Tutorial NeXus version 2.1.0

Contents:

Introductory Topics

1. Introduction

What NeXus Is

NeXus is a data format for the exchange of neutron and synchrotron scattering data between facilities and user institutions. It has been developed by an international team of scientists and computer programmers from neutron and X-ray facilities around the world.

The data format is

- portable
- binary
- extensible and
- self-describing.

The NeXus format defines the structure and contents of neutron/synchrotron data files in order to facilitate the visualization and analysis of these data. In addition, an Application Program Interface (API) has been produced in order to improve reading and writing of NeXus files.

Generally the NeXus format uses the HDF Data Format, developed by the National Center of Supercomputing Applications (http://ncsa.uiuc.edu). HDF, which stands for Hierarchical Data Format, is a common data format that has been developed to aid scientists and programmers in the storing, transfer and distribution of data sets and products created on various machines and with different software.

Tutorial Contents

This tutorial covers the basics of NeXus data objects, the file structure, and the NeXus API functions necessary for creating, reading and modifying data objects. All API functions are illustrated by simple examples for a better understanding.
The tutorial describes primarily the new version of NeXus API (version 2.1.0). This version supports both HDF-4 and HDF5 and it is the extension of the NeXus API version 1.3.3. The older version supports only HDF-4. Last but not least most of the tutorial topics are also relevant for the NeXus API version 1.3.3.

We hope that the step-by-step examples and instructions will be given a quick start with NeXus.
We are generally interesting in a continuous improvement of this tutorial. That's why it is desired that you send your comments and suggestions to uwe.filges@psi.ch !

A general description of NeXus can be found on the WWW – page:

http://lns00.psi.ch/NeXus/index.html

2. NeXus File Organization

A NeXus file is a container for storing neutron and x-ray scattering data and is composed of two primary types of objects:

- groups and
- datasets.

NeXus group

The NeXus grouping structure contains zero or more NeXus objects (e. g. sub-groups, datasets), together with supporting attributes. The NeXus groups are comparable to the directory structure of a UNIX/LINUX file system.
NeXus groups have a group name and an additional class name. In particular, we use group classes to define the type of group object and its expected contents whereas the group name labels a particular instance of that object. In some cases, the groups will actually define physical objects, such as crystal monochromators or disk choppers. In others, the group will define a logical set of descriptive data.

NeXus dataset

A NeXus dataset is a multidimensional array of data elements of any type, together with supporting attributes. The datasets are stored in the NeXus groups.

Supporting features of NeXus main objects

Any NeXus group or dataset object may have an associated attribute list. A NeXus attribute is a user-defined NeXus structure that provides extra information about a NeXus object. Additionally, NeXus files will themselves be annotated with global attributes, which are used to define the NeXus version, file owner in example.
For organizational reasons it might be useful to refer a dataset in more then one group. But it should be avoided to duplicate data. For this reason linking of groups or datasets are possible inside a NeXus file. This concept is quite similar to a symbolic link in a Unix file system.

More details to the NeXus structure are available under:

http://lns00.psi.ch/NeXus/NeXus_structure.html

3. The NeXus API based on HDF4 or/and HDF5

Concerning the update of NeXus API version 1.3.3 to version 2.1.0 it is important to know that the version 2.1.0 can use the HDF4 library and/or the HDF5 library. In praxis that means the user will be decided with the choice of NeXus installation (see next chapter) whether his application uses resources of

- HDF4 only
- HDF5 only
- both HDF4 and HDF5.

Here we want to cite the NCSA developer group:

"… Note that HDF and HDF5 are two different products. HDF is a data format first developed in the 1980s and currently in Release 4.x (HDF Release 4.x). HDF5 is a new data format first released in Beta in 1998 and designed to better meet the ever-increasing demands of scientific computing and to take better advantage of the ever-increasing capabilities of computing systems. HDF5 is currently in Release 1.x (HDF5 Release 1.x)."

This statement should underline that the new HDF5 data format is not a release version of the older HDF format. The formats are not compatible. Consequential the both HDF format require different tools each for data access and analysis.
The NeXus-API however is almost identical both for HDF-4 and HDF5. Only at file creation time it is necessary to decide which file type is requested.

The NeXus library provides several interfaces, or APIs. The library itself is implemented in C although we also provide Fortran 77 and 90 wrappers.

In the NeXus library all C routines begin with a prefix of the form NX*, where * stands for more as one lower case letters describing the type of object.

4. Installation Guide for the NeXus API

At the beginning we want to point out that this chapter will not cover the installation routines for all different platforms. It should be seen more as a general guide.

The first think is to decide which HDF library should be used (HDF library release 4.x or HDF5 library release 1.x)!

Generally it can be recommended to use the HDF5 library. The reasons are:

(1) the new HDF5 library removes some limitations of the older HDF format (e.g. now a single file can store more than 20.000 objects and can be larger than 2 Gb)
(2) the HDF5 data model is nearly equivalent to the data model of NeXus (The HDF5 data model includes two basic structures: a multidimensional array of record structures, and a grouping structure like NeXus).
(3) the HDF5 library has some potential for further improvements of the NeXus-API (e.g. mounting files).

Of course we also appreciated that it is important to support backward compatibility with existing NeXus applications.

That's why we are meaning that a user who already works with NeXus (HDF-4 based version) should not hesitate to install the NeXus API version 2.1.0 using both HDF-4 and HDF5 libraries. A NeXus newcomer should prefer a NeXus installation only using the HDF5 data format.

The next step is to install the appropriate HDF/HDF5 library on your machine. It is helpful that the NCSA developer group offers a large amount of HDF binary distributions for common platforms.

After the successful installation of the HDF library the NeXus API must be created. A C-compiler must be available on your system in order to create the API.

If you are looking into the makefiles (*Makefile* and his sub-makefiles *makefile_hdf4*, *makefile_hdf5* and *makefile_hdf45*) which are a part of the new NeXus package) then you can find there a typical installation routine for UNIX/LINUX file systems. If you call the main makefile by typing:

make

into a terminal window in the directory to which the new NeXus source code was copied. The output will be:

make: lib4 lib5 lib45 !

In the next step you must choose one of these three options. For example you must type <make lib4> into the terminal window when your application should use only the HDF library release 4.x . Consequentially your application can use the HDF and HDF5 libraries when you call <make lib45>.

Remark
You may need to edit the makefiles in order to have the paths to the prior installed HDF libraries.

It is important to know that the NeXus source code will be compiled partial depending from the defined library. From the user side it is stringently necessary to set the compiler flag –DHDF4 and/or –DHDF5 of option NOPT in the used makefile (or in a developer environment like the Visual C++ studio). The following table shows an overview to the flag options.

Flag	linked Library
HDF4	only HDF release 4.x
HDF5	only HDF5 release 1.x
HDF4/HDF5	HDF release 4.x and HDF5 release 1.x

Table 1: Flag options for linking the NeXus API

If the flag was set, the appropriated functions will be linked into the NeXus API.

