1 NeXus Programmers Reference

Mark Koennecke
January 2007

This is a description of the internal working of the ANSI-C language NeXus-API. This is required reading for everyone who attempts to make changes to the NeXus core API. But be warned: this is not for the faint hearted. A successfull NeXus-API hacker needs a solid understanding of advanced C programming techniques and the HDF-4, HDF-5 and Mini-XML API's. Including their quirks and limitations. And for writing language bindings one needs to know about the foreign function interface conventions of the target language of the binding.

2 The Top Level NeXus API

The NeXus-API consists of a set of functions for creating NeXus files and storing data and attributes in them. All user visible NeXus data types and functions are protoyped in the header file: napi.h. besides the normal function protoypes, there als exists prototypes for function names adjusted in such a way that they can be called from FORTRAN. Also several internal support functions for the FORTRAN interface are defined.

As of 2007, the NeXus-API supports three different file formats: HDF-4, HDF-5 and XML. This has a couple of implications: One of them is that there are two categories of functions in the NeXus-API:

- Most functions are specific to the actual file format used.
- Some functions can be expressed in terms of the more primitive file access functions. Such functions are directly implemented in napi.c.

In an object oriented world the issue with the file type specific funtions would be solved through ploymorphy: there would be a base class specifying the interface and file type specific derived classes which overload the methods with appropriate ones. For reasons of portability the NeXus group choose to implement the NeXus-API in C. Thus polymoprhy had to be implemented in plain C. In order to do this all NeXus functions either create or take a pointer to a NeXus private data structure as a parameter. The current implementation of this data structure looks like this:

```
typedef struct {
   NXhandle *pNexusData;
   int stripFlag;
   NXstatus ( *nxclose)(NXhandle* pHandle);
   NXstatus ( *nxflush)(NXhandle* pHandle);
   NXstatus ( *nxmakegroup) (NXhandle handle, CONSTCHAR *name, CONSTCHAR* NXclass);
   NXstatus ( *nxopengroup) (NXhandle handle, CONSTCHAR *name, CONSTCHAR* NXclass);
   NXstatus ( *nxclosegroup)(NXhandle handle);
   NXstatus ( *nxmakedata) (NXhandle handle, CONSTCHAR* label, int datatype,
                            int rank, int dim[]);
   NXstatus ( *nxcompmakedata) (NXhandle handle, CONSTCHAR* label, int datatype,
                                int rank, int dim[], int comp_typ, int bufsize[]);
```

```
   NXstatus ( *nxcompress) (NXhandle handle, int compr_type);
   NXstatus ( *nxopendata) (NXhandle handle, CONSTCHAR* label);
   NXstatus ( *nxclosedata)(NXhandle handle);
   NXstatus ( *nxputdata)(NXhandle handle, void* data);
   NXstatus ( *nxputattr)(NXhandle handle, CONSTCHAR* name, void* data, int iDataLen,
                          int iType);
   NXstatus ( *nxputslab)(NXhandle handle, void* data, int start[], int size[]);
   NXstatus ( *nxgetdataID)(NXhandle handle, NXlink* pLink);
   NXstatus ( *nxmakelink)(NXhandle handle, NXlink* pLink);
   NXstatus ( *nxmakenamedlink)(NXhandle handle, CONSTCHAR *newname, NXlink* pLink);
   NXstatus ( *nxgetdata)(NXhandle handle, void* data);
   NXstatus ( *nxgetinfo)(NXhandle handle, int* rank, int dimension[], int* datatype);
   NXstatus ( *nxgetnextentry)(NXhandle handle, NXname name, NXname nxclass,
                               int* datatype);
   NXstatus ( *nxgetslab)(NXhandle handle, void* data, int start[], int size[]);
   NXstatus ( *nxgetnextattr)(NXhandle handle, NXname pName, int *iLength, int *iType);
   NXstatus ( *nxgetattr)(NXhandle handle, char* name, void* data, int* iDataLen,
                          int* iType);
   NXstatus ( *nxgetattrinfo)(NXhandle handle, int* no_items);
   NXstatus ( *nxgetgroupID)(NXhandle handle, NXlink* pLink);
   NXstatus ( *nxgetgroupinfo)(NXhandle handle, int* no_items, NXname name,
                               NXname nxclass);
   NXstatus ( *nxsameID)(NXhandle handle, NXlink* pFirstID, NXlink* pSecondID);
   NXstatus ( *nxinitgroupdir)(NXhandle handle);
   NXstatus ( *nxinitattrdir)(NXhandle handle);
   NXstatus ( *nxsetnumberformat)(NXhandle handle, int type, char *format);
   NXstatus ( *nxprintlink)(NXhandle handle, NXlink* link);
} NexusFunction, *pNexusFunction;
```

