgroup` or `nxattribute` argument.

create_attribute – attributes only

The `create_attribute` function template creates, as its name suggests, an attribute attached to a field or group. The parent object for the attribute must be passed as an instance of `nxobject`. If the `nxobject`-instance does not store a group or field a `type_error` exception will be thrown. To create a scalar attribute

```
auto field = get_object(root, ":NXentry/:NXinstrument/:NXdetector/data");
auto attr = create_attribute<string>(field, "depends_on");
```

As for the `create` member function of the attribute manager interface, every subsequent attempt to create an attribute on the same object with equal names will throw an exception. However, like for the attribute manager interface there is an overwrite flag

```
auto field = get_object(root, ":NXentry/:NXinstrument/:NXdetector/data");
auto attr = create_attribute<string>(field, "depends_on");
```

```
// this would throw an exception
// attr = create_attribute<string>(field, "depends_on");
```

```
// however - this will work
auto attr = create_attribute<string>(field, "depends_on", true);
```

8. libpniio in more detail

For multidimensional attributes the shape has to be passed after the name of the attribute.

```
auto detector = get_object(":NXentry/:NXinstrument/:NXdetector");
auto field = get_object(instrument,":NXtransformation/tth");
auto attr = create_attribute<float64>(field,"offset",shape_t{3});
```

Like for scalar attribute, in order to overwrite a multidimensional attribute, append the overwrite flag to the argument list. For both function templates runtime function exist where the datatype is passed as an instance of `type_id_t` allowing the pick the datatype at runtime

```
auto attr = create_attribute<string>(field,"depends_on",type_id_t::STRING);
```

get_attribute – attributes only

The `get_attribute` function template retrieves an attribute from either a field or a group parent. The parent can be passed as an instance of `nxobject`, `nxfield`, or `nxgroup`. The function currently only allows retrieval by name.

```
auto g = get_object(root,":/NXentry/:NXinstrument");
std::cout<<get_name(get_attribute(g,"NX_class"))<<std::endl;
```

If the requested attribute does not exist `key_error` is thrown.

get_rank()

This function template works for fields as well as for attributes and returns the number of dimensions of a field or attribute. In particular for *scalar* fields or attributes the results are slightly different. Unlike fields, attributes cannot alter their size. Consequently, the rank of an attribute storing a single value is 0. Fields can be resized and even if they have a size of 1 they can be extended along a single dimension. Thus, fields have at least a rank of 1. The function template accepts as its single argument, the object for which to determine the rank, as an instance of `nxfield` or `nxattribute`. The application of the template is fairly easy

```
if(get_rank(attr) == 0)
    std::cout<<"scalar attribute"<<std::endl;
```

or

```
if((get_rank(field) == 1) && (get_size(field)==1))
    std::cout<<"scalar field"<<std::endl;
```

get_shape()

Get shape returns a user defined container type with the number of elements along each dimension from a field or attribute. Similar to `get_rank` there is a subtle difference in the output for scalar fields and scalar attributes. For scalar attribute, having a rank of 1, the return value is an empty container. For scalar fields the container has a single element whose value is 1. Using `get_shape`, again, is rather simple

```
auto shape = get_shape<shape_t>(field);
```

The only argument of `get_shape` can either be an instance of `nxfield`, `nxattribute`, or `nxobject`.

get_type()

Returns the type ID of the field or attribute passed to this function template. The argument can be either an instance of `nxobject`, `nxfield`, or `nxattribute`. In the former case a field or attribute must be stored in the `nxobject`, otherwise `type_error` will be thrown.

```
auto object = get_object(parent,path);
type_id_t tid = get_type(object);

switch(tid)
{
    ....
};
```

grow()– fields only

The grow template function allows growing a field provides either as an instance of `nxfield` or `nxobject`. The usage is quite similar to that of the `grow` member function of `nxfield`. The following snippet grows a field along its first dimension by 100 elements

```
auto field = get_object(root,":NXentry/:NXinstrument/:NXdetector/data");
grow(field,0,100);
```

read() and write()

The `read` and `write` functions work for attributes¹ as well as for fields. The `read` and `write` functions take either an instance of `nxfield`, `nxattribute`, or `nxobject` as an argument. To write or read an entire field or attribute simply use

```
auto wbuffer = ....;
auto rbuffer = ....;

auto object = get_object(parent,...);
write(object,wbuffer);
read(object,rbuffer);
```

One can also do partial IO with the `read` and `write` functions (although the syntax maybe a bit strange)

```
auto wbuffer = ....;
auto rbuffer = ....;

auto object = get_object(parent,...);
write(object,wbuffer,0,slice(0,3),slice(0,5));
read(object,rbuffer,0,slice(0,3),slice(0,5));
```

For partial IO just append the selection (as you would do it with the native `()` operator) to the argument list of `read` or `write`.