Additionally a test program and a simple NeXus browser will be created with the above mentioned makefiles. After the successful installation you can run the test program. If all things are working correctly you can start the programs *test5* (lib45) or *test5o* (lib5) producing the following screen output.

NeXus_version : 2.1.0.
file_name : NXtest.h5
HDF5_Version : 1.4.2
file_time : 2001-11-23 14:13:04+0100
ch_data : 4
Values : NeXus data
i1_data : 20
Values : 1 2 3 4
i2_data : 22
Values : 1000 2000 3000 4000
i4_data : 24
Values : 1000000 2000000 3000000 4000000
r4_data : 5
Values : 1.000000 2.000000 3.000000 4.000000
: 5.000000 6.000000 7.000000 8.000000
: 9.000000 10.000000 11.000000 12.000000
: 13.000000 14.000000 15.000000 16.000000
: 17.000000 18.000000 19.000000 20.000000
Dataset has no attributes!
r8_data : 6
Values : 1.000000 2.000000 3.000000 4.000000
: 5.000000 6.000000 7.000000 8.000000
: 9.000000 10.000000 11.000000 12.000000
: 13.000000 14.000000 15.000000 16.000000
: 17.000000 18.000000 19.000000 20.000000
ch_attribute : NeXus
i4_attribute : 42
Number of attributes: 3
Number of attributes: 0
Group:'entry/detector/data' with class name: NXdata and 7 item(s)
HDF5_Version: 1.4.2 Length:(5)
Number of attributes: 4

5.Creating, Opening, and Closing a NeXus File

NeXus files are created/opened with the NXopen() function which requires the following three parameters.

NXopen (*char** file_name, *int* access_method, *NXhandle* file_id)

The first parameter *file_name* is of course the name of the NeXus file. The second parameter *access_method* is the access mode valid for the file. Four access modes are supported:

NXACC_READ
: open a NeXus file in read only mode. The file could be written with the NeXus API 1.x (HDF-4) or 2.x (HDF5)

NXACC_RDWR
: Opening an existing NeXus file for modification or for appending. Also here the file format can be HDF-4 or HDF5

NXACC_CREATE
: Create a new NeXus file using the HDF-4 library

NXACC_CREATE5

Create a new NeXus file using the HDF5 library

If the access method is NXACC_ READ or NXACC_RDWR the NXopen function checks automatically for the appropriate HDF format.

The last parameter *file_id* is a pointer to handle for the NeXus file and will be used in subsequent alls in order to refer to the file.
Closing files is accomplished through the NXclose() function with the file_id parameter.

Example 1: Creating a NeXus file using the HDF5 library

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>
<8>   if (NXopen (file_name, NXACC_CREATE5, &fileid) != NX_OK) return 1;
  :
<9>   if (NXclose(&fileid) != NX_OK) return 1;
<10> }
```

Remarks

Line 1: including the definition file of the NeXus API
Line 5: the file_name parameter specifies the name of the file to be created
Line 6: the data type NXhandle is defined in napi.h as a void pointer
Line 8: create the file “NXtest.h5” using the HDF5 file format; output the fileid for the file handling; the function returns NX_OK = 1 if successful; otherwise returns NX_ERROR = 0; in case of NX_ERROR the program will be stopped
Line 9: the file “Nxtest.h5” is closed using the fileid; the function returns NX_OK = 1 if successful; otherwise returns NX_ERROR = 0;

6. Working with Groups

A group is the NeXus equivalent of a directory. Alike to a directory hierarchy, a hierarchy of groups can be built in a NeXus file (figure1).

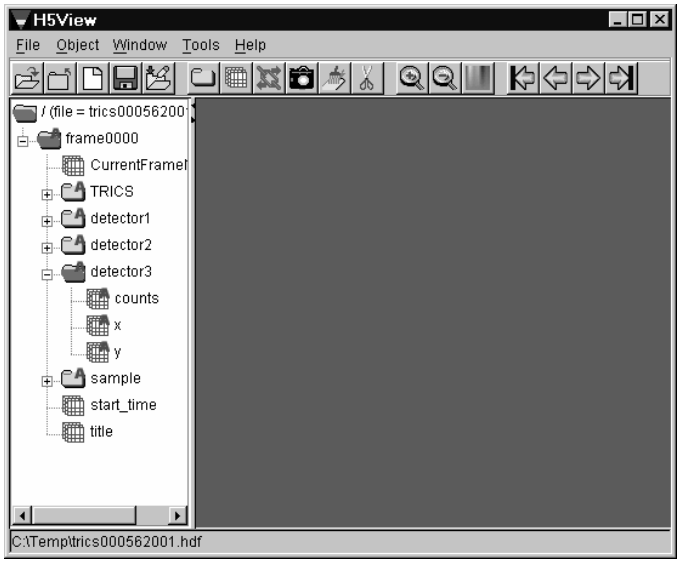

Figure 1: Typical NeXus structure with groups and sub-groups

For creating a new group (or directory) hierarchy the functions NXmakegroup(), NXopengroup() and NXclosegroup() are used. The sequence of functions is:

(1) Creating the group with the function NXmakegroup ();
(2) Opening the group with the function NXopengroup();
(3) Closing the group with the function NXclosegroup ();

Between step (2) and (3) maximum 32 sub-groups can be created using the same function sequence (1) to (3). The named functions take the NXhandle argument in order to refer to the opened NeXus file.

The function

NXmakegroup (NXhandle file_id, char* group_name, char* group_class)

requires two further parameter. NeXus groups are specified by two string items: the group name (equivalent to a directory name) and the group class (interpretable as meaningful labels for user applications). In this context we will be mentioned that using of group classes is a special feature of the HDF-4 standard. In opposite the new HDF5 standard don't know anymore group classes. But for backward compatibility the group class parameter is supported also in the new NeXus API version. The difference to the older version is that NeXus code based on HDF5 creates an additional group attribute of type character and copies inside the content of group class. Note, the NXmakegroup don't open the group.

The group name includes only the name of the group (sub-group) and not the full path inside the NeXus file. The position of a new group depends from the group stack of the opened NeXus file.

In order to use a group we need a means of traversing the group hierarchy. For this the functions:

NXopengroup (NXhandle file_id, char* group_name, char* group_class)

and

NXclosegroup (NXhandle file_id)

are provided. The NXopengroup function is used to open an existing group (sub-group). The parameters group_name and group_class are used to identify the group. The opening parameters are identical to the NXmakegroup parameters. The NXclosegroup function is the opposite of the NXopengroup function. The function closes the active group (sub-group) and steps one group lower in the group hierarchy.

Example 2: Creating/Opening NeXus groups

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>
<8>   if (NXopen (file_name, NXACC_CREATE5, &fileid) != NX_OK) return 1;
<9>      if (NXmakegroup (fileid, "001entry", "NXentry") != NX_OK) return 1;
<10>    if (NXopengroup (fileid, "001entry", "NXentry") != NX_OK) return 1;
<11>         if (NXmakegroup (fileid, "001data", "NXdata") != NX_OK) return 1;
<12>         if (NXopengroup (fileid, "001data", "NXdata") != NX_OK) return 1;
<13>         if (NXclosegroup (fileid) != NX_OK) return 1;
<14>    if (NXclosegroup (fileid) != NX_OK) return 1;
<15>  if (NXclose(&fileid) != NX_OK) return 1;
<16> }
```