Basically this structure holds another pointer to a file type specific data structure and a lot of function pointers for the NeXus functions. A typical top level NeXus-API function implementation then looks like this:

```
NXstatus  NXmakegroup (NXhandle fid, CONSTCHAR *name, CONSTCHAR *nxclass)
{
     pNexusFunction pFunc = handleToNexusFunc(fid);
     return pFunc->nxmakegroup(pFunc->pNexusData, name, nxclass);
}
```

It just exchanges the fid against a pointer to the NexusFunction structure described above and calls the appropriate file type specific function.

Now a careful reader should ask how the NexusFunction structure is initialized to point to applicable functions for each file type. This happens in the NXopen function. NXopen figures the file type out by either looking at file creation flags or through inspection of the actual NeXus file to be openend. It then initializes the NexusFunction structure and proceeds to call the file type specific nxopen function in the NexusFunction structure.

NXopen has to implement another complication: the NeXus-API searches NeXus files in a NeXus search path defined through the environment variable NX_LOAD_PATH. This is done in NXopen; the located file is then opened through NXinternalopen.

The NeXus-API has an external file linking feauture. When the NeXus-API encounters a group with the attribute *napimount* it looks for a URL to a group in another file and opens the group in the other file without the user noticing anything. If such a special group is closed, the external file must be closed and the original file reenterd. This had to be implemented at the top level as the external linking feauture is supposed to work on top of any of the supported file formats. The functions NXopengroup and NXclosegroup have been instrumented in a suitable way to support external linking. But in order to do this some information is needed:

- The nesting hierarchy of files
- The information when to close an external file and step back into the source file of an external link.

This information is held in a file stack which is implemented in nxstack.h and nxstack.c. When entering a file the files name and corresponding NexusFunction structure is pushed onto the stack. When a file is closed, this data is popped again. For the test when an externally linked file is to be closed, the NXlink IDs as returned by NXgetgroupID are used. A pointer to such a file stack is currently the actual structure to which the NeXus file handle NXhandle points to. External linking is also the cause for the call to handleToNexusFunc in the function example above: this function retrieves the appropriate NexusFunction structure from the file stack.

Thus the NeXus-API can be approached as consisting of three different layers:

1. The file stack layer
2. The NexusFunction layer which implements file type polymorphy
3. A driver layer implementing the functions to access files in the different file formats of NeXus.

2.1 NeXus-API Error Handling

All API's throw errors any now and then. Most are caused by the user, the rest are more serious... Whatever, the NeXus-API programmer needs a way to process such errors. In many cases the NeXus default: printing the error to stdout is good enough. But in a GUI one might pop up a message box or in a Java wrapper one might want to convert the error into an exception. The good news is that the NeXus-API is designed to support this. All errors are reported through a function NXIReportError. NXIReportError has the signature:

```
void NXIReportError(void *pData, char *errorText);
```

A NeXus-API user now can replace the default error reporting function through an own implementation with the function NXMSetError. NXMSetError also allows to pass in a pointer to a user defined data structure which is passed as pData to the error reporting function.

3 NeXus File Drivers

3.1 NeXus HDF File Drivers

The NeXus file drivers for the HDF file formats HDF-4 and HDF-5 share common features. I recall that NeXus uses a hierarchy in order to organize information storage. But both HDF-API's are not tree based but use an interface which allows to open and close groups and datasets. This is sensible as HDF is designed to support very large data sets which may not necessarily fit into a computers memory in one go. But this also implies that the current position in the hierarchy of a given NeXus file has to be maintained by the NeXus-API. Such information is maintained in a stack which is pushed an popped while moving through the hierarchy. This stack also has to hold the positions within pending group and attribute searches through the NXgetnextentry and NXgetnextattr functions. Otherwise recursive searches would break.

