Submitted Comments on Neuroshare API Specification Rev 0.9a

April 22, 2002

Comments regarding rev 0.9a are reproduced here. They have been listed in the order they were received. We have responded to them with our rationale and welcome feedback and further suggestions.

Contributors:
AR – Anthony Reina
EB – Edward Peterlin, R&D, BIOPAC System
SG – Shane Guillory
TB – Tim Bergel

1). Library function returns a uint32. Is this overkill? It seems like a uint8 would be fine. I don't think we'll have more than 256 different return flags. Again, uint32 for the *hFile. In ns_FILEINFO structure the members wTime_Month, DayofWeek, Time_Day, Time_Hour, Time_Min, Time_Sec. all could be uint8. Again, uint32 for dwEntityType could be uint8, etc. <AR>

> *We have decided to use uint32 for all library function return codes, dwEntityType, dwEventType, dwSourceID, nFlag, and all time and date members of ns_FILEINFO and for file handles hFile. This is to take advantage of the natural data alignment in 32-bit systems. Function arguments are aligned on 4-byte (word) boundaries and thus passing arguments of smaller size does not offer any savings. The time and date variables of the data structure will also be defined as uint32 for consistency.*

2). p. 12. - The variable is listed as TimeRange, however it is only one value. I assume that the beginning of the file is time 0 and that this number would define the maximum time in the file. Could this be re-named dTimeMax? Or, could this be a double[2] with the option for the first index being the minimum time and the second being the maximum time? <AR>

> *We will rename it dTimeSpan to indicate that it covers one interval of time, the time span of the file. The file starts at time 0 seconds and dTimeSpan specifies the time in seconds at the end of the file.*

3). I would recommend providing either an explicit callback or an associated XML file to convey information about the neuroshare API a plugin understands and its corresponding file types. Encoding them within DLL header info is inherently not cross platform friendly. <EP>

> *Our intent was to be able to obtain library information without actually loading the library. We thought this would help with potential version problems and that*

it would save resources allocated by unused, but loaded, libraries. The Win32 DLL version query on p. 33 is an example of how it could be done in Windows. We assumed that a similar query mechanism could be defined for Mac and Unix/Linux platforms in the future.

We are not opposed to an associated file providing library version information to go along with each API library, but it would be nice for each library to be a stand-alone file.

As a compromise, what about providing a GetLibraryInfo function that reports all of the title, version, and preferred filetype information. We then also request that libraries use discipline and not allocate large amounts of resources until they are called upon to open files. If having unused libraries open is a problem for applications, they can simply unload unused libraries after the "Open File" dialog box closes.

4). If the sizes of the data structures are encoded in the structures themselves instead of additional parameters, parsing of the size can be used to detect endianness. <EP>

When we debated back and forth on this, we couldn't really come up with a good reason to put the size in the data structure. These structures are most likely to be delcared locally in functions when used and it seemed that putting the size in the structure would require an extra-line of code to initilize this member as opposed to just putting the value in the function call. We are completely willing to revise this if the consensus is that it is a good idea and necessary for endianness checking. However, won't most libraries "know" if they are being compiled for little or big endian systems?

5). The alignment for the structures should be specified or padding should be explicitly inserted. The location of members of the structures that occur in sequences like uint32, char[16], double will depend on whether the data members need to be 2 or four byte aligned. <EP>

Alignment for data structure members will be explicitly declared as 4-byte and all data structure members in the definition will be defined to require no padding for this alignment. This will be added to the document.

6). We program in threaded environments here, but there is no mention of thread-safety in the API. It should either be stated or allowed for each individual plugin to advertise whether or not it is safe to call the plugin reentrantly. Interacting with large files can easily be a critical I/O bound task within a program and a prime candidate for being performed in parallel. If a plugin is not reentrant, the API should explicitly require the host application to not call the plugin reentrantly. <EP>

Excellent point. We will add a section to the specification to discuss this important issue. We should also pose this question to the working group. Should libraries be required to be safe for multiple threads, or should each library be able to report whether or not it is mult-thread safe and leave it to the applications to execute functions appropriately?

7). Not all environments may use extensions to identify file types. For example, classic MacOS (one of our target environments) uses two four-byte identifiers to encode file type and creator. If a platform independent method of reporting plugin information is developed, it should be flexible enough to incorporate non-extension based information. <EP>

Noted. Even on PC platforms, extensions are not a safe method of determining type. We were expecting applications to use file extensions as suggestions, not requirement for type. We are expecting applications to allow the user to select the intended library and as well as files and let the library report an error if it cannot open the file or file group specified. We will clarify this in the standard.