Lines 1- 8: see example 1
Line 9: create the group *001entry* in the root level – directory level “/001entry”
Line 10: open the group *001entry*
Line 11: create the subgroup *001data* of the opened group *001entry* – directory level “/001entry/001data”
Line 12: open the group *001data*
Line 13: close the group *001data* – go down to directory level “/001entry”
Line 14: close the group *001entry* – go down to root level “/”
Line 15 close the file Nxtest.h5

Remark

In the example all group names have a numerical prefix in form of ‘00x’. The reason is HDF5 stores alphabetically the group names. But the NeXus file structure should

reflect the creating sequence of the groups. This goal is reached by using the named prefix. Concerning the example a group with name ‘data’ would stand in the order before group ‘entry’ although at first the group ‘entry’ was created. In difference the NeXus API based on HDF-4 stored automatically group and data names concerning creating sequence (a prefix is not necessary).

7. Creating a Dataset

A dataset is a multidimensional array of data elements, together with supporting metadata. To create a dataset, the application program must specify the location at which to create the dataset, the dataset name, the data type and data itself.

NeXus uses the native data types from the HDF5 library. The data types are mapped to a NeXus number type. The following table lists all allowed number types of NeXus.

Name	Description
NX_CHAR	8 bit character
NX_INT8	8 bit integer (byte)
NX_UINT8	8 bit unsigned integer
NX_INT16	16 bit integer
NX_UINT16	16 bit unsigned integer
NX_INT32	32 bit integer
NX_UINT32	32 bit unsigned integer
NX_FLOAT32	32 bit float
NX_FLOAT64	64 bit float (double)

Table 2: NeXus API data types

These number types are defined as constants in the napi.h and will be transferred in the appropriate HDF data types.

When creating a new file a means is needed for creating datasets in the new NeXus file. A dataset is fully characterized by its name (parameter *data_name*), its number type (out of the list above – *data_type* parameter), the number of dimensions it has (its rank) and its size in each dimension. With this information a dataset can be created with the function:

NXmakedata (NXhandle file_id, char* data_name, int data_type, rank, int dims[])

with dims[] being an integer array holding the size of the dataset in each dimension. The NXmakedata function takes the NXhandle argument in order to refer to the opened NeXus file.

An example for creating a dataset will be shown in chapter 8.

8. Reading from or Writing to a Dataset

Before writing/reading data is possible, the call NXopendata is required. Note, the NXmakedata function does not automatically open the dataset. Analog to a file in a file system a dataset must be opened before anything can be done with it and closed when processing is finished. The appropriate calls are:

NXopendata (NXhandle file_id, char* data_name)

and

NXclosedata (NXhandle file_id).

The parameters of both functions are simple and will be demonstrated in Example 3.

If a dataset is open data can be read out or written to it. Two means of data transfer functions are provided.

NXputdata (NXhandle file_id, void* data)

NXgetdata (NXhandle file_id, void* data)

The NXputdata function writes data to a dataset, which has been opened before. The function NXgetdata is used to read data from an existing dataset. Both functions NXputdata and NXgetdata transfer a whole dataset in one go. The data to write to or to read from is expressed as the pointer to void parameter in the function calls. These pointer must point to a data array matching the size of the dimension of the dataset. Otherwise your program might start behaving very strangely. Suitable sized arrays can be allocated with the utility functions NXmalloc and free with NXfree.
NXputdata and NXgetdata take the NXhandle argument *file_id* in order to refer to the opened NeXus file.

Example 3: Writing and reading a dataset

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   int counts[4] = {100, 200, 300, 400};
<7>   int data_buffer[4];
<8>   int n_t = 4;
<9>   NXhandle fileid;
<10>
<11>   /* writing dataset */
<12> NXopen (file_name, NXACC_CREATE5, &fileid);
<13>     NXmakegroup (fileid, "001entry", "NXentry");
<14>     NXopengroup (fileid, "001entry", "NXentry");
<15>         NXmakedata (fileid, "001counts", NX_INT32, 1, &n_t);
<16>         NXopendata (fileid, "001counts");
<17>         NXputdata (fileid, counts);
<18>         NXclosedata (fileid);
<19>     NXclosegroup (fileid);
<20> NXclose(&fileid);
```

```
<21>
<22> /* reading dataset */
<23> NXopen (file_name, NXACC_RDWR, &fileid);
<24>    NXopengroup (fileid, "001entry", "NXentry");
<25>        NXopendata (fileid, "001counts");
<26>        NXgetdata (fileid, data_buffer);
<27>        printf("1. data value :  %d\n", data_buffer[0]);
<28>        NXclosedata (fileid);
<29>    NXclosegroup (fileid);
<30> NXclose(&fileid);
<31> }
```

Line 6: declaration of data for writing and reading
Line 12: a new NeXus file will be created
Line 13/14: the group '001entry' will be created and opened
Line 15: the dataset '001counts' is created in the group '001entry'with datatype 32 bit integer, rank 1 and dimension 0;
Line 16: opening the dataset '001counts'
Line 17: writing of the array counts[4] in the dataset '001counts'
Line 18-20: closing dataset, group and file
Line 23: opening the existing file NXtest.h5 with READ/WRITE access
Line 24: the existing group '001entry' is opened
Line 25: the existing dataset '001counts' is opened
Line26: the dataset '001counts' is reading in the variable data_buffer
Line27: the first value of data_buffer is printed out on the display
Line 28-30:closing dataset, group and file

Remark
More helpful reading/query functions will be described in the Advanced Topics of this tutorial!

9. Reading/ Writing Attributes

Attributes are auxiliary information stored in a NeXus file. There are three variants: global attributes at root level, group attributes and attributes at dataset level. The attribute part of the API acts on global attributes and group attributes if no dataset is open. To write a attribute on the dataset level the appropriate dataset must be opened. Attributes can be written with the

```
NXputattr (NXhandle file_id, char* attr_name, void* value, int length, int type)
```

and read with the

```
NXgetattr (NXhandle file_id, char* attr_name, void* value, int length, int type)
```

function.

Both functions use also the NXhandle argument *file_id* in order to refer to the opened NeXus file.

The parameter *attr_name* is used to define the name of attribute, which will be created (NXputattr) or read (NXgetattr). The parameter *value* is a 1-dimensional data array of different kind of data (e.g. strings). The size of the array is defined by the *length* parameter and parameter *type* specifies the data type of the attribute. For the data type parameter the input must be conform to the number types of table 2. For example a string attribute with data 'He-3 detector' has a *length = 13* and a *type = NX_CHAR*. The following example shows writing and reading of attribute of type float containing the *pi* value. The number of values can be also increased through increasing the parameter *length*.

Example 4: Writing/Reading an attribute