Both HDF APIs make extensive use of integer ID's which act as handles to file interfaces and HDF objects. Of course the HDF NeXus file data structures must maintain a fair share of such ID's too. Great care has to be taken to release all used ID's at the appropriate time. Otherwise memory may be leaked. Or worse things may happen.

3.2 HDF-4 NeXus File Driver

It is worthwhile to know that each HDF-4 object in a HDF-4 file is unambigously identified through its tag and reference (ref) ID. Which happen to be integer numbers. For the following discussions it is also worth to know that Vgroups in HDF-4 are implemented as lists of reference and tag IDs of the objects contained in the Vgroup. The HDF-4 API is very rich. NeXus only uses a subset of the HDF-4 API, namely the Vgroup , the SDS and the annotation interface.

The HDF-4 NeXus file driver internally uses this data structure:

```
typedef struct __NexusFile {
  struct iStack {
    int32 *iRefDir;
    int32 *iTagDir;
    int32 iVref;
    int32 __iStack_pad;
    int iNDir;
    int iCurDir;
  } iStack[NXMAXSTACK];
  struct iStack iAtt;
  int32 iVID;
  int32 iSID;
  int32 iCurrentVG;
  int32 iCurrentSDS;
  int iNXID;
  int iStackPtr;
  char iAccess[2];
} NexusFile, *pNexusFile;
```

iStack The hierarchy stack.

iStackPtr a pointer into the hierarchy stack.

iAtt for storing the state of an attribute search.

iVID The ID for the Vgroup interface. To be used when interacting with Vgroups.

ISID The ID for the SDS interface. To be used when interacting with datasets.

iCurrentVG The Id of the currently open Vgroup

iCurrentSDS the ID of the currently open SDS. Must be 0 if no SDS open.

iNXID an identifier for this data structure.

iAccess The access code (read or write) for this file.

The hierarchy stack has the following fields:

iVref The reference ID's of previous Vgroups.

iRefDir an array of reference numbers used during searches with NXgetnextentry.

iTagDir an array of tag numbers used during searches.

iCurDir the current index into iRefDir and iTagDir.

iNDir the length of iRefDir and iTagDir.

__iStack_pad This makes compilers on 64-bit operating systems happy.

At this point it is convenient to discuss how group searches with NXgetnextentry work. On the first call to NXgetnextentry, all reference and tag ID's in the current group are read and copied into iRefDir and iTagDir. iNDir is set to the total number of objects held. iCurDir is set to 0 and data for the first object returned. Subsequent calls to NXgetexentry increment iCurDir and return appropriate data. Until the directory is exhausted and NX_EOD is returned. The internal functions NXIInitDir and NXIKillDir help with the management of this.

All group and SDS search code in the NeXus-HDF-4 driver suffer from the fact that different search functions have to be used when searching at root level or within a Vgroup. NXIFindVGroup and NXIFindSDS are helper functions for locating the appropriate objects. Both return the reference ID of a suitable object on success or NX_EOD in the case of failure.

Attribute searches are simpler: HDF-4 objects have arrays of attributes. And the ID of an attribute is simply the index into that array. Thus the total number of attributes is stored in iNDir at the start of an attribute search and iCurDir set to 0. Further calls increment iCurDir and return appropriate data until the attributes are exhausted. (NOTE: Here is a subtle bug waiting to happen: when a group search is mixed with an attribute search, things may go wrong. It would be better to have separate fields for the attribute search. That this has not been noticed yet is partly due to the fact that group attributes were introduced only recently).

Another issue is the implementation of named links. This is links to other objects in the NeXus file which appear under a different name in the linking Vgroup. This is not supported by the HDF-4 API: objects are identified by their reference ID and tags and names and class names are attributes to the object. This was solved by creating SDSs and Vgroups with the required name. Such objects then have an attribute *NAPIlink* which holds the tag and reference IDs of the linked item. The internal function findNapiClass and NX4opengroup and NX4opendata check for the existence of this attribute and act accordingly.

The initializations in NX4open and NXclose have to happen in the order as implemented. Otherwise ugly things may happen. It just does not work.