8). It would be good to provide a function to obtain a string representation of the last error. This can be a big help in UI design, allowing a plugin to report more detailed error information than encoded in the return types. An example of this would be a plugin that can't open a file due to the file's own file format being too old. We already have approximately 20-30 revisions of our own file format, so this type error could easily crop up for us. The function should also take an optional hFile parameter to allow the plugin to distinguish errors based on multiple open files, if desired. <EP>

We were trying to keep the return codes somewhat limited in scope to encourage people to actually use them in applications and libraries. We were expecting to add a few more codes in the future, but the potential errors are fairly straight forward for the defined function set.

However, it would be potentially useful for a library to be able to pass an extended error message directly to the user in string form. Our inclination would be to add it to the standard later if the set of error constants proves to be inadequate.

Comments from the Working Group?

9). Answers to specific questions posed when the standard was distributed: <TB>

A) ns_ is fine as a prefix, though I am used to 3-char upper case prefixes such as NSH, so ns_OpenFile would be NSHOpenFile. Doesn't really matter what you use does it?

We found that we liked the lowercase prefix because it's easier to read. For example, when you read `nsOpenFile()` *or* `ns_OpenFile()` *it seems easier to read "OpenFile" rather than* `NSOpenFile()` *or* `NSHOpenFile()`. *Additional comments from the working group? Does anyone know of established standards besides Hungarian Notation for this type of thing?*

B) Each entity having indexed items seems natural to me.

C) I think that the level of Hungarian notation that you are using in the 0.9a spec is fine, it makes the structures readable without getting too fussy.

D) We should use enumerations from 0 to number-1 throughout as far as I am concerned. Stuff MATLAB.

In retrospect, when we write the matlab function set for using libraries, it would be reasonable to have the matlab functions renumerate the entity and index lists. Starting at 0 definitely seems more natural for C programming.

E) Passing the size as a separate parameter works for me. We need to specify that a) future API versions will only increase the size and b) new items will only be added to the end to provide for minimal forwards-compatibility. If we can also specify that a default value of zero in all new data added to these structures is acceptable, then older DLLs should be able to emulate a newer API version quite well....

Yes, that should be explicitly stated. < Please see issue 4 on the previous page. >

F) Yes, give the time range for each entity I think.

We agree. It wouldn't be hard for most libraries to provide that information, or if they don't want to, to simply provide 0 and the timespan of the file for the range.

G). I'm not quite sure what you mean by this, but it sounds as if, if my file contains classified segment entities, I have to return the classifications as simulated neural entities. I don't want to have to do this (almost cannot) and I thought we agreed that current classification info would be returned with the segment entity data (indeed I recall going on about it at some length). Or have I misunderstood totally? In any case I go on about segment entities a bit more below...

Ok, you caught us on that one. We were flipping back and forth on including the classification codes. It definitely makes sense to include the classification code in segments, especially for spike sorter applications, and we'll add it back.

The problem we keep coming back to is that analysis applications that only care about spike times will have to look for both classified segments AND neural events to get spike trains. Is it reasonable to require libraries to also present spike data classification codes as neural event entities? There are several reasons why this would be quite useful:

a) Easy for Analysis Apps – Analysis applications that use spike timing only need to look at the neural event entities and the spike timing data can always be found there regardless of other data contained in the file.

b) Timing within Segments – Many groups reference the first peak or other waveform trigger as the "time" of the spike. These are not necessarily a fixed interval from the beginning of the segments depending on how the segments were thresholded. If libraries are savvy about this, they can export the data directly as a neural event with correct timing rather than a segment that also has an additional value returned for "time to the spike within the segment."

c) *Superpositions or multiple spikes in a single segment – This is where things really get ugly. When we go to include a classification code for a segment, should there only be one allowed unit per segment? Should there be unit 1 to 32 plus a flag if superpositions are also detected in the segment? Should there be a variable length structure which lists unit ID's and times within the segment that each unit waveform is found?*

There are a lot of ways that we could handle the single and multi unit timing issues by adding more information to segment functions, but it seemed that this would make things increasingly difficult for applications to simply get neural spike timing out of segments. Establishing a convention of exporting all classified units as neural entities (in addition to segments) would make it easier for the user applications. This is a really important issue that we need feedback from the working group on.

Another problem is that we are really not providing lots of support for reporting the classification methods that were used on the data. For example, it would be nice for offline sorters to be able to retrieve the sorting templates that were used on the data file before they attempt to re-sort the data. This will probably have to be addressed in the next revision.

10). Page 7 - Where you describe analogue entities, I think you should mention both their equally-spaced in time nature, and more crucially that there can be gaps in the data, and the way that these appear. <TB>

Yes. Also see comment 16 below.