¹As of version 1.0.0, support for partial IO has been added to attributes. Hence, from the point of reading and writing, attributes behave exactly the same as fields.

8.3. Iterating groups

8.3.1. Simple iteration

The `nxgroup` type provides an STL compliant iterator interface to iterate over the direct children of a group. In this very simple example we loop over all entries stored in a file

```
h5::nxgroup root = f.root();

for(auto entry: root)
    std::cout<<entry.name()<<std::endl;
```

Another interesting example would be to count all instances of `NXdetector` within an instrument group

```
#include <pni/core/types.hpp>
#include <pni/io/nx/nx.hpp>
#include <pni/io/nx/algorithms.hpp>

using namespace pni::core;
using namespace pni::io::nx;

//predicate function
bool is_detector(const h5::nxobject &o)
{
    if(is_group(o)) return is_class(o,"NXdetector");
    else return false;
}

int main(int argc, char **argv)
{
    h5::nxfile file = h5::nxfile::open_file("test.nxs");
    h5::nxgroup instrument = get_object(root, "/:NXentry/:NXinstrument");

    size_t ndetectors = std::count_if(instrument.begin(), instrument.end(),
                                      is_detector);
    std::cout<<"Found "<<ndetectors<<" detectors!"<<std::endl;
    return 0;
}
```

8.3.2. Recursive iteration

As we have seen in the previous section: iterating over the children of a group is rather simple and works like with any other STL container. However, in many cases iteration should run recursively over all objects of a group and its subgroups. For this purpose `libpniio` provides the `flat_group` template. The *flat* means that the entire structure below a group is *flattened* into a single linear container. The `flat_group` provides a fully STL compliant container interface and thus, just like `nxgroup`, can be used along with all the algorithms provided by the STL.

In the next example we want to print the path of all objects stored within an entry to standard output

```

auto flat = make_flat(get_object(root, "NXentry"));

for(auto object: flat)
    std::cout<<get_path(object)<<std::endl;

```

Note, that `flat_group` is constructed using the `make_flat` utility function. The objects `flat_group` returns are of type `nxobject`. Thus, we have to use the algorithms to work with them or convert them to their native types.

8.4. Deleting items

Sometimes one may want to remove an object from a file. Though it is possible to remove objects from a file one should use this operation with care. The reason for this is due to a limitation of the HDF5 backend currently used for storing data to disk. HDF5 cannot remove a data item physically from disk. It only removes all links to it so that the object cannot be accessed any more. This is the price one has to pay for random access. Physically removing the data from disk would leave a hole in the HDF5 file. If one has to delete a large object, like a field with detector data, one should run the `repack` command afterwards on the file which will delete all the deleted objects by rewriting the file to disk.

In order to remove an object use the `remove` member function of `nxgroup`. Let us suppose we want to delete the detector group from a NeXus-file. This could be done as follows

```

h5:nxgroup beamline = get_object(root, "NXentry/NXinstrument");
beamline.remove("detector");

```

The attribute manager provides a `remove` member function too. This can be used to remove an attribute from a field or group.

```

h5:nxfield field = get_object(root, ...);
field.attributes.remove("temp_attribute");

```

8.5. Custom field chunks

Chunking is an important issue but requires some more detailed information about how the data is stored. That's why this section concludes this advanced usage chapter. Chunking is particularly important for multidimensional data. The chunking feature is unique to the HDF5 library and thus maybe not available in future other implementations.

