Streaming Data with NeXus

Rationale

Both at synschrotron sources and neutron sources increasingly data is collected in streaming mode. This means that data is streamed from some fast detector together with a timestamp. This data may be correlated in data reduction with other data which also has been streamed. The NeXus file format mainly supports fairly static data collection in a sit-and-count model or a scanning model. Clearly rules are needed to enhance NeXus to support data streaming in a sensible way.

Existing NeXus Structures

Within the NeXus standard two structure exist which are already used for time stamped data. These are NXlog and NXevent_data.

NXlog

This structure looks like this:

```
NXlog
    time[]
       @start
   value[]
```

The interpretation is that the time array contains time stamp values relative to the time given in the start attribute. This structure was designed with things like sample environment and such in mind. There is the drawback that for each entry to append, the start time has to be subtracted. This may cause a performance issue.

NXevent_data

```
NXevent_data
    time_of_flight[i]
    pixel_number[i]
    pulse_time[j]
          @offset
    events_per_pulse[j]
```

This structure is designed for storing neutron event data from a pulsed neutron source or another instrument in time-of-flight (TOF) mode. The events_per_pulse field links the indices j and i. For example finding the events for pulse pulseID involves summing events in events_per_pulse[0-pulseID]. This sum indexes into the time_of_flight and pixel_number arrays from which you then read events_per_pulse[pulseID] events for the pulse.

This structure is very easy to write efficiently:

- Append to time_of_ flight and pixel_number
- Append the pulse_time
- Append the number of events to events_per_pulse

For reading, two cases have to be distinguished:

- You only ever care about the time_of_flight relative to the pulse.

Then you can efficiently read as much of time_of_flight and pixel_number as you can process easily. This is a common use case in traditional neutron scattering data reduction
- You wish to correlate the neutron event data with other data items. Then the effort necessary to reconstruct absolute time from this structure may become a problem.

Streaming Constraints

The point of streaming is often to handle high data rates. Thus everything we do with streaming must be very efficient:

- Writing must be fast enough to keep up with data rates
- Reading must be fast, but is not that critical

Data rates can be so high that streaming to a single conventional disk (~60MB/sec) is not fast enough. Thus we have to account for scaling data writing.

It is out of the domain of a data format but it needs to be mentioned that in order to correlate data from different sources with each other the timestamps of the different sources must be synchronized. To which level of accuracy depends on the application and the speed of the phenomena to be measured. It will almost always be advisable to rely on the timestamps recorded at the source of the data rather then perform the time stamping at the file reader. This is because some delays may have occurred before the data arrives at the file writer.

Data to Stream

It is worth looking at the data which needs to be streamed.

Value Streaming

The simplest form of streaming is a value timestamp pair. This data may come from sample environment or anything else. Such data may come at a high rate.

Another common use case of this is streaming images. Then value becomes a 2D array for each timestamp. Images may also be a streamed with a fixed frame rate. This means that the time interval between consecutive images is fixed. Then the time information is the time of the first frame plus the frame interval.

Pulsed Neutron Event Data

Such data is generated at pulsed neutron sources or at continuous sources when the time of flight (TOF) technique is used through the use of a chopper system. The data comes in pulses as defined by the source or/and the chopper system. The pulse data consists of a list of timestamp/pixelID pairs which describe a neutron being recorded at detector element pixelID at the time stamp specified. These time stamps are relative to the pulse. As a C data structure, the data looks like this:

```
struct {
   unsigned int pulseID;
   unsigned int nEvents;
   double pulseTimeStamp;
   unsigned long pixelID[nEvents];
   unsigned long relative_time[nEvents]
} Pulse;
```

```
struct Pulse data[nPulses];
```

HDF5 has a compound data types which in principle can express the Pulse structure. However, HDF5 compound data types have the limitation that the size of each member is fixed at creation time. As the number of events differs between pulses, HDF5 compound data types cannot be used. We thus have to map the structure above into a set of arrays.

Most data analysis software for TOF instruments only ever cares for the pulse relative time as a new pulse effectively resets time-of-flight. However, for correlating TOF data with other data it must be possible to reconstruct absolute time.