```
<1> #include "napi.h"
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = "NXtest.h5";
<6>   NXhandle fileid;
<7>   float r, data_buffer;
<8>   int NXlen, NXtype;
<9>
<10>   NXopen (file_name, NXACC_RDWR, &fileid);
<11>   NXputattr(fileid, "file_owner", "J.Miller", strlen("J.Miller"), NX_CHAR);
<12>     NXopengroup (fileid, "001entry", "NXentry");
<13>          r = 3.1415;
<14>          NXputattr(fileid, "001attr", &r, 1, NX_FLOAT32);
<15>          NXlen = 1;
<16>          NXtype = NX_FLOAT32;
<17>          NXgetattr(fileid, "001attr", &data_buffer, &NXlen, &NXtype);
<18>          printf ("001 attribute :  %f\n", data_buffer)
<19>     NXclosegroup (fileid);
<20> NXclose(&fileid);
<21> }
```

Line 9: the file Nxtest.h5 is opened with READ/WRITE access

Line 10: a global attribute with name *file_owner* is copied in the root group ("/"); the attribute is from type string and has the value *J.Miller*

Line 12: opening the group *001entry*

Line 13: initialization of variable r with the value of Pi

Line 14: writing the attribute 001attr; the attribute is from type float and includes the value Pi in variable r with a length of one

Line 17: the prior written attribute *001attr* is read in the variable data_buffer (remark: NXlen and NXtype can be also determined automatically with the function NXgetnextattr)

Line 15: variable *data_buffer* is printed out on the display

Remark

More helpful attribute functions will be described in the Advanced Topics of this tutorial! The additional functions are important for the work with unknown file structures.

Advanced Topics

A. Selecting a Portion of a Dataset

Hyperslabs are portions of datasets. A hyperslab selection can be one point in a dataset, or it can be a block in a dataset. You can select a hyperslab to write to or read from with the NeXus functions

NXputslab (NXhandle file_id, int data, int start[], void* size[])

and

NXgetslab (NXhandle file_id, int data, int start[], void* size[]).

The function NXputslab writes a hyperslab of a multidimensional data array, specified by the starting indices and size of each dimension, into the currently open dataset.

In opposite the function NXgetslab reads a hyperslab of the data in the current dataset specifying the starting indices and size of each dimension. The caller is also responsible for allocating enough memory for the data.

For illustration a 5 x 6 integer array is defined. Inside this array a hyperslab (like 3x4 in figure 2) or one point P[2,2] of the array can be selected.

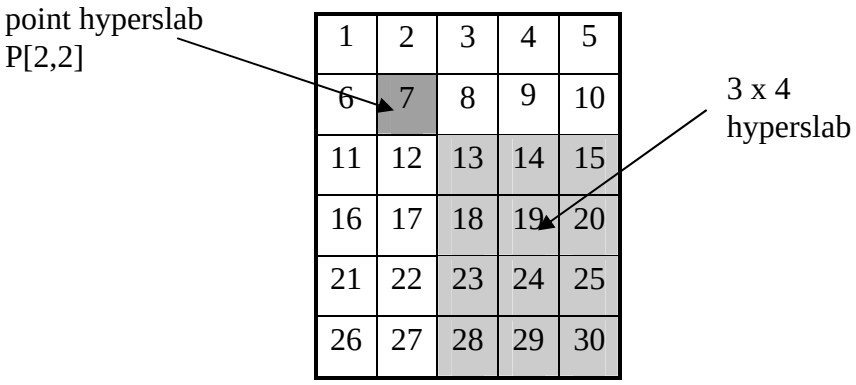

Figure 2: 5x6 integer array with selection of point und block hyperslab

Example 5: Writing and reading a hyperslab of data

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>   int counts[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
<8>   int array_dims[2] = {4,4};
<9>   int slab_start[2], slab_size[2];
<10> int data_buffer[4];
<11>
<12>   /* writing hyperslab of data */
<13> NXopen (file_name, NXACC_CREATE5, &fileid);
<14>     NXmakegroup (fileid, "001entry", "NXentry");
<15>     NXopengroup (fileid, "001entry", "NXentry");
<16>         NXmakedata (fileid, "001counts", NX_INT32, 2, array_dims);
<17>         NXopendata (fileid, "001counts");
<18>         slab_start[0] = 0; slab_start[1] = 0; slab_size[0] = 2; slab_size[1] = 4;
<19>         NXputslab (fileid, counts, slab_start, slab_size);
<20>         NXclosedata (fileid);
<21>     NXclosegroup (fileid);
<22> NXclose(&fileid);
<23>
<24> /* reading hyperslab of data */
<25> NXopen (file_name, NXACC_RDWR, &fileid);
<26>     NXopengroup (fileid, "001entry", "NXentry");
<27>         NXopendata (fileid, "001counts");
<28>         slab_start[0] = 1; slab_start[1] = 0; slab_size[0] = 1; slab_size[1] = 4;
<29>         NXgetslab (fileid, data_buffer, slab_start, slab_size);
<30>         printf(“Values :  %d\n“, data_buffer)
<31>         NXclosedata (fileid);
<32>     NXclosegroup (fileid);
<33> NXclose(&fileid);
<34> }
```

Line 7: declaration of data for writing and reading
Line 13: a new NeXus-HDF5 file will be created
Line 14/15: the group ‘001entry’ will be created and opened
Line 16: the dataset ‘001counts’ is created in the group ‘001entry’with datatype 32 bit integer, rank 2 and dimension [4,4];
Line 17: opening the dataset ‘001counts’
Line 18: definition of hyperslab for writing – a 2x4 block is selected with the values {1, 2, 3, 4, 5, 6, 7, 8}
Line 19: writing of the 3x3 integer array counts in the dataset ‘001counts’
Line 20-22: closing dataset, group and file
Line 25: opening the existing file NXtest.h5 with READ/WRITE access
Line 26: the existing group ‘001entry’ is opened
Line 27: the existing dataset ‘001counts’ is opened
Line 28: definition of hyperslab for reading – a 1x4 block is chosen with the values {5, 6, 7, 8}
Line 29: the hyperslab of dataset ‘001counts’ is reading in the variable data_buffer
Line 30: the data_buffer is printed out on the display
Line 31-33: closing dataset, group and file

B. Linking of Groups and Datasets

The NeXus standard sometimes requires that a given dataset is accessible in different groups. However, we do not want to duplicate data. The solution is to use a reference to an already written dataset at the location (inside the group hierarchy) where the dataset also be found. Such a reference is called a link in NeXus API terminology. These links are similar to symbolic links in a UNIX file system. Complete groups can be linked as well.

Linking a dataset or group requires some precautions. First some information needed for linking must be retrieved while the group or dataset (which will be linked) is still open. The calls:

NXgetgroupID (NXhandle file_id, NXlink* group_id)

and

NXgetdataID (NXhandle file_id, NXlink* data_id)

get the identifiers (references) *group_id* or *data_id* of the currently open group or dataset as an NXlink structure. The NXhandle argument covers (like all other function) the *file_id* getting from the NXopen function.
Then, after moving to the new location (whither the data should be linked) the call:

NXmakelink (NXhandle file_id, NXlink* link_id)

will install finally the link. The parameter *link_id* can include a *group_id* as well a *data_id* taken with the above mentioned functions.