8.5.1. What are chunks?

Without chunking a field is always accessed as a whole. This can be a severe issue if the field is very large (maybe even larger than the available main memory). Figure 8.2 shows such a situation. Consequently, whenever data is accessed the entire field has to be read or written from and to the disk. In order to avoid this situation data can be subdivided into smaller *chunks*. Assuming a multidimensional field, chunks can be considered as slices (selections) within this field. A single chunk is contiguously written to disk. However, the next chunk might be written on an entirely different location within the file rather than in order to avoid this situation data is written in smaller portions: the so called *chunks*. A chunk can be considered a slice of the multidimensional field which should be written. This slice represents

8. *libpniio* in more detail

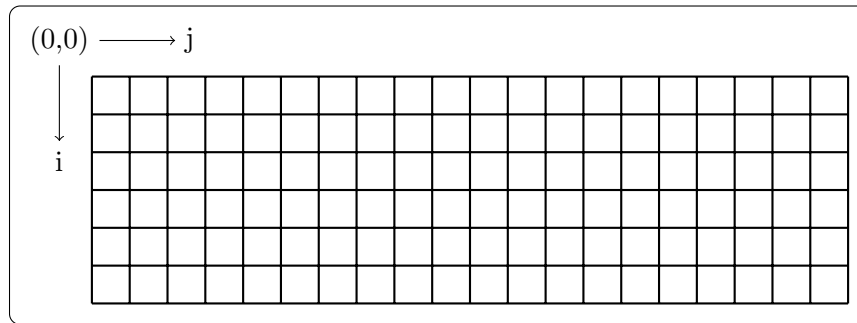


Figure 8.2.: Nexus field without chunks: i denotes the slow and j the fast dimension. The entire field has to be written as once.

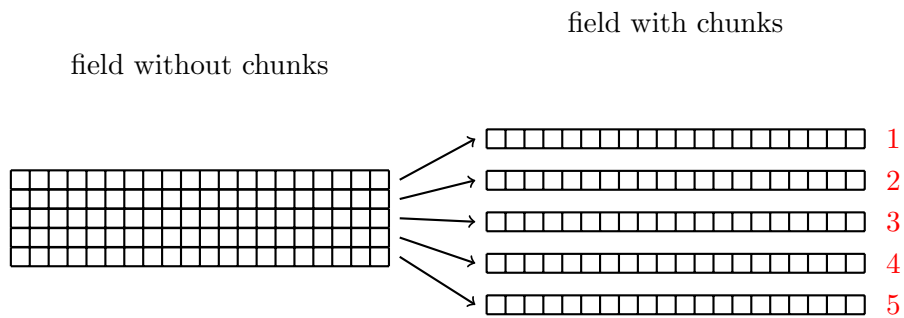


Figure 8.3.: The field from Fig. 8.2 is distributed over several chunks. In this case the chunks are chosen along the fast dimension of the field which is usually a good choice (assuming C-ordering of multidimensional data). The red numbers indicate the chunk index. The individual chunks do not have to be stored consecutively in memory.

the unit of data which is contiguously written to disk. Figure 8.3 shows an example how a NeXus field is distributed over several chunks. In this example the chunks have been chosen along the fast dimension. This is typically a good choice if multidimensional data is stored in C-order. Chunking is entirely transparent to the user. The way how fields are distributed in chunks does not influence the read and write functions. However, it can have significant influence on the performance. This is particularly true when using compression. Without chunks the entire field is compressed. This can cause dramatic performance penalties when accessing data as the entire data must be read from disk and afterwards been inflated. When using chunks the compression is only applied to a single chunk. Though this may slightly reduced the compression ration which can be achieved, it can given an extreme performance boost when reading data. Only the chunk where the requested data resides in must be read from disk and being inflated.

8.5.2. Setting the chunk shape

The size of a chunk must be set during field creation. Usually you do not have to care about chunking. By default *libpniio* sets the chunks shape to a reasonable value. However, in some special cases you may want to use a custom chunk shape. This can be done with the `nxgroup` member function templates used to create field, and with the corresponding

algorithm function. For the former one you can use

```
h5::nxgroup detector = get_object(root, "NXentry/NXinstrument/NXdetector");

shape_t shape = {0,1024,1024};
shape_t chunk = {1,1024,1024};

h5::nxfield data = detector.create_field<uint32>("data",shape,chunk);
```