FEL Data

Essentially a FEL is also a pulsed source. Each laser impulse is slightly different and thus it is necessary to store a lot of pulse data in order to perform data analysis.

How Others Do Streaming Data

It is worth fishing for ideas in other streaming applications for concepts and ideas. Some of them may be applicable, some not.

Video

One interesting streaming application is video. One approachable specification is matroska (matroska.org). A number of ideas from matroska might be useful:

- The data is kept in blocks of ~5MB in size
- There is a cueing section which indexes time into the various streams (video, audio, etc)

Another specification I looked at was Quicktime. Variations of Quicktime constitute many more modern video formats. Some notes:

- Quicktime is hierarchical in nature
- Different streams map to different tracks
- Tracks are constituted of a time ordered series of chunks. Where each chunk contains data for some stretch of time.
- There is meta data in the file to locate the tracks and how they are to be processed.

Dynamic Systems Simulations

This is based on a paper from Pfeiffer, Bausch-Gall and Otter titled Proposal for a Standard Time Series File Format in HDF5 . It comes from the dynamic systems simulation community. The authors differentiate between different kinds of time series data:

- *Continuous* where the time data is sampled at fixed intervalls and it is sufficient to store the start time and a fixed intervall for finding time.
- *Discrete* with time data at different time intervalls. Each row has its private time value
- *Fixed* for constants and parameters

The authors store data in different large tables according to time series type and data type. Then they have a group which contains meta data which indexes a stored variable to the object ID of the table in which it is stored and the column index.

Interesting is that the authors tried the NeXus approach with an array per variable also. They learned that this caused performance problems for them because of all the structures and meta data which needed to be created.

TsTables

This is a tool for storing finacial time series data. It is a simple wrapper around PyTables. The only really interesting part about is that it partitions large time series in daily sets. This greatly improves dataset searching and access times compared to the one fat table approach, according to the experience of the author.

LCLS

Apparently, at the Standord FEL, they are using HDF5 as a streaming format too. This format is documented at https://confluence.slac.stanford.edu/display/PSDM/The+contents+of+HDF5+files. Several things are noteworthy about this format.

- They run a deep hierarchy with different groups for calibration, real runs, configuration etc.
- EPICS PV's are stored under their EPICS names. There is no mapping from EPICS names to meaningful names in the context of the instrument.
- In addition to the time stamp they have a *damage and* mask group together with their event data. This tells them if the data is any good or not.
- They also like to divide data into different elements when the stream becomes to long.
- For timestamps they use a home grown compound data type as described below.

The time dataset is actually a structure (HDF5: compound data type) containing the following fields:

- seconds – whole seconds part of the timestamp
- nanoseconds – nanoseconds (range 0–999999999) part of the timestamp
- fiducials – fiducials counter for the event
- ticks – 119MHz counter within the fiducial
- vector – event counter since Configure (not necessarily increasing order)
- control – EVR event code

A Plan for Streaming NeXus

Streaming NeXus File Layout

Streaming NeXus files must be built along the following sets of rules:

- Streamed data items appear at their usual place and under their usual name in the NeXus hierarchy. However, the data field is replaced by a base class of type NXstream with the same name as the data item. The NXstream group then holds the streamed data for the variable.
- The NXdata group contains links to all stream data items. This allows analysis software to figure out easily what was streamed against what.

As an illustration, see the file structure of a streaming NeXus file for an application where a magnetic field at the sample was streamed against detector images. For clarity, only the relevant parts of the hierarchy are shown.

```
entry,NXentry
   example,NXinstrument
       detector,NXdetector
              data,NXstream
   sample,NXsample
        magnetic_field, NXstream
   data,NXdata
       data -> /entry/example/detector/data
       magnetic_field -> /entry/sample/magnetic_field
```

NXstream

For the purpose of the discussion I decided to suggest a new NeXus structure called NXstream. One of the decisions NIAC has to make is if the older NXlog is updated to the functionality of NXstream or if both structures are to be kept.

Another considerations in the design of NXstream is that pulsed data is really a general requirement as the data at FELS also is ordered by pulse. Thus the suggested structure for NXstream also covers the NXevent_data use case.

