MATLAB Native Client API and Sample Clients for OmniPlex

Version 1.0
11/20/2018

Introduction

The OmniPlex Native Client API (NCA) is a set of MATLAB callable functions that allow user programs to read live data from a Plexon OmniPlex (OPX) data acquisition system, including timestamped spike, continuous, and digital event data.

Previously, OmniPlex clients read online data via a MAP (Harvey Box) compatible API, but this legacy API did not provide access to OmniPlex-specific information such as source descriptions, and contained MAP-specific functions that were not applicable to OmniPlex systems. The NCA is streamlined and optimized for use with OmniPlex systems and cannot be used with MAP systems. Clients can continue to use the legacy API with both OmniPlex and MAP systems, but legacy and NCA functions cannot both be used within the same client program.

The NCA SDK contains a set of sample client programs that demonstrate how to read spike, continuous and digital event data from OmniPlex. Each client is a single MATLAB script (.m file).

The first sample client, TimestampClient, only reads spike and digital event timestamps, not spike waveforms or continuous data. It is the client most similar to a legacy API client, in that OmniPlex sources are not used, and it is close to the minimum amount of code needed to read data from OmniPlex. TimestampClient prints out the spike and event timestamps to the MATLAB command window, and does not display any graphics.

The second sample client, SpikeWaveformClient, reads and draws spike waveforms from a single channel.

The third client, ContinuousDataClient, shows how to display continuous data from a single field potential (FP) channel, although the same techniques can be used to draw the signal from any continuous channel.

The fourth client, MultiSourceClient, shows how to read and display data from multiple OmniPlex sources, such as spikes (SPK), spike continuous (SPKC), wideband (WB) and field potential (FP).

The fifth client, InfoAndConversionsClient, does not read spike, continuous, or event data from OmniPlex, but instead shows how to use NCA functions to query information on all the sources in an OmniPlex system, such as sampling rates, channel names and spike waveform lengths, and how to convert from linear to source-relative channel numbers and vice versa.

The sixth sample client, LowLatencyClient, shows how to wait on the Server synchronization event in order to minimize the latency (delay) from data acquisition, which can be important for closed-loop applications such as neural prosthetics.

Each of the sample clients are described in more detail in later sections. It is recommended that you read about the clients in the order provided, since the descriptions of the later clients do not repeat material that was covered in previous sections.

The style of the sample clients has been kept simple, and only basic MATLAB statements and functions are used. The emphasis (except in InfoAndConversionsClient) is on showing how to use a core set of API functions which are most commonly used in clients.

To write your own client programs, you can start with one of the provided sample clients and extend and modify it to suit your needs (Plexon users have written some very complex clients this way), or you can start from scratch and merely use the samples as a reference.

Unpacking and running the sample client programs

The sample clients and the SDK support files required to run them are contained in the MATLAB online client SDK zip file, named `MatlabClientDevelopKit.zip`, within the folder `MatlabOPXClientSDK`. The contents of the zip file are organized as follows:

```
MatlabClientDevelopKit
  ClientSDK
  MatlabOPXClientSDK
    code
    doc
  PlexDoSDK
```

The `code` folder contains the .m files for the sample client scripts and the NCA functions which they call, with each NCA function as a separate .m file. For example, `TimestampClient.m` is the first sample client and `OPX_InitClient.m` contains the NCA function `OPX_InitClient` which is called by all clients. In addition, the files `mexOPXClient.mexw64` and `PlexClient.dll` contain the bulk of the code that implements the NCA; the `OPX_*.m` functions are very simple interfaces between client programs and these two files.

Other top-level folders within the zip file are `ClientSDK`, containing the legacy "Harvey Box" version of the online client SDK, and `PlexDoSDK`, an auxiliary SDK for controlling digital output devices, which may be used with either the new or the legacy client SDKs. All new client development for OmniPlex systems should use `MatlabOPXClientSDK`.

Note that the `PlexClient.dll` which is provided with the OmniPlex MATLAB SDK is a 64 bit version which must be present in the same directory as `mexOPXClient.mexw64` and your clients; MATLAB clients should use only this version of `PlexClient.dll`, and *not* the version which is installed elsewhere as part of an OmniPlex installation.

Detailed documentation of each NCA function, including the parameters passed to and returned by each function, is contained in the individual .m files in the form of comments. For the simpler NCA functions, such as `OPX_CloseClient`, this is brief and fairly self-explanatory. For other functions with multiple return values such as `OPX_GetNewData`, the comments are more

elaborate and are an important reference for understanding the usage of the functions, in combination with the sample client code.

There are a number of NCA functions which are not used in the sample clients, and once you become familiar with the basics of client programming, you may wish to examine the .m files for these functions to learn about the additional or alternate functionality they provide. However, the functions used in the sample clients are sufficient for the majority of client programming.