Here we start with shape where the first dimension is 0. The idea is to grow the field along this dimension with every detector frame which should be stored. The chunk shape, however, must not be of size 0 (it would also not make sense). We chose here a chunk shape which covers a single frame. For 2D detectors this is a quite reasonable choice. This can also be used along with a compression filter

```
h5::nxdeflate_filter comp(2,true);
h5::nxgroup detector = get_object(root, "NXentry/NXinstrument/NXdetector");

shape_t shape = {0,1024,1024};
shape_t chunk = {1,1024,1024};

h5::nxfield data = detector.create_field<uint32>("data",shape,chunk,comp);
```

The same is true for algorithms version of `create_field`. The last line of the above snippet would change to

```
h5::nxfield data = create_field(h5::nxobject(detector), "data",shape,chunk,comp);
```


9. NeXus ASCII representation

In many cases (see Section ??) an ASCII representation of a NeXus-file is required. There are two principle directions an ASCII representation of a NeXus-file can be used

From NeXus to ASCII an existing NeXus-file is converted to its ASCII representation

From ASCII to NeXus a new NeXus-file is created based on the information provided by an ASCII representation of the file.

9.1. Use cases

9.1.1. Generating NeXus structures from XML

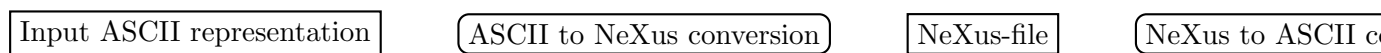
NeXus-files, when properly used, can store much more information than a typical ASCII data file. The structure of a NeXus file can become rather complex. In a rather complex environment like at a synchrotron beamline it would be naive to use static code to create the file. A more feasible approach would be to implement generic code which generates the structure of a file from an XML template. The template could be generated either manually by the user or by means of another program written in whatever language is reasonable for this purpose.

9.1.2. Metadata ingestion of a file

In some situations a 3rd party may needs to process some of the metadata stored in the file while not having access to the file. One possible application would be a data catalogue. Instead of making the file available the XML dump of the NeXus file could be sent to the system while moving the NeXus file to its final location on the storage system.

9.2. Necessary limitations of an ASCII representation

The ASCII representation of a NeXus-file is considered as a utility to provide a solution to the above use cases. Thus, every ASCII representation must have limitations. In particular a certain asymmetry exists between the ASCII representation obtained by one of the two basic use cases. The ASCII representation produced by a NeXus to ASCII converter is not necessarily exactly the same as the one used to create the file in the first place.



9.3. NeXus and XML

As NeXus organizes data objects in a tree like manner, XML is the obvious ASCII representation for a NeXus file. `libpniio` thus provides a small but powerful set of functions to read and write NeXus data objects from and to XML files. The framework is based on the `boost::property_tree` library. Clearly, the XML functionality provided by `libpniio` does in no case replace a full XML parser. For instance, there are no functions provided to manipulate the XML content returned from any of the functions. However, as the `node` type used to represent XML data is an alias for the `boost::property_tree` node type, one can use functions from `boost::property_tree` to do some additional work with the XML results. `libpniio` provides two interfaces

- a high level interface which consists of basically two functions. The high level interface is described in section 9.3.2
- And a low level interface – these are the classes and functions used to build the high level interface (described in section 9.3.3).

Aside from this there are some simple functions which are common to both interfaces and described in section 9.3.1.

9.3.1. Basic XML handling

All XML related functions and classes resided within a separate namespace `pni::io::nx::xml`. An XML document is represented by an instance of type `node`. `node` is an alias for `boost::property_tree::ptree`. Thus it can be used along with all functions and methods provided by the `ptree` type provided by `boost::property_tree`. Aside from the functionality from `boost::property_tree` `libpniio` provides some NeXus related convenience functions.

An instance of `node` can be created either from a string

```
xml::node n = xml::create_from_string(xmldata);
```

where `xmldata` is a string with the XML content, or from a file

```
xml::node n = xml::create_from_file(fname);
```

where `fname` is a string with the name of the file. Both functions return the root node of the XML document. See the `property_tree` documentation in the `BOOST` distribution for more information what one can do with such an object. For the rest of this chapter no additional knowledge about `BOOST`'s `property_tree` library is required. To write the content of a node to a stream one can use the default output stream operator

```
xml::node n = ....;
std::cout<<n<<std::endl;
```

9.3.2. High level XML interface

The high-level XML interface basically consists of two function templates

`xml.to_nexus` which creates NeXus objects from XML templates

`nexus.to_xml` which generates XML from NeXus objects.