Example 6: Linking a group and a dataset

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>   int counts[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
<8>   int array_dims[2] = {4,4};
<9>   NXlink glink, dlink;
<10>
<11> NXopen (file_name, NXACC_CREATE5, &fileid);
<12>     NXmakegroup (fileid, "001entry", "NXentry");
<13>     NXopengroup (fileid, "001entry", "NXentry");
<14>         NXgetgroupID(fileid, &glink);
<15>         NXmakedata (fileid, "001counts", NX_INT32, 2, array_dims);
<16>         NXopendata (fileid, "001counts");
<17>         NXgetdataID(fileid, &dlink);
<18>         NXputdata (fileid, counts);
<19>         NXclosedata (fileid);
```

```
<20>     NXclosegroup (fileid);
<21>     NXmakegroup (fileid, "002link", "NXentry");
<22>     NXopengroup (fileid, "002link", "NXentry");
<23>         NXmakelink(fileid, &glink);
<24>         NXmakelink(fileid, &dlink);
<25>     NXclosegroup (fileid);
<26> NXclose(&fileid);
<27> }
```

Line 11: a new NeXus-HDF5 file will be created
Line 12/13: the group ‘001entry’ will be created and opened
Line 14: getting of the group ID of ‘001entry1’ group
Line 15: the dataset ‘001counts’ is created in the group ‘001entry’with datatype 32 bit integer, rank 2 and dimension [4,4];
Line 16: opening the dataset ‘001counts’
Line 17: getting of the data ID of ‘001counts’ dataset
Line 18: writing of data into ‘001counts’ dataset
Line 19/20: closing dataset ‘001counts’ and group ‘001entry’
Line 21/22: the group ‘002link’ will be created and opened
Line 23: making the link to the ‘001entry’ group – structure is now: 002link/001entry1 with the content of dataset ‘001counts’
Line 24: making the link to the ‘001counts’ dataset – inside the ‘002link’ group the dataset ‘001counts’ exists
Line 25/26: closing group ‘002link’ and file

C. Creating Extendible Datasets

NeXus allows you to define datasets where the first dimension is extendible. It is possible to append data along the first dimension. This feature is needed if the number of dimensions is unknown for a measurement parameter.
To create an extendible dataset the known function NXmakedata (part I section 7) is used. The parameter *dims[]* must include as first dimension the keyword ‘NX_UNLIMETED’ (e.g. a1_dim[NX_UNLIMETED, dimension of second rank,..]) in order to define that the first dimension can be increased to run time of the application. In particular the function

NXputslab (NXhandle file_id, void* data, int start[],int size[])

is used to extend the dimension. The function supports to write slabs of a dataset. In opposite to the NXputdata function two additional parameters are available. The parameter *start[]* gives the possibility to choose a starting location for data writing. The parameter *size[]* defined the size of extension beginning from the starting location (parameter *start[]*).The array will be automatically extended.

Example 7: Creating an extendible dataset

```
<1> #include “napi.h”
<2>
<3>   int main()
```

```
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>   int i, slab_start[1], slab_size[1];
<8>   int ext_dim[1] = {NX_UNLIMITED};
<9>
<10> NXopen (file_name, NXACC_CREATE5, &fileid);
<11>     NXmakegroup (fileid, "001entry", "NXentry");
<12>     NXopengroup (fileid, "001entry", "NXentry");
<13>         NXmakedata (fileid, "001counts", NX_INT32, 1, ext_dim);
<14>         slab_size[0] = 1;
<15>         for (i=0; i<10; i++) {
<16>             slab_start[0] = i;
<17>             NXopendata (fileid, "001counts");
<18>             NXputslab (fileid, &i, slab_start, slab_size);
<19>             NXclosedata (fileid);
<20>         }
<21>     NXclosegroup (fileid);
<22> NXclose(&fileid);
<23> }
```

Line 10: a new NeXus-HDF5 file will be created
Line 11/12: the group ‘001entry’ is created and opened
Line 13: the dataset ‘001counts’ is created as an extendible dataset, the variable ‘ext_dim’ includes the keyword ‘NX_UNLIMITED’
Line 14: the slab size is set on 1
Line 15 – 20: the dimension of dataset ‘001counts’ is increased by steps of one and is filled by the value of the variable i (1-9), using the NXputslab function.
Line 21/22: closing group ‘001entry’ and file

D. Compressed Datasets

The possibility to compress data is an important feature of the NeXus format. An efficient storage of measured data is helpful in case of large amount of data. Note, the way to store compressed data is changed with the new NeXus API version 2.1.0 .

D1. Using NeXus API version 2.1.0

An especially API function was written to support data compressing in NeXus version 2.1.0 . Here the compatibility to the older version could be not observed. The new function is called with

NXcompmakedata (NXhandle file_id, char* data_name, int data_type, int rank, int dims[], int compress_type, int chunk_size[]).

The function is an extension of the already explained function NXmakedata(). This function was extended with two additional parameters in order to define the compression parameters. The parameter *compress_type* is used to set the compression method. Presently NeXus API version 2.1.0 supports as compression method only the gzip type compression (keyword: ‘NX_COMP_LZW’). The other new parameter

means *chunk_size* and is needed for an improved performance of data storage. The chunk_size parameter is an array of dimension *rank*, which holds sizes for each dimension. Although it is most efficient if I/O requests are aligned on chunk boundaries, this is not a constraint.

Beyond the used function sequence must be changed in opposite the older NeXus API version. The setting of compress parameters must be stand in front of the function NXopendata. The new function sequence is:

<1> NXcompmakedata()
<2> NXopendata()
<3> NXputdata()
<4> NXclosedata().

At first the dataset must be created with the compression parameters (NXcompmakedata). Subsequently the dataset can be opened (NXopendata) and then the compressed data can be written (NXputdata).

Note: The NXmakedata function is not needed for data compression!

D2. Using NeXus API version 1.3.3

The function

NXcompress (NXhandle file_id, int compress_type,)

is needed to store compressed data using the NeXus API version 1.3.3. The function uses also the NXhandle argument *file_id* in order to refer to the opened NeXus file. The compression method is selected with the second parameter. The following table contains the possible compression methods and the appropriate keyword for the API function.

The function sequence is:

<1> NXmakedata()
<2> NXopendata()
<3> NXcompress()
<4> NXputdata()
<5> NXclosedata().

At first the dataset is created normally with the NXmakedata function. Subsequently the dataset is opened. Thereupon the compression parameters are set for the opened dataset and then the data can be written (NXputdata) with the selected compression method.

Example 8: Creating a compressed dataset (NeXus version 2.1.0)

<1> #include “napi.h”