11). Page 8 -You definitely need a ns_GetLibraryInfo(&dwVersion, &szLibName) function. <TB>

Originally, we were trying to make the version info available without loading the library. However, there have been multiple complaints on this. We will add a GetLibraryInfo function. Please see the discussion in (3).

12). Page 9 - Is there are reason for giving each error type a separate bit? I would suggest negative numbers (so ns_RETURN becomes int32) starting at -somelargevalue (to keep out f the way of Microsoft codes). This way we can have warning codes that are +ve but not zero if we need to. Not important though.
What do people think? <TB>

Not a bad suggestion. We had originally put it in as separate bit codes to allow multiple simultaneous errors, but that's really not particularly useful or necessary. Unless anyone objects, we will make that change.

13). Page 10 - The remarks should state that hFile is not (necessarily) a file handle as used by the underlying OS and can only be used with the ns_ functions in the DLL that opened the file. <TB>

Yes, this will be clarified in the specifications.

14). Page 15 - >From a purist POV, we should enumerate entity types from zero as well, so ENTITY_EVENT should be 0. Not important. <TB>

> *We were thinking that zero had a bit of a null, undefined, or default feeling to it. As these are defined types, not array indexes, it made sense to start with 1. (We'll update the event type enumerations for consistency) In fact, maybe it is worth having an ENTITY_UNKOWN to report that something is in the file that is not abstractable by the library (except by name).*

15). Page 18 - I have a great deal of trouble with GetEventData. It is fine as it stands, but we <really> need functions that return an array of data for any sort of efficiency (the same only more so for segments). I know that, because both of these data types are of variable size, it will take a bit of heavy coding to handle them, but that should be a price we are willing to pay. I am more than willing to suggest suitable designs. For example for event entities: <TB>

a) All of the items will be rounded-up in size to the max possible, so all items from one channel are the same size.

b) We define a structure to hold the data type which includes size information:

```
typedef struct textEvent
{
  double  dTimeStamp;
  int32  nChars; // We don't really need to know this as null terminated...
  char  szText[16384];
}TTextEvent;
```

note that all this is much easier with the non-text types of events, which are of fixed-size.
c) We can access the text event data, even though its size is not known at compile-time, in a void array of some known size by using pointers and a bit of care:

```
void* pBuf; // We assume this is pointing at lots of events
void* pEnd; // Points to first byte in pBuf past the end of the data,
// ie (pBuf + bytes read)
int nSize = sizeof(double)+sizeof(int32)+(MaxCharsForEntity);
// Size at max in bytes
TTextEvent* pTE = (TTextEvent*)pBuf;
while (pTE < pEnd)
{
... Here we do something with the first event
 printf("At %g seconds: %s\n", pTE->dTimeStamp, pTE->szText);
pTE = (TTextEvent*)(((char*)pTE)+nSize);      // Move pointer to next event
}
```

Did this all make sense? I hope so because I think array access is a must. I would guess that we should retain the functions to only read a single item for ease of use.

<reply to comment on previous page> Hmmmmm, there are several issues here. It is unfortunately only the variable-length nature of the types that makes them difficult to process in this way. The best way to do this would seem to be a function of where the bottlenecks in the system are.

For example, if a library internally builds a table of segment file locations when a data file is opened, then the bottleneck would be the file access to retrieve each segment from the data file, not the library function call. In this case, it would be better to have a single item function call for simplicity.

However, if the library caches certain information, such as event entity items, it would be a bit faster to return arrays of entity items.

Another way of addressing the variable length problem is to pack the data structures returned in block array accesses (which have some type of member that conveys the length within each structure) and access them sequentially. This almost seems as ugly as having large fixed-length data structures that may not contain any data.

What about starting with the functions as defined, and then adding "block" functions later for segments and events if needed? The current functions might not be that big of a efficiency problem for most users and vendors since we have library functions for inventorying and searching indexed items by time.

16). Page 21 - Reading analogue data will be more complicated because of possible gaps, you will need another parameter updated with the count of items actually read, or some such. The point is that, if you try to read data indexes that 'jump' a gap, only data up to the start of the gap can be returned, so if you got less than you asked for it's because you got to a gap. <TB>
I think this should be discussed in some detail at this point in the spec so that people get the point.

Ooops. Yes, the gap issue was neglected. When we were drafting the document, we made analog data access to be index-based rather than time based. This makes it consistent with the other GetData functions. Each sample of the analog entity has an index.