Additionally there are some convenience functions available. All of this will be described in more detail in this section.

NeXus objects from XML

Creating NeXus objects from XML might be the most common operation performed by this part of the library. The next example illustrates the most common use case

```
#include <pni/io/nx/nx.hpp>
#include <pni/io/nx/xml.hpp>

using namespace pni::io::nx;

int main(int argc, char **argv)
{
    xml::node n = ....;
    h5::nxfile f = ....;
    h5::nxgroup root = f.root();

    xml::xml_to_nexus(n, root);

    return 0;
}
```

Here, the structure of a NeXus tree is described by XML and then converted to NeXus by means of the `xml_to_nexus` function. The first argument of this function is the XML node while the second is the parent object below which the new structure should be created. All data already available in the XML file will be stored in the fields and attributes of the created NeXustree.

XML from NeXus objects

The work horse for NeXus to XML conversion is the `nexus_to_xml` function template. The most probably simplest use case is demonstrated in the next example

```
#include <pni/io/nx/nx.hpp>
#include <pni/io/nx/xml.hpp>

using namespace pni::io::nx;

int main(int argc, char **argv)
{
    h5::nxfile f = ....;
    h5::nxgroup root = f.root();

    xml::node root_node;
    xml::nexus_to_xml(root, root_node);
    std::cout<<root_node<<std::endl;

    return 0;
}
```

Here, the entire structure of the NeXus file is stored below the XML root node which is at the end dumped to standard output. This simple example already raises an important question: how to deal with the data stored in the NeXus file. As NeXus files can be used to store large amounts of data it would not be wise to convert all this data to ASCII (think about a 3D

9. NeXus ASCII representation

image stack stored in the file). However, some data might be required. The `nexus_to_xml` template thus provides a third optional argument which is a predicate function which decides for which field or attribute data will be written to the file. The signature of the predicate is

```
template<
    typename GTYPE,
    typename FTYPE,
    typename ATYPE
>
bool predicate(const nxobject<GTYPE,FTYPE,ATYPE> &o);
```

The function returns `true` if the data of a particular object should be included in the XML output. It is wise to not make this function too specific. Thus, the name of a field is not a good criterion for deciding whether or not to write data. A much better approach is to check for certain properties of an object. For the previous example a possible predicate could look like this

```
//code omitted
bool write_scalars(const h5::nxobject &o)
{
    if(is_field(o) || is_attribute(o))
    {
        return size(o)==1;
    }
    else
        return false;
}

int main(int argc, char **argv)
{
    //code omitted
    xml::nexus_to_xml(root, root_node, write_scalars);

    //code omitted
    return 0;
}
```

This predicate determines that only the data from fields and attributes is written to the XML tree if their size is equal to 1 (in other words – only scalars are written to the file). Such an approach keeps the resulting XML document small while using a rather general predicate which would match quite a lot of use cases. The default policy is to write no data.

9.3.3. The XML low level interface

The entire XML stack in `libpniio` is based upon the `boost::property_tree` library. The `pni::io::nx::xml::ptree` type is nothing else than an alias to the `node` type provided by the `property_tree` library.

Basic node-operations

In order to simplify the work with `boost::property_tree` `libpniio` has introduced some convenience functions. To create a new XML node one could use either

```
xml::node = xml::create_from_string("...");
```

or

```
xml::node = xml::create_from_file("filename.xml");
```

The former function returns a node which refers to the root element of the XML structure provided by a string passed to the function as its only argument. The latter one reads the XML data from a file and returns a handle to its root element too.

`boost::property_tree` uses nodes to store attributes of a tag which are accessible under a special name. To make accessing the attributes simpler there two functions.

```
if(xml::has_attribute(field_node,"units"))
{
    xml::units_node = get_attribute(field_node,"units");

    // do something with the attribute
}
```

`has_attribute` returns true if a node possesses a particular attribute of given name and `get_attribute` returns a node instance referring to this attribute. The `xml::node` type and its related functions are provided by the `xml/node.hpp` header file.