NX4flush is implemented as a close and a open of the file. This is because HDF-4 has no proper flush.

3.3 HDF-5 NeXus File Driver

The HDF-5 API addresses objects in HDF-5 files through unix like path strings. This requires some string processing in the file driver implementation. HDF-5 does not support class names for groups as HDF-4 did. The HDF-5 NeXus file driver solves this problem through the use of a group attribute called NX_class. There are also no file global attributes as in HDF-4. Such attributes are implemented as attributes to the root (/) group in the HDF-5 file driver.

Searches in HDF-5 work differently too: HDF-5 provides iterators which call a user supplied function for each element to be searched. The user supplied function then must store the data it needs about the element in an own data structure. The return value of the user supplied function also determines how the iteration proceeds.

Another source of complexity is HDF-5 data transfer. HDF-5 transfers data between a file data space and an in memory data space. Each of these data spaces has its own type, size etc. And each of these items has its own ID. This causes an proliferation of IDs. The nice thing about the scheme though is that the HDF-5 library takes care of all necessary conversions which need to happen between the various data spaces.

A special topic is closing files. In any other API closing a file also removes all resources associated with the file. This is by default not the case with HDF-5: if any ID is not released the HDF-5 library will keep the file open somehow. The HDF-5 team told me that some (paying) customer wanted this. Anyway: with the call to H5Pset_fclose_degree in NX5open proper operation is reestablished again: i.e. a call to NX5close really closes the file. Additioannly there is some code in NX5open which can print the number of handles left open. This can be useful for debugging.

The HDF-5 NeXus driver private data structure is this:

```
typedef struct __NexusFile5 {
      struct iStack5 {
        char irefn[1024];
        int iVref;
        int iCurrentIDX;
      } iStack5[NXMAXSTACK];
      struct iStack5 iAtt5;
      int iVID;
```

```
        int iFID;
        int iCurrentG;
        int iCurrentD;
        int iCurrentS;
        int iCurrentT;
        int iCurrentA;
        int iNX;
        int iNXID;
        int iStackPtr;
        char *iCurrentLGG;
        char *iCurrentLD;
        char name_ref[1024];
        char name_tmp[1024];
        char iAccess[2];
  } NexusFile5, *pNexusFile5;
```

iAtt The hierarchy stack

iStackPtr the pointer into the hierarchy stack.

iFID HDF-5 file handle

iCurrentG handle of the current open group.

iCurrentD handle of currently open dataset

iCurrentT handle to type of currently open dataset

iCurrentS handle to data space of currently open dataset

iCurrentA temporary handle of an open attribute.

iNX used in group searches

iNXID signature of data structure

iCurrentLGG name of last openened group

iCurrentLD name of last openen dataset. Has length 0 when no dataset open.

name_ref path to current group

name_tmp some group path

iAccess file access code

The hierarchy stack has the fields:

irefn The name of the group

iVref handle to group

iCurrentIDX the current position in a group search

3.4 XML Nexus-API Driver

The XML format for NeXus was demanded for two reasons:

- XML is the buzzword of the day
- People want a format where they can edit their data with an editor.

In due course a NeXus-XML file format had been defined. However, XML has one problem: it is not designed to handle large amounts of numeric data well. This showed during a survey of XML parsing libraries: most would handle a large block of numbers as a large block of text which is very unwieldly for handling numbers. The implementation in Mike Sweets Mini-XML library was slightly better: a node would be created for each number. Still this is difficult for copying data in and wastes a lot of space. This is another difference to the HDF data formats: for XML the whole tree would have to be read into memory for reading or created in memory before it could be written to file. This is because XML has no way to address single objects in a file as HDF has. A way to circumvent this problem was to introduce a custom data node into Mini-XML together with user definable callback functions for reading and writing such data. This was done and was included into the standard Mini-XML library by its author: Mike Sweet. The custom data which is used to keep data in memory is an own abstraction of a multi dimensional dataset. This is implemented in nxdataset.h and nxdataset.c. Most of the functions there are straight forward. The custom I/O functions required to interface to Mini-XML live in the files nxio.h and nxio.c. The heart of this are the callback functions:

nexusTypeCallback callback to determine the type of a node when reading.

nexusLoadcallback a callback function for reading numeric data.

nexusWriteCallback a callback function to write numeric data.