```
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>   int chunk_size [2];
<8>   int counts[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
<9>   int array_dims[2] = {4,4};
<10>
<11> NXopen (file_name, NXACC_CREATE5, &fileid);
<12>     NXmakegroup (fileid, "001entry", "NXentry");
<13>     NXopengroup (fileid, "001entry", "NXentry");
<14>         chunk_size[0] = 2;
<15>         chunk_size[1] = 2;
<16>         NXcompmakedata (fileid, "001counts", NX_INT32, 2, array_dims,  NX_COMP_LZW,
                chunk_size);
<17>         NXopendata (fileid, "001counts");
<18>         NXputdata (fileid, counts);
<19>         NXclosedata (fileid);
<20>     NXclosegroup (fileid);
<21> NXclose(&fileid);
<22> }
```

Line 11: a new NeXus-HDF5 file will be created
Line 12/13: the group ‘001entry’ is created and opened
Line 14/15: the chunk_size for each dimension is set
Line 16: the dataset ‘001counts’ is created, a 4x4 integer array, chunked 2x2 using the compression method LZW
Line 17: the dataset ‘001counts’ is opened
Line 18: the data from variable ‘counts’ are written into the dataset ‘001counts’
Line 19-21: closing dataset ‘counts’, group ‘001entry’ and file

E. Additional Query Functions

E1. Listing group contents

Reading the structure of a NeXus file can be done with the functions NXgetgroupinfo and NXgetnextentry. The first function is used to query the number of objects in a current opened group (including root group).
The call is:

NXgetgroupinfo (NXhandle file_id, int item_number, char* group_name, char* group_class).

The parameter *item_number* returns the number of items in the current group. Additionally the name and class of the current group is returned. Subsequently the returned number of objects is used for analyzing the group objects step by step (using a looping structure – example 9). For this job the NeXus API provides the function

NXgetnextentry (NXhandle file_id, char* object_name, char* object_class, int* data_type).

The output parameters *object_name*, *object_class* and *data_type* can be used subsequently for reading data with the standard reading function NXgetdata. If the object is a group, only its name and class is returned.

Example 9: Listing group structure

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [9] = “NXtest.h5”;
<6>   NXhandle fileid;
<7>   int numb, i, nxtype;
<8>   char group_name[64], class_name[64], name[64];
<9>
<10>   if (NXopen (file_name, NXACC_READ, &fileid) != NX_OK) return 1;
<11>      if (NXgetgroupinfo (fileid, &numb, group_name, class_name) != NX_OK) return 1;
<12>      For (i=1; i<=numb; i++) {
<13>          if (NXgetnextentry (fileid, name, class_name, &nxtype) != NX_OK) return 1;
<14>          printf(“object name: %s, object class: %s, data type: %d\n”, name, class_name, nxtype);
<15>      }
<16>   if (NXclose(&fileid) != NX_OK) return 1;
<17> }
```

Line 10: the existing file NXtest.h5 is opened in the *Read* mode
Line 11: the number of items inside the root group is read in (parameter *numb*)
Line 12-15: step by step the NeXus objects are analyzed; The name, class and data type of objects are printed out (If the item is a group, its name and class is returned. For groups the value of data type is zero.).
Line 16: the file is closed

E2. Listing attributes

Listing of attributes of a dataset or group is another feature of the NeXus API. For it a combination of two NeXus functions are needed. At first the function NXgetattrinfo is used to determine the number of attributes. The call is

NXgetattrinfo (NXhandle file_id, int* attr_number).

The output is the parameter *attr_number*. Subsequently the returned number of attributes is used for reading the attributes completely (using a looping structure like presented in the example of E1). Inside the loop the call

NXgetnextattr (NXhandle file_id, char* attr_name, int* length, int* type)

is used to read the attribute name, its dimension (parameter *length*) and the number represents of the data type (each attribute step by step - the NXgetnextattr function increases automatically the pointer of the attribute number). Then the standard function NXgetattr can be applied for reading the attribute values.

E3. Query of dataset information

When dealing with an unknown NeXus file we might need to find out about the characteristics of a current opened dataset. The name and class of a dataset was found out before with the function NXgetnextentry. The function

NXgetinfo (NXhandle file_id, int rank, int dims[], int data_type)

Offers the opportunity to read out the characteristics of a dataset. The first output parameter is the *rank* of the dataset. The second output parameter *dims[]* will hold the size of the dataset in each dimension. Make sure, that dims[] is large enough to hold all dimensions. Note: 32 is the maximum number of dimensions supported by NeXus. The last output parameter is *data_type*, which includes the number represents the data type of the dataset.

F. Conversion Tools

h4toh5 converter

h4toh5 is a file conversion utility that reads an HDF4 file, and writes an HDF5 file, containing the same data. If no output file is specified, h4toh5 derives the output filename from the input filename by replacing the extension .hdf with .h5. For example, if the input file scheme3.hdf is specified with no output filename, h4toh5 will name the output file scheme3.h5.

Example:

```
h4toh5  h4file.hdf  h5file.h5
```

Each object in the HDF4 file is converted to an equivalent HDF5 object, according to the mapping described in the paper ‘Mapping HDF4 Objects to HDF5 Objects’ of NCSA.

h5toh4 converter

The h5toh4 converter is the tool for conversion of HDF5 to HDF-4 file and is called in an equivalent manner like the h4toh5 converter.

Example:

```
h5toh4 h5file.h5 h4file.hdf
```

G. Using the NXdict-API

Writing NeXus files with the NeXus Core API includes a large amount of repetitive code to implement the NeXus structure. Now, repetitive tasks are one area a computer is good at. That's why the NXDICT (**N**e**X**us **DICT**ionary) concept was initiated. The NXDICT approach reduces programming effort on the user side. NXDICT has the additional benefit that if the file structure changes it is sufficient to edit the dictionary data file with no changes to the source code writing or reading the data.
NXDICT's purpose is to define the structure and data items in a NeXus file in a form which can be understood by a human programmer and which can be parsed by the computer in order to create the structure. For this a dictionary based approach will be used. This dictionary will contain pairs of short aliases for data items and definition strings holding the structure information.

The dictionary can be used in the following way: A NXDICT programmer needs to specify only the alias and the data to write and everything else is taken care of by the NXDICT API, which uses the NeXus Core API in order to do its job. For example creation, opening groups, writing data in the group and closing the dataset and the group will be done in one step.

Another use may involve the creation of definition strings completely or partly at run time. Then the string can be used by an API function in order to create the structures defined by the definition string. The same holds for writing as well.

In comparison to the standard NeXus functions the use of NXDICT API functions require to create an additional ASCI-II file (dictionary file, which needs the starting line *##NXDICT-1.0*). The dictionary file includes the alias for use in the data file.

The alias structure is:

alias = definition string.