Utility classes

There are two utility classes which should be described before we are dealing with the special classes for individual tasks. The two classes are

`data_node` a simple and special node which can be used to retrieve data from a node.

`io_node` the base class for the `attribute` and `field` class.

The `data_node` class provides only two static methods `read` and `write`. Both methods deal with the entire data content stored in a tag. To read the textual data stored in a tag one can simply use

```
xml::node n = ...;
string data = data_node::read(n);
```

and to write

```
xml::node n = ....;
string data = ....;
data_node::write(n,data);
```

The `io_node` class is used to access common properties of the `attribute` and `field` tag. The properties in common to attributes and fields are

- data type
- number of dimensions (rank)
- number of elements along each dimension (shape)
- total number of elements stored in the tag (size).

9. NeXus ASCII representation

The usage of the corresponding static methods is fairly simple

```
xml::node n = ...;
auto shape = xml::io_node::shape<shape_t>(n);
size_t rank = xml::io_node::rank(n);
size_t size = xml::io_node::size(n);
std::cout<<"Type ID: "<<xml::io_node::type_id(n)<<std::endl;
```

In addition to those methods there are two template methods which can be used to retrieve data from or to write data to the node. To read a single scalar value use

```
xml::node n = ...;
auto data = xml::io_node::data_from_xml<float64>(n);
```

one can also use `std::vector`

```
xml::node n = ...;
auto data = xml::io_node::data_from_xml<std::vector<float64>>(n);
```

Handling tags

Each tag in libpniio's XML protocol is handled by an individual class and its static methods. Each of these *tag*-classes provides two major methods `object_from_xml` and `object_to_xml`. The former one creates a new object of the tags type in memory while the latter one creates an XML representation from a particular type.

The dimensions tag

The `dimensions` tag provides information about the number of elements along each dimension of a multi-dimensional field or attribute. This tag is handled by the `dimensions` class provided by `xml/dimensions.hpp` header file. The `dimensions` class allows for inquiry of the `dimensions` tag within the XML structure.

```
xml::node dims = .....;

std::cout<<"rank: "<<dimensions::rank(dims)<<std::endl;
std::cout<<"size: "<<dimensions::size(dims)<<std::endl;
```

The `rank` member function returns the number of dimensions of the `dimensions` tag and the `size` method the total number of elements described by the `dimensions`-tag.

The memory representation of the `dimensions`-tag is a container whose elements are of type `size_t` or any other compatible integer type. The default container is the `shape_t` type provided by `libpnicore`. To create an instance of `shape_t` use

```
shape_t shape = dimensions::object_from_xml(dims);
```

However, there is also a template version of `object_from_xml`

```
typedef std::list<uint128> dim_type;
auto shape = dimensions::object_from_xml<dim_type>(dims);
```

which can be used to store the number of elements in an arbitrary container.

The inverse operation, building a `dimensions`-tag from a container is implemented by the `object_to_xml` member function.

```
shape_t shape{...};
xml::node dim_node = dimensions::object_to_xml(shape);
```

This template member function accepts an arbitrary container with integer elements as its only argument and return a `xml::node` instance with the shape data.

The attribute tag

The **attribute** tag is handled by a class of the same name which is a child class of `io_node`. Aside from the inherited interface from `io_node` the **attribute** class adds two additional static methods

`object_from_xml` which generates an attribute from an XML attribute tag

`object_to_xml` which converts a NeXus attribute to an XML tag.

It is important to note that this class deals only with attributes which are described by the attribute tag and not those which are part of the tag itself.

The group tag

The group tag is a simple container which can store either field or other group tags. The group tag has two attributes: the name of the group and its (base)class. At least the name of the group is mandatory and must be available in any case. The base class is optional. If not provided, a non-NeXus group without an `NX_class` attribute will be created.

The field tag

The **field** tag is handled by the `field` class which behaves pretty much as the **attribute** class. There is one major exception from the point of XML: fields do support compression. Currently only HDF5s deflate compression is supported. To add compression use the **strategy** tag inside the **field** tag. The usage of the **strategy** tag is a remembrance from Jan Kotanskies NeXus data server. It might be replaced in future as more compression algorithms become available.