Then there are some pretty self explaining support functions.

With this out of the way most of the NeXus-API could be expressed in terms of the Mini-XML tree navigation and creation functions. Data transfer works through the nxdataset functions. But there is a limitation: unlimted dimensions are not supported for XML.

But there is another complication: XML does not know a thing about links. For links a special node with the name NAPIlink was introduced. An attribute *target* to such a node points to the target of the link. If the link is a named link the NAPIlink node will also have an attribute *name*. The implementations of NXXopengroup, NXXopendata NXXclosegroup and NXXclosedata check for NAPIlink nodes and silently follow them. However this means that the XML driver jumps criss and cross through the Mini-XML tree structure when following links. In order to remember to which node to go back a stack was once again needed in the XML-NeXus driver data structure.

```
typedef struct {
    mxml_node_t *current;
    mxml_node_t *currentChild;
    int currentAttribute;
```

```
}xmlStack;
/*---------------------------------------------------------------------*/
typedef struct {
  mxml_node_t *root;           /* root node */
  int readOnly;                /* read only flag */
  int stackPointer;            /* stack pointer */
  char filename[1024];         /* file name, for NXflush, NXclose */
  xmlStack stack[NXMAXSTACK];  /* stack */
}XMLNexus, *pXMLNexus;
```

The stack structures fields are:

current the current node

currentChild The current child from which to continue in a group search

currentAttribute The number of the current attribute in an attribute search.

The XMLNexus structure has:

root The root of the Mini-XML node tree

readOnly a flag for a read only file

stack The stack

stackPointer The current position in the stack

filename the filename of the XML-neXus file

To make this explicitly clear: Reading and writing NeXus-XML files works as operations on in memory trees. This has a couple of consequences:

- NeXus-XML is not suitable for large datasets. If you have very large datsets: use HDF-5!
- A file will only be writen when a NXflush or a NXclose is called.

Another limitation is that dataset compression does not make sense in XML. The compression related functions are empty.

4 NeXus Language Bindings

Various bindings exist from the C-language NeXus-API to other programming languages. This section discusses implementation details for some of the supoorted language bindings.

4.1 FORTRAN 77

The F77 language bindings reside in the files napif.inc and napif.f. Napif.inc must be included by all programs using the NeXus F77 language bindings. Napif.f defines various constants and all NeXus functions as functions returning integers. Napif.f then implements the actual F77-API. Mostly it is a very thin layer around the NeXus functions but there are a few twists.

All functions using or returning strings must take care to convert F77 strings to C-strings and vice versa. For this support functions are provided.

The NeXus-API stores data in C-storage order but F77 requires fortan storage order. In order to achieve this dimensions have to be reversed. This happens through support functions in napi.c.

The NeXus API requires some structures to be passed around, most notably the NXhandle and the NXlink structures. This is done by copying such items onto F77 arrays large enough to hold the data.

4.2 Scripting Language Bindings through SWIG

Many popular scripting languages have a foreign function call interface. This allows them to interface to user supplied functions written in ANSI-C and packaged as a shared library. There is a tool called Software Wrapper and Interface Generator (SWIG) which takes as input an API description file and the ID of a scripting language and the goes away and generates wrapper code for the scripting languages foreign function call interface. Such an API description file has been generated for the NeXus-API in order to support all scripting languages for which suitable SWIG drivers exist. See the SWIG WWW-site (http://www.swig.org) for details.

Most scripting languages have bad support for multi dimensional arrays. This is why the NeXus SWIG interface supplies its own abstraction for such datasets. This is implemented in nxdataset.h and nxdataset.c. Nxdataset.i is the SWIG interface definition for the dataset API.

The raw NeXus-API proved to be difficult to wrap with SWIG. It became necessary to wrap most NeXus-API functions with a SWIG helper wrapper. Those helpers live in nxinterhelper.h and nxinterhelper.c. The helper functions basically make functions modifying pointers return pointers and data handling functions use the nxdataset abstraction.

The SWIG wrapped NeXus-API functions either return integer error codes or, when pointers are desired, NULL pointers when errors occur. In such cases the text message for the error can be inquired with an additional function: nx_getlasterror.