*From this perspective, if gaps are present, they don't interfere or alter the way that the function would work, but the user would have to test that the time interval between the first and last index would be equal to (lastidx-firstidx)*SamplePeriod to be assured that there were no gaps. This seems awkward.*

What about adding a flag to the AnalogInfo structure that reports whether or not there are gaps present in the analog entity and adding a pGapIndex variable to the GetAnalogData function. When you get data for a range of analog indexes, pGapIndex would be filled with zero if there were not gaps in the requested range of indexes. If there are gaps in the requested range, pGapIndex would be filled in with the first index following the first gap in the requested index range. In this way, pGapIndex could provide both a test for gaps and a way of efficiently moving through them.

17). Page 23 - A segment needs a time offset from the time stamp to the time of the first point in the first source (or something). Segment data commonly has a time stamp for the peak in the data, not the start. <TB>

[we assumed that "first point" meant first peak in a spike waveform] A proposal to handle the time to the first spike in a segment was addressed in Comment 9-G.

18). Page 25 - Maybe the segment sources should be arranged so that the dSubSampleShift for the first source is always zero? <TB>

Not necessarily. If the first source is out of phase with the system clock (as it can be in some systems), this would cause some awkward situations. Suppose the system clock resolution is 25us and the sub-sample lag of channel 1 is 5us. This would mean that segments with channel 1 as source 1 could come back with times that aren't multiples of 25us. It seems cleaner to us to have the segment be time-stamped with the "system clock" and report the offset of source 1 as 5us. Comments?

19). Page 26 - As I said above we need to be able to read multiple segment items at once and I thought that a segment would include a classification code. The classification code is no problem technically but may need discussion, but again I am sure that reading lots of items at once is important, even if it does produce code that is a bit tortuous at times. The code techniques described above will handle arrays of segment items just as well as text events. <TB>

See reply to comment 15.

20). Page 30 - ns_AFTER should give the first index at or after dTime I think, to keep things clean. To find the first index after dTime you use ns_AFTER and add the time stamp resolution value to dTime. <TB>

We figured that most of the time, users will use the indexes directly to step through entity entries in time. To get the next or previous index in time, you simply increment or decrement the index numter. The only time that GetIndexFromTime would be used would be for "out of the blue" searches such as getting the range of indexes that fall within a user-controlled display window. From this perspective, you are right in that most people will probably want to do inclusive ("<= dTime" or ">=dTime") searches, but it should probably apply to both the BEFORE and AFTER cases. If people wanted to do non-inclusive searches, they could add or subract an arbitrarily small value to/from the search time. The CLOSEST option would probably still be a handy option for and since time is a double, it can cope with "0.99999999 != 1.0" issues.

Alternatively, you could provide an additional parameter to specify whether or not the search is meant to be inclusive. You could also more search options such as LT, LE, CLOSEST, GE, GT, but these feel inelegant. *<continued, next page>*

What about starting with making BEFORE and AFTER perform inclusive searches and adding non-inclusive options in the future if necessary?

21). Page 32 - I am generally opposed to putting the neuroshare DLLs in a central place or indeed specifying how they are to be installed in any way. If they have to be central then different packages can interfere with each other in <fascinating> ways - ask any programmer about ComCtl32.dll and the like. I am particularly unhappy about anything inside \WINNT - that is absolutely Microsoft's bailiwick. Just say that it is each application's responsibility to know where to look for its DLLs and to provide instructions to the user how to add a new DLL? <TB>

The only time this is a problem is when you have multiple applications that need to use a common set of libraries. Having a known, dedicated directory such as \WINNT\Neuroshare or \Program Files\Neuroshare (and the Mac equivalents) means that there is only one place to look if want to add, remove, or update a neuroshare library. Without a common place, you have to keep a copy for each app or require the apps to be sophisticated enough to keep track of their location.

This also raises the issue of conventions for version management. For example, if company X has multiple versions of the library for their Y format, should they name each library differently (ie, append the version number to the filename)? This would make it easy to keep multiple versions as compatibility problems are discovered. In a general sense, the name of the library is irrelevant as it will report it's information and file affinities in other ways. However, if different versions do not have different names, there must be conventions for keeping multiple copies in different locations.

As a compromise, what about having each application check it's own preferred directory first, and if no libraries are found, it checks a common directory? What should the common directory be for Windows? Mac? Linux? We also propose that encouraging, but not requiring, different names for different versions of the same library would be a good idea.

These are good issues for comments from the general working group. Comments?

22) I didn't think we should concern ourselves with DLL resources at all, but I guess the stuff about language neutral Unicode is relevant enough. Everything that we can handle with specified ns_ functions should be handled in this way. <TB>

See 3) and 11) for the rationale of using this method for getting library information.

Additional Question to Ponder - Should we place some sort of “reasonable” limit to the size of event entity and segment entity items? Does 1024 bytes seem big enough for event entities and 1024 samples for segment entities? <SG>