The link tag

The link tag indicates that the object that should be created is a link to another object. Like the group tag it has only two attributes: the name of the link within its parent group and the target to which the link should refer. The target can be either an object in the same file (internal link) or an object in a different file (an external link). The `link` class which processes the information provided by the link tag has only an `object_from_xml` method as links are not indicated in XML output.

A. Parsing ASCII data

In times of binary data formats like Nexus it seems anachronistic to devote an entire chapter of the appendix to the problem of ASCII parsing. However, there are several good reasons why one should care about correctly reading ASCII data (in particular numbers)

1. for historical reasons there is a lot of legacy ASCII data out there which may should be processed – for this reason alone it is necessary to deal with ASCII data in a reasonable way.
2. uses still provided input to programs via ASCII files (for instance using XML) or via the command line – in both cases the program has to process these files correctly.
3. even input fields in GUI toolkits typically return the data entered by the user as a character string which must be parsed in order to obtain a numerical value.

One crucial aspect when processing ASCII data is number parsing. This chapter will describe in more detail the parser framework provided by `libpniio`.

All parsers basically utilize two exceptions to denote errors

`parser_error` which is thrown in situations where the ASCII representation is malformed, or

`range_error` which is thrown when the ASCII string is well formatted but the numeric range represented by the number exceeds the target type **not yet implemented**

This information should be sufficient to recognize the error in the input data. Before discussing the individual functions and types provided by `libpniio` for parsing ASCII a thorough discussion of the ASCII representations of the primitive data types will be made.

A.1. ASCII representations of primitive data types

A.1.1. Integers and floating point numbers

The ASCII representations of integer and floating point numbers follow the C++ standard conventions and will not be discussed here in greater detail.

A.1.2. Complex numbers

Complex numbers can be represented in two forms: as real and imaginary part or as imaginary part only. A real-part only representation is not possible as such a number would be indistinguishable from a simple floating point type. The full representation (real and imaginary part) looks somehow like this

$$A \pm KB \tag{A.1}$$

A. Parsing ASCII data

where A and B are denoting the real and imaginary part respectively. K is the symbol which denotes the complex unit $\sqrt{-1}$. This can be either i , j , or I . To give an example

`1.2+j3.4`

would be a valid complex number while

`1.2+J2.4`

not. It is important to note that the sign between the real- and imaginary-part must not be surrounded by blanks. Furthermore, the imaginary part must not have an extra leading sign as its sign is already determined by the sign between real- and imaginary part. A imaginary-part only number would look like this

`+i3`

where the sign in front of the complex unit is optional and could have been omitted in this particular case

`i3`

A.1.3. Boolean values

The two boolean values `true` and `false` are represented exactly in this way as ASCII strings. Be aware that the syntax is case sensitive. Thus `True` or `False` would cause a `parser.error`.

A.2. Parser rules

B. NeXus-XML protocol

`libpniio` uses XML an XML dialect very similar to NXDL in order to serialize NeXus-objects. The deviations from NXDL are due to the fact that `libpniio` has to build or dump concrete objects to XML rather than describing the fundamental properties (as NXDL does). A good example here is the data type. NXDL uses `NX_FLOAT` to denote that a field or attribute has to be a floating point type. However, it does not prescribe which one to use. For the purpose of NXDL this is not of importance - the data type must be any of the available floating point types. `libpniio`, on the other side, needs to write objects to as well as to construct objects from XML. Hence, a type like `NX_FLOAT` would be too general.

B.1. Common deviations from NXDL

This section deals with those deviations from NXDL which are common too all objects in the NeXus-universe.

B.1.1. Data types

While NXDL uses general type descriptors like `NX_FLOAT` or `NX_CHAR`, which are more closely related to `libpnicore`s type class concept, `libpniio` uses the string representation of data types as described by the `libpnicore` users guide. For a field one may use

```
<field name="data" type="float32"/>
```

where `float32` tells `libpniio` exactly which floating point type to use. Strings are represented by `string` instead of `NX_CHAR`.

B.1.2. Enumerations

`libpniio` does not support enumerations. An enumeration in NXDL lists the possible values a particular field or attribute can take. However, for `libpniio` the objects already has a particular state when it is written to XML or has to be constructed in a particular state when read from XML. Enumerations thus would not make too much sense.