Note that the clients use *assert* statements to check error return codes from NCA functions. The sample code usually only asserts that the return value is 0, i.e. no error. In a real client, you would typically use a conditional statement to check the error code and take appropriate action, such as displaying an error message to the user. Most error codes are related to invalid parameter values being passed to an NCA function, such as a non-existent source number or an out of range channel number. A list of the error codes, plus other predefined constant values used by NCA functions, can be found in `OPXConstants.m`. It is a matter of judgment how much error checking should be done in a "production," i.e. debugged and tested, client script. In general you should include error checking, unless it causes an unacceptable performance penalty.

TimestampClient

TimestampClient reads spike and digital event timestamps (but not spike waveforms or continuous data) from OmniPlex Server. A simplified pseudocode outline of TimestampClient might be:

```
wait for data acquisition to start in OmniPlex
call OPX_InitClient to initialize the connection to OmniPlex Server
loop
{
  call OPX_GetNewTimestamps to read new data
  print out the data
  sleep for 100 milliseconds
}
until data acquisition stops in OmniPlex
call OPX_CloseClient to close the connection to OmniPlex
```

The other sample clients use variations of this basic structure.

You may wish to read along in the code for TimestampClient for the following details.

First, a few constant values for status and error codes are defined, such as `OPX_ERROR_TIMEOUT`. These definitions are taken from the file `OPXConstants.m`.

The first NCA function called is `OPX_InitClient`, which connects the client to the online data from Server. It also specifies the data format as `OPX_CHANNEL_FORMAT_LINEAR`. This is a format where spike channel numbers are in a single linear range from 1 to the highest spike channel and event channel numbers are in a single linear range of 1 the highest event channel. This linear format was used in the legacy client API and in PLX data files. TimestampClient uses this

format both for simplicity, and because it is adequate when only spike and event timestamps are being read. However, when continuous data is being read, as in some of the other sample clients, then OmniPlex sources and source-relative channel format are preferred. The differences between linear and source-relative channel numbering are described in the later section

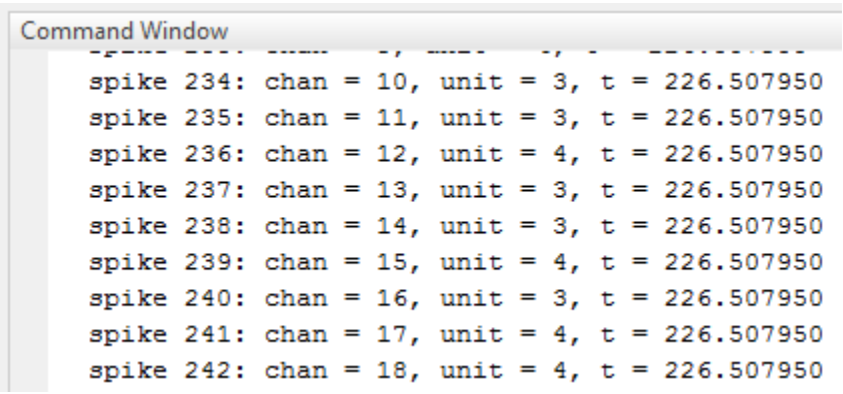
OmniPlex sources and channel numbering.

After calling `OPX_InitClient` to connect to Server, TimestampClient calls `OPX_WaitForOPXDAQReady`, which will wait for up to the specified amount of time for data acquisition to start in OmniPlex, in this case 60 seconds. If data acquisition is already running, it continues immediately.

Once data acquisition is running, `OPX_ClearData` is called; this function discards all available data, until either no more data is immediately available, or the specified timeout elapses. You might wonder why throwing away data would be desirable. The idea is that, especially at high channel counts, or if a large amount of data is being buffered in OmniPlex Server, when a client starts reading data, if OmniPlex has *not* just started running, the client may be overwhelmed by a large initial "backlog" of old data, whereas clients are typically only interested in the new data that arrives after they have started running.

Finally, execution enters the main loop. In each pass through the loop, we call `OPX_GetNewTimestamps`, which reads timestamped data from Server and sets `numSpikes and numEvents` to the actual number of spike and digital events timestamps returned. Spikes are returned in the variables `spikeHeaders` and `spikeTimes`, where `spikeHeaders(i)` and `spikeTimes(i)` together represent a single spike. Likewise, events are returned in the variables `events` and `eventTimes`, where `events(i)` and `eventTimes(i)` together represent one event.

After the latest batch of live data has been returned by `OPX_GetNewTimestamps`, TimestampClient prints to the command window the count of spikes, followed by per-spike information, and the count of events, followed by per-event information. The per-spike information includes the channel number, unit, and timestamp; the per-event information included the channel number, event data, and timestamp. Note that only multi-bit (or "strobed") events (linear channel 257) have valid event data.



After reading and printing each batch of new data, TimestampClient pauses for 100 milliseconds before returning to the top of the loop. However, there are two cases where the 100 ms pause is *not* done. First, if the return value from `GetNewTimestamps` indicates that more data was available than could be read, execution continues without pausing, in order to keep up with the incoming data. Second, if a call to `OPX_GetSystemStatus` indicates that data acquisition has been stopped in OmniPlex. All of the sample clients terminate either when data acquisition stops in OmniPlex, or you use Ctrl-C in MATLAB to stop the client. If the client is terminated "cleanly," i.e. not by Ctrl-C, it calls `OPX_CloseClient` to disconnect itself from OmniPlex Server before exiting.