The next insight is that a pulse ordering is just another form of a cueing structure into the main data arrays. As cueing greatly facilitates random access to time stamped data, NXstream provides for cueing in a general way.

The suggested structure for NXstream is:

```
name,NXstream
  timestamp[nItems]
    @timestamp_type=
  values[nItems,dataDim, dataDim, …]
  pulse_timestamp[nPulses]
  pulse_index[nPulses]
  cue_timestamp[nCues]
  cue_index[nCues]
```

nItems is the number of samples which were streamed.

The structure represents three array pairs:

- timestamp, values
- pulse_timestamp, pulse_index
- cue_timestamp, cue_index

  The first pair contains the data, the two other pairs are cueing structures which index into the first pair.

The pair timestamp, values constitute the main data content of NXstream. This is where the data goes, all the rest is support and optional. There is a one-to one relationship between timestamps and entries in the value field. I.e. values[i] contains the data recorded at timestamp[i].

The values field holds the data collected at the respective time stamp. The first dimension is again nItems. Values may have further dimensions according to the dimensionality of the data collected at each position in time. For example, when collecting images, values will have the dimensions: values[nItems,xdim,ydim].

The timestamp field holds the time data for the values. Timestamps can be in different forms. The default are absolute timestamps in the form described below. If the default is not upheld, then the timestamp field must have a timestamp_type attribute. Currently three timestamp types are suggested:

- **absolute** The default
- **relative** Relative timestamps. This implies another attribute start_time which gives the time value to which the timestamps in the array are relative to.
- **pulse_relative** Timestamps are relative to pulses. This attribute implies the existence of the pulse_timestamp, pulse_index pair.

The array pair pulse_timestamp, pulse_index is an optional cueing pair for supporting data from pulsed sources. Pulse_timestamp is an absolute timestamp for the pulse. Pulse_index contains the index into the timestamp, values pair at which data for this pulse has started to be recorded. This pair supports the FEL and NXevent_data use case.

The array pair cue_timestamp, cue_index is an optional set of arrays for supporting cueing. The cue_timestamp array holds absolute timestamps for cue points. The cue_index array holds indexes into the data when data matching the cue_timestamp was started to be recorded. What the cue_index indexes into depends on the nature of the data:

- If the data is pulsed it indexes into the pulse_timestamp, pulse_index pair.
- If the data is not pulsed, it indexes directly into the timestamp, values pair.

Writing NXstream is straight forward: just append to the arrays as data comes in. Optionally append a cueing entry at sensible time intervalls.

If all you need is to linearly process all collected values then the NXstream structure is a slight overkill. But NXstream shines if you need to select a subset of the data. The only data to load immediately then is the cueing pair: cue_timestamp, cue_index. Then a software can find the interesting time interval in the cue_timestamp array. Should be fast as this is usually sorted. The cue_indices then provide the subset of the timestamp, values pair to load. With TOF data there would be the additional layer of indirection through the pulse array pair.

Timestamps

There are timestamps which are nice for humans, mostly stored as text, and timestamps nice for computers, stored as numbers. As NeXus is concerned with the processing of large amounts of data, the preference will be numeric time stamp data. Text timestamps take to much time to parse into numbers.

For absolute time stamps I suggest to use:

```
double DoubleTime(void)
{
  struct timeval now;
  /* the resolution of this function is usec, if the
machine supports this
     and the mantissa of a double is 51 bits or more
(31 bits for seconds
     and 20 for microseconds)
   */
  gettimeofday(&now, NULL);
  return now.tv_sec + now.tv_usec / 1e6;
}
```

which gives us time in UTC. NeXus wants UTC, otherwise we might see time jumps when switching back and forth for daylight savings time.

This requires timestamps to be float or double types. If we insist on integer types we have the option to go fixed point. I.e. we multiply with 1000 or 10000 until we get the required resolution and cut the remainder off. In this case the timestamp field should have a divisor attribute which tells us how to get at the real value. But the NIAC has to decide if it even considers this.