B.1.3. Data in XML content

Unlike NXDL, `libpniio` allows data to be stored in the XML output. This is particularly useful for scalar and rather small arrays of numeric data and strings. For numeric data all features are allowed. Data from multi dimensional fields can easily be stored and read to and from XML. The data is simply dumped as a linear sequence of string representations of the numeric values separated by whitespace characters. As the dimensions of a multi dimensional field is stored in a special tag the field can easily be constructed from the linear storage.

The special case of strings

Strings are difficult. Thus only scalar string data can be currently stored. The reason for this is simple. While the ASCII representations of numeric data can easily be separated by whitespace characters, this is no longer the case for strings. The whitespace may be part of a single element. In other words, it is unclear what the delimiter characters are.

Binary data

B.2. XML attribute representation

Attributes have to be represented by a dedicated tag if they are not mandatory for a particular tag type. For instance the **group** and **type** attribute are mandatory for virtually all objects and are thus always part of a tag. However, the **transformation_type** attribute is optional for a **field** tag and thus has to be encoded in a separate attribute tag.

B.2.1. Scalar attributes

```
<attribute name="transformation_type" type="string"> rotation </attribute>
```

The **name** and **type** attributes have their obvious meaning and do not need extra explanation. The data is kept in the CDATA section of the tag and needs to be parsed in case of reading.

B.2.2. Multidimensional attributes

As of the NIAC meeting at the APS in December 2014 Nexus officially supports

```
<attribute name="vector" type="float32">
  <dimensions rank="1">
    <dim value="3" index="1"/>
  </dimensions>
  0 1 0
</attribute>
```

More interesting, how to manage multidimensional strings

```
<attribute name="log" type="string">
  <dimensions rank="1">
    <dim value="4" index="1"/>
  </dimensions>
  "line 1: blablabla" "line2: blablabla"
  "line 3: more text" "line 4: the last line"
</attribute>
```

Strings must be stored quoted. This is important for the case of multidimensional arrays as it would be hard otherwise to define a delimiter between the different strings. As a consequence it is not possible to store strings which contain a :

B.3. XML field representation

B.3.1. Scalar fields

```
<field name="integral" type="uint16" units="cps"> 102 </field>
```

The attributes `name` and `type` have the same meaning as for attributes. More interesting is the `units` attribute as its meaning differs from NXDL. In NXDL the `units` attribute describes the dimension (length, angular, etc.) while in our serialization approach it represents a physical units. In this case this would be *counts by second* (*cps*).

B.3.2. Multidimensional fields

Multidimensional fields are described like multidimensional attributes

```
<field name="rotmat" type="float64" units="a.u.">
  <dimensions rank="2">
    <dim index="1" value="3"/>
    <dim index="2" value="3"/>
  </dimensions>
  1 2 3
  4 5 6
  7 8 9
</field>
```

Unfortunately this is not enough when we consider HDF5 as a storage backend. We definitely need more information.

B.3.3. Adding chunking

The first thing required for HDF5 is that we have to store chunk information.

```
<field name="images" type="uint16" units="a.u.">
  <dimensions rank="3">
    <dim index="1" value="0"/>
    <dim index="2" value="1024"/>
    <dim index="3" value="2048"/>
  </dimensions>
  <chunk rank="3">
    <dim index="1" value="1"/>
    <dim index="2" value="1024"/>
    <dim index="3" value="2048"/>
  </chunk>
</field>
```

B.3.4. Adding compression

```
<field name="images" type="uint16" units="a.u.">
  <dimensions rank="3">
    <dim index="1" value="0"/>
    <dim index="2" value="1024"/>
    <dim index="3" value="2048"/>
  </dimensions>
```

B. NeXus-XML protocol

```
<chunk rank="3">
  <dim index="1" value="1"/>
  <dim index="2" value="1024"/>
  <dim index="3" value="2048"/>
</chunk>
<filters>
  <filter index="1" name="fletcher32"/>
  <filter index="2" name="deflate">
    <param name="rate"> 8 </param>
  </filter>
</filters>
</field>
```

Bibliography

- [1] Nicolai M. Josuttis David Vandevorode. *C++ Templates: The Complete Guide*. Addison Wesley, 2002.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.