The definition string holds all information of the position of a data item inside of a NeXus file. The definition string has the following structure:

/root group name, class of root group/1. subgroup name, class of 1.subgroup/ .../KEYWORD name **-type** *data_type* **-rank** *number* **-dim** *{dim0,dim1,…,dimn} -compression type* **-chunk** *{size0,size1,…,sizen}* **-attr** *{name,value}.*

The first part of the string (up to the KEYWORD parameter) describes the path or position of a group or dataset. Remember that a group requires additionally a class name. That's why two items (group name and the appendant class name) are necessary. A valid path string would be:

/frame0001,NXentry/instrumentx,NXinstrument/...

The next part of the definition string is a KEYWORD. Three different keywords are defined:

- NXVGROUP
- NXLINK

- SDS

The NXVGROUP keyword is only useful for the definition of links to groups. After this keyword no further options are admitted. The NXLINK keyword offers the possibility to link a NeXus object. The keyword must follow a valid alias to another object. The SDS keyword follows a more complex structure. The SDS keyword indicates that the definition string describes a dataset item. This keyword is followed by options, which define the characteristics of the dataset item. The following options exist:

- *name* specifies the name of the dataset (must be always be there)
- **-type** *data_type* defines the data type of the dataset; 'data_type' may be all NeXus data types (see chapter 7)
- **-rank** *number* defines the rank of the dataset
- **-dim** *{dim0, dim1,...,dimn}* defines the dimensions length. The number of ranks must be equal to the number of dimension parameters.
- *compression type* defines the kind of compression. Possible values are LZW, HUF and RLE (remember that NeXus version 2.1.0 only supports the LZW compression method)
- **-chunk** *{size0,size1,...,sizen}* defines the chunk size. The options must be set if a compression method's selected. The number of chunk size parameter is equal to rank number and should not be equal to the number of dimension parameters.
- **-attr** *{name,value}* defines an attribute

If nothing is specified except the name, a dataset is creates which holds a single floating point value.
As an example see the definition of 3D array 'test' of 32 bit integer compressed by LZW method and an attribute 'units'. The 'test array' is created in the directory '/frame0000/instr1'.

```
alias = /frame0000,NXentry/instr1,NXinstrument/SDS test -type NX_INT32 \
        -rank 3 -dim {20,20,10} -LZW -chunk {4,4,2} -attr {units,counts}
```

The following functions are used for accessing and maintaining a dictionary.

```
NXDinitfromfile(char *filename, NXdict *pData)
NXDclose(NXdict handle, char *filename)

NXDputalias(NXhandle hFil, NXdict dict, char *pAlias, void *pData)
NXDgetalias(NXhandle hFil, NXdict dict, char *pAlias, void *pData)
NXDaliaslink(NXhandle hFil, NXdict dict,char *pTarget, char *pSource)
```

The first one NXDinitfromfile creates or opens a dictionary file. If filename not available then a new dictionary file will be created.

The NXDclose function is used to write a dictionary file. Subsequently the dictionary structure is closed. If filename is NULL, no file is written.

The next functions are really used to write, read or link data. The function NXDputalias is used to write data using an alias name from the dictionary. Equivalent the NXDgetalias function reads data.

The last main function NXDaliaslink supports linking. With 'pTarget' the linking position is defined. The ‘pSource’ variable includes the NeXus object which should be linked. In both cases aliases can be used.

At the end a very helpfule utility function should be named. The NXUwriteglobals function offers a simply way to write all necessary staff concerning global attributes.

The structure is: NXUwriteglobals(NXhandle pFile,
char *filename,
char *owner,
char *adress,
char *phone,
char *email,
char *fax,
char *instrument name).

The content of the function is self-describing.

A more detailed description of NXDICT functions can be downloaded from:

http://lns00.psi.ch/NeXus/NeXus_API.html - Core

Example 10: Using NXdict library for writing a NeXus file

Dictinary File instr1.dict

```
<1> ##NXDICT-1.0
<2> temp1 = /entry1,NXentry/TRICS,NXinstrument/SDS temperature \
<3> -type NX_FLOAT32 -rank 1 -dim {5} -attr {Units,Fahrenheit}
<4> temp2 = /entry1,NXentry/sample,NXsample/SDS name \
<5> -type NX_CHAR -rank 1 -dim {12}
<6> det1 = /entry1,NXentry/detector1,NXinstrument/SDS counts \
<7> -type NX_INT32 -rank 2 -dim {20,20} -attr {Units,counts} \
<8> -LZW -chunk {5,5} -attr {axis,1}
<9> data_link = /entry1,NXentry/NXVGROUP
```

Line 1: starting line of each dictionary file

Line 2-3: definition of an alias name for creating the dataset ‘temperature’ under path entry1/TRICS; the dataset is from type FLOAT32; the dataset has a rank of one with 5 dimensions; additionally the dataset has an attribute ‘Units’ of type character with the value ‘Fahrenheit’

Line 4-5: definition of an alias name for creating the dataset ‘name’ under path entry1/sample; the dataset is from type CHARACTER; the dataset has a rank of one and a length of 12

Line 6-8: definition of an alias name for creating the dataset ‘counts’ under path entry1/detector1; the dataset is from type INT32; the dataset has a rank of two with 20 dimensions for each rank; additionally the dataset has an

attribute ‘Units’ of type character with the value ‘counts’; the dataset will be compressed with the chunking size 5*5

Line 9: definition of an alias name for linking a dataset

Data file dict1.c

```
<1> #include "dynstring.h"
<2> #include "napi.h"
<3> #include "nxdict.h"
<4>
<5>   int main() {
<6>      NXdict pDict = NULL;
<7>      NXhandle hfil;
<8>      float temp1_write[5] = { 20.5, 30.2, 45.4, 64.8, 110.0};
<9>      float temp_read[5];
<10>     char name[12] = "D20 30K SNP";
<11>     char pBuffer[132];
<12>     int counts[20][20];
<13>     int i, j;
<14>
<15>     /* test nxdict */
<16>     NXDinitfromfile("instr1.dict",&pDict);
<17>     NXopen("trics01.h5", NXACC_CREATE5, &hfil);
<18>     NXUwriteglobals(hfil, "trics01.h5", "Uwe Filges",
<19>          "Paul Scherrer Institut", "+41-56-3104606",
<20>          "Uwe.Filges@psi.ch", "+41-56-3102939","TRICS");
<21>     NXDputalias(hfil,pDict, "temp1", temp1_write);
<22>     NXDgetalias(hfil,pDict, "temp1", temp_read);
<23>     for (i=0;i<5;i++){
<24>        printf("Temperture T[%d]: %f\n",i,temp_read[i]);
<25>      }
<26>     NXDputalias(hfil,pDict, "temp2", name);
<27>     for (i=0;i<20;i++){
<28>        for (j=0;j<20;j++){
<29>           counts[i][j] = random(100);
<30>        }
<31>     }
<32>     NXDputalias(hfil, pDict, "det1", counts);
<33>     NXDaliaslink(hfil,pDict, "data_link", "det1");
<34>     NXclose(&hfil);
<35>  }
```

Line 1-3: including header-files for working with NXdict

Line 16: opening the dictionary file ‘instr1.dict’ and the dictionary is initialized

Line 17: creating the file ‘trics01.h5’ in HDF5 format using the standard function NXopen