NeXus will need relative timestamps too. These are offsets to the absolute timestamp given as the start attribute or relative to the pulse. We may need to support non standard time ticks. To this purpose I suggest a new attribute *time_unit* which specifies the length of an integer tick in milliseconds.

Scaling NeXus Data Streaming

The expected use case of NeXus data streaming is fast detectors. So we have to make streaming go fast.

Parallel Writing

The first option is to use parallel HDF5 on top of a parallel file system. This is actually the preferred option. Multiple sources/processes can write to the same HDF5 file in parallel. This HDF5 feature is called collective I/O. This must be used in the following way:

1. A master process creates the necessary HDF5 structures
2. The master signals all the slaves who then write data to the respective HDF5 datasets they serve.

Static metadata is either written by the master when generating the structure or added after collective I/O has finished to that data file.

Multiple Files

Though parallel writing using the proper infrastructure is the preferred solution, NeXus will have to cater for the situation when adequate IT-infrastructure is not available. It happens that people get the money to buy the fast detector (cool) but not the necessary IT-infrastructure (uncool) to actually use it. Still, there is the option to use multiple files.

When this option is used, then there is a master file which contains most of the structure and links other files containing streamed data via external linking. Then multiple processes, possibly on multiple machines, can stream data to individual files and for the user it still looks like streaming to a single file. Or the files can be merged after writing finishes using h5repack.

Stream Segmentation

The results from the streaming survey suggests that segmenting streamed data into manageable chunks is a preferred option even for projects using HDF5. The experience is that even HDF5 gives better performance when arrays do not grow to big. Thus I suggest that NeXus considers a means of segmenting streamed data too.

If we accept segmenting streamed data as a requirement we have options how to do this:

It can be done within a group with separate datasets and a naming scheme. This could look like:

```
name,NXstream
     timestamps_000[]
     values_000[]
     timestamps_001[]
     values_001[]
```

```
      ….
      timestamps_077[]
      values_077[]
```

Or data segmentation can be implemented through separate groups. This could look like:

```
   data_000,NXstream
   data_001,NXstream
   data_002,NXstream
   ….
   data_077,NXstream
```

In both cases, a naming scheme for associating related data is necessary. Implicitly I suggested appending a order number to the names in the examples.

If we decide to go for separate groups there is also the option to get away without an naming convention by defining yet another NeXus structure and introducing yet another level of indirection. This could be a group with the suggested name **NXstreamcollection** which would appear in place of NXstream and hold all NXstream groups belonging to the data item replaced by NXstream.

Then a file structure could look like:

```
entry,NXentry
    example,NXinstrument
        detector,NXdetector
               **data,NXstreamcollection***
                  data_1,NXstream
                  data_2,NXstream
                  ….
```

```
sample,NXsample
     magnetic_field, NXstream
data,NXdata
    data -> /entry/example/detector/data
    magnetic_field -> /entry/sample/magnetic_field
```

The NXstreamcollection group could hold an optional pair of **cue_timestamp, cue_name** arrays. This pair indexes which time range is stored in which data segment. Cue_timestamp holds the start_time for each segment, cure_name holds the name of the segment. The cue_timestamp, cue_name pair must be optional as it is conceivable that different data segments are not ordered by time.

It needs to be pointed out that the segmentation of datasets can also help with performance issues. This by writing to dataset segments in separate files and separate processes. In the pulsed neutron data use case it may make sense to write different pulses into different segments (possibly residing in different files) in order to get the required performance.

Things for the NIAC to consider

1. Is something missing in this white paper? Especially for storing FEL data?
2. Is everyone happy with the general layout of NeXus streaming files?
3. Do we accept NXstream or merge it into NXlog?
4. Do we accept NXstream also for neutron event data from pulsed sources or do we want a separate structure?
5. Does NIAC endorse the cueing array pair?
6. Is everyone happy with the suggested scheme for timestamps?
7. Do we consider dataset segmentation?

```
1. Do we segment by groups or by datasets?
```

2. Do we provide a naming scheme for segmentation?
3. Is NXstreamcollection a viable and desirable option?
4. If NIAC accepts NXstreamcollection, do we accept the cue_timestamp, cue_name pair?