How the SWIG wrappers for the NeXus-API work is of course slightly different for each scripting language. The description in nxinter.tex for Tcl together with the SWIG documentation for your target scripting language of choice should get you started.

4.3 NeXus-Java Binding

The ANSI-C NeXus-API was wrapped with the Java Native Methods Interface (JNI) for use in Java programs. The Java-NeXus code consists basically of two parts:

- Some Java code implementing a Java interface
- Some C code which translates Java JNI calls into NeXus-API functions and maps return values back to Java values. This code together with the base NeXus-API and the required libraries is used to build a shared library which is loaded into the Java Virtual Machine at runtime prior to the first use of the Java-NeXus bindings.

The Java-NeXus wrapper had to solve a couple of problems:

- Design issues
- Java has no pointers
- Data handling
- Error handling
- Link data handling

In order to go with the Java look and file one might expect that the Java-NeXus binding would consist of Java classes for files, groups, datasets and attributes. After careful consideration this idea was discarded in favour of a plain Java class implementing a NeXus file object. One reason is that such objects are not really idependent objects in a NeXus file but are part of a complex state machine withing the NeXus and HDF-APIs. Keeping such objects in sync with a NeXus file would have caused a nightmare. Instead such a more object oriented Java NeXus-API may be implemented on top of the basic Java-NeXus binding. No one cared enough to do this until now. Thus we are left with a Java interface where we have the NexusFile as the base interface objects. This implements a NexusFileInterface. This layer of abstraction was added to support future NexusFileInterface implementations based perhaps on a networked access to NeXus files. Which never was implemented. But no one complained about this too. Further classes are a helper class for links and an exception class.

The Java language has no pointers but the NeXus-API requires pointers as file handles. This problem was solved through a little dictionary which maps integer handles to real pointers. Java thus only has to deal with the integer handles which get translated into pointers at each call in the JNI-interface. This looks like this:

```
nxhandle =  (NXhandle)HHGetPointer(handle);
```

The dictionary implementation lives in the files handle.h and handle.c. The current implementation supports 8192 files open at the same time only. But this proved enough so far.

The data in the NeXus-API is provided as arrays of native number types. But Java uses network byte order as its own binary representation of numbers in the Java Virtual Machine. A conversion was required. For reasons of laziness and admiration, the conversion routines were copied from the HDF-4 Java API. The conversion code lives in the ncsa/hdf/hdflib directory and its JNI counterpart in hdfnativeImp.c. The conversion happens in the Java code sections acting upon an HDFArray.

The NeXus-API prints errors to stdout. Java handles errors by throwing exceptions all over the place. In order to achieve this the Java-NeXus API replaces the standard NeXus-API error handler with an own one which throws NexusExceptions. This is implemented in the function JapiError in NexusFile.c.

In order to implement linking of objects in NeXus files some information about the objects to link must be carried around. In the NeXus-API this happens in the NXlink structure. This structure is mapped to a NXlink class in the Java-NeXus API. This has to have corresponding fields to the C-language structure. The JNI wrapper copies the data required for this structure back and forth.

Otherwise most of the wrapper routines in NexusFile.c just contain the stuff required to access Java data types from C, invoke the NeXus-API routine, and copy data from C back into Java. See the JNI documentation for details.

5 New NeXus Functions

With all this, extending the NeXus-API with a new function involves a lot of steps. I assume the function requires driver layer functionality, else the first few steps may be disregarded.

1. Implement the new function in each driver layer
2. Add the new function to the NeXusFunction structure
3. Make sure that new function is assigned properly in the driver implementations.
4. Make a new protoype for the function in napi.h
5. Implement the function in napi.c
6. Add the new function to the lists in napif.inc and provide a wrapper in napif.f
7. Create a new SWIG helper function in nxinterhelper.h and nxinterhelper.c
8. Edit nxinter.i to have a SWIG wrapper generated for the new function.
9. Add the new function to the NexusFileInterface and NexusFile classes in the Java-API
10. Write a JNI wrapper in NexusFile.c: take care of the convoluted function names required by JNI and the JNI conventions.
11. Make sure the new function also appears in language bindings not mentioned in this document.

6 Summary

Congratulations! You made it to the end of this boring and lengthy article.