Line 18-20: writing all global attributes in one step using the utility function NXUwriteglobals; the function includes the parameter sequence ‘filename; user name; address, phone number, e-mail, fax number, instrument name’

Line 21: writing the data ‘temp1_write’ specified by the alias ‘temp1’ (from file instr1.dict) to the NeXus file; the dataset has the name ‘temperature’

Line 22: reading the dataset ‘temperature’ into the variable ‘temp_read’ from the NeXus file using again the alias ‘temp1’

Line 23-25: printing out of variable ‘temp_read

Line 26: writing the data ‘name’ specified by the alias ‘temp2’ (from file instr1.dict) to the NeXus file
Line 27-31: the ‘counts’ array is filled by random values
Line 32: writing the data ‘counts’ specified by the alias ‘det1’ (from file instr1.dict) to the NeXus file; the dataset has also the name ‘counts’
Line 33: Linking the dataset ‘counts’ from group ‘/entry1/detector1’ to group ‘entry1’ using aliases ‘data_link’ and ‘det1’ from file ‘instr1.dict’
Line 34: closing the file ‘trics01.h5’

H. Creating the NeXus File layout (example)

The following chapter describes the recommended layout of NeXus files. The defined layout is just an agreement on what information is included and in what order. Generally it is recommended to organize the NeXus file in four sources of information:

(1) administrative information
(2) sample information
(3) instrument information
(4) experimental data.

The different kinds of information are assigned to different V-groups inside the NeXus hierarchy. The administrative information stands in front of the file as global attributes. Following the other three kinds of information are stored in a main Vgroup NXentry. This structure allows to store several related datasets in one file. The main Vgroup NXentry contains an instrument V-group, a sample Vgroup and a data Vgroup. The instrument Vgroup has a number of further sub-Vgroups which respectively describes a component of instrument.

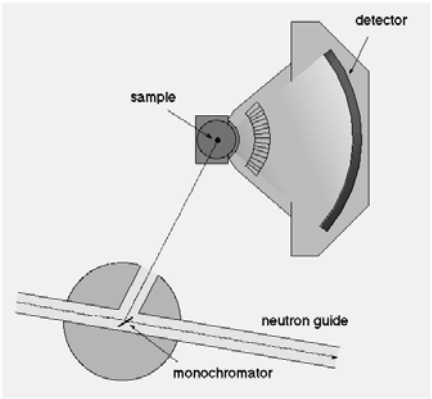

Figure 3: Simple layout of a Powder Diffractometer

The described layout should be shown at a simple powder diffractometer (figure 3), which contains only a monochromator, a sample and a detector system. The appropriate NeXus layout would be

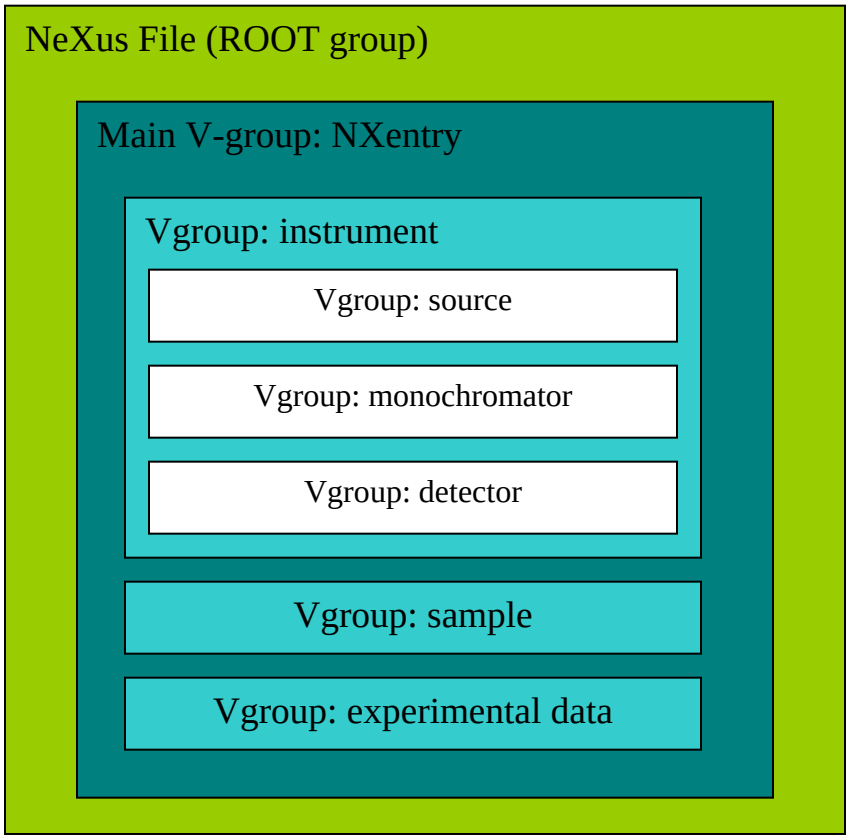

Figure 4: NeXus file layout

Following the appropriate NeXus application

Example 11: Creating Layout of a NeXus file

```
<1> #include “napi.h”
<2>
<3>   int main()
<4>  {
<5>   char file_name [11] = “NXlayout.h5”;
<6>   NXhandle fileid;
<7>
<8>    NXopen (file_name, NXACC_CREATE5, &fileid);
<9>       \* Location for writing global attributes containing administrative information *\
<10>  NXmakegroup (fileid, "001entry", "NXentry");
<11>          NXopengroup (fileid, "001entry", "NXentry");
<12>          NXmakegroup (fileid, "001instrument", "NXinstrument");
<13>                  NXopengroup (fileid, "001instrument", "NXinstrument");
<14>                  NXmakegroup (fileid, "001source", "NXsource");
<15>                          NXopengroup (fileid, "001source", "NXsource");
<16>                              \* Location for writing source information *\
<17>                          NXclosegroup (fileid);
<18>                  NXmakegroup (fileid, "002monochromator", "NXcrystal");
<19>                          NXopengroup (fileid, "002monochromator", "NXcrystal");
<20>                              \* Location for writing monochromator information *\
<21>                          NXclosegroup (fileid);
<22>                  NXmakegroup (fileid, "003detector", "NXdetector");
<23>                          NXopengroup (fileid, "003detector", "NXdetector");
```

```
<24>                                \* Location for writing detector information *\
<25>                            NXclosegroup (fileid);
<26>                    NXclosegroup (fileid);
<27>            NXmakegroup (fileid, "002sample", "NXsample");
<28>                    NXopengroup (fileid, "002sample", "NXsample");
<29>                        \* Location for writing sample information *\
<30>                    NXclosegroup (fileid);
<31>            NXmakegroup (fileid, "003data", "NXdata");
<32>                    NXopengroup (fileid, "003data", "NXdata");
<33>                        \* Location for writing data information *\
<34>                    NXclosegroup (fileid);
<35>            NXclosegroup (fileid);
<36>    NXclose(&fileid);
<37> }
```