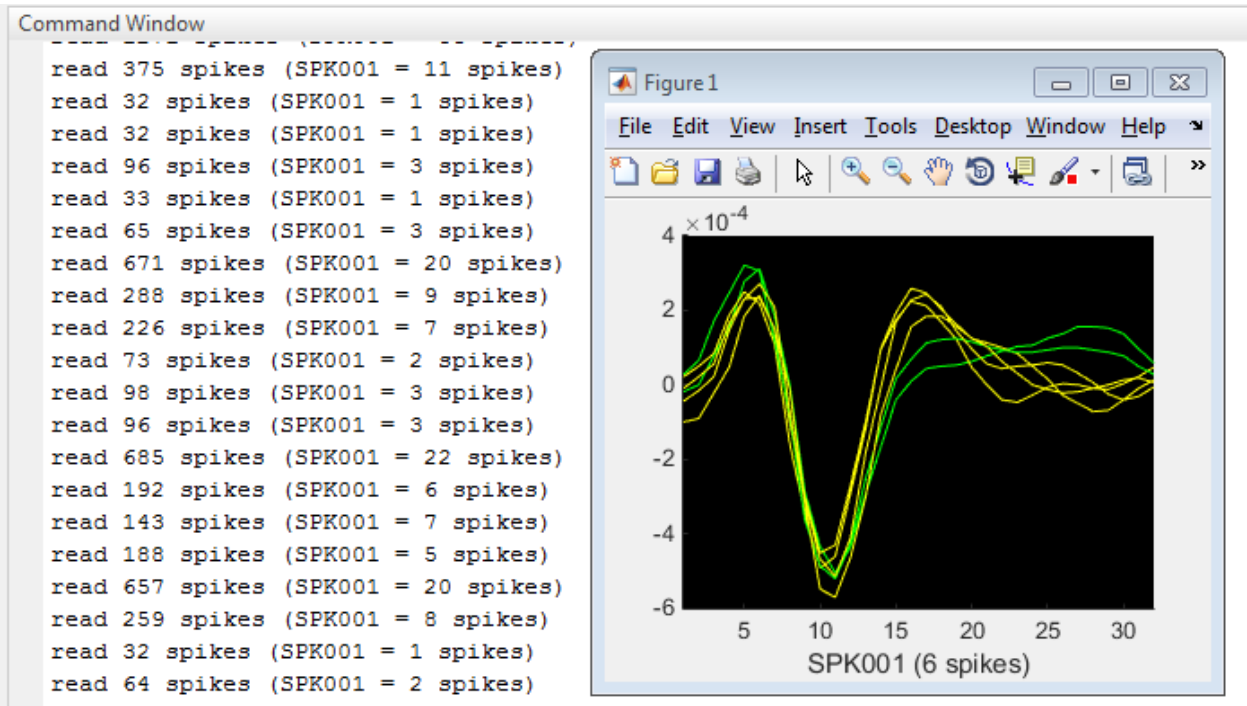
SpikeWaveformClient

SpikeWaveformClient displays the spike waveforms for a designated spike channel (determined by the variable `chanToDraw`, by default channel 1, SPK001) in a plot window. The differences relative to TimestampClient are described below.

Whereas TimestampClient called `OPX_GetTimestamps` to read online data, SpikeWaveformClient calls `OPX_GetNewData`, a more general function which can return spike waveforms and continuous data, in addition to the timestamps, channel numbers, and unit numbers returned by `OPX_GetTimestamps`. In SpikeWaveformClient, we only use the spike-related return values: `numSpikes, spikeHeaders, spikeTimes and spikeWaveforms`. In addition to the spike channel and unit numbers, we also use the number of points per spike and the spike waveform points to draw the spikes. The unit number is used to index into an array of colors so that each spike is drawn in the default unit colors used in PlexControl.

Note that in SpikeWaveformClient, although the amplitude (y) scaling is correct (in volts), for the x axis we simply use the waveform sample point index for the x scale, e.g. 1 to 32 for the default 32 point waveform length. In effect, the timestamps are ignored for the purposes of display, and all the waveforms on the channel are drawn on top of each other, as in the current-channel spike display in PlexControl. The display is erased for each new batch of spikes read by `OPX_GetNewData`.

After a batch of spikes is drawn, the plot is labeled and we print out a line in the command window, showing the total number of spikes on all channels in this batch, and the number that were drawn for the selected channel.



Before describing the ContinuousDataClient and MultiSourceClient programs, it will be helpful to review some information about OmniPlex sources.

OmniPlex sources and channel numbering

Sources

An OmniPlex source is basically a contiguous range of channels on a specific hardware or software device. For example, the WB source is the set of full-bandwidth channels output from your system's main neural data acquisition device (a DHP, DigiAmp/MiniDigi, or AD64) and the FP source is the set of channels output from the FP Separator device (a software device which performs digital filtering, noise removal and sample rate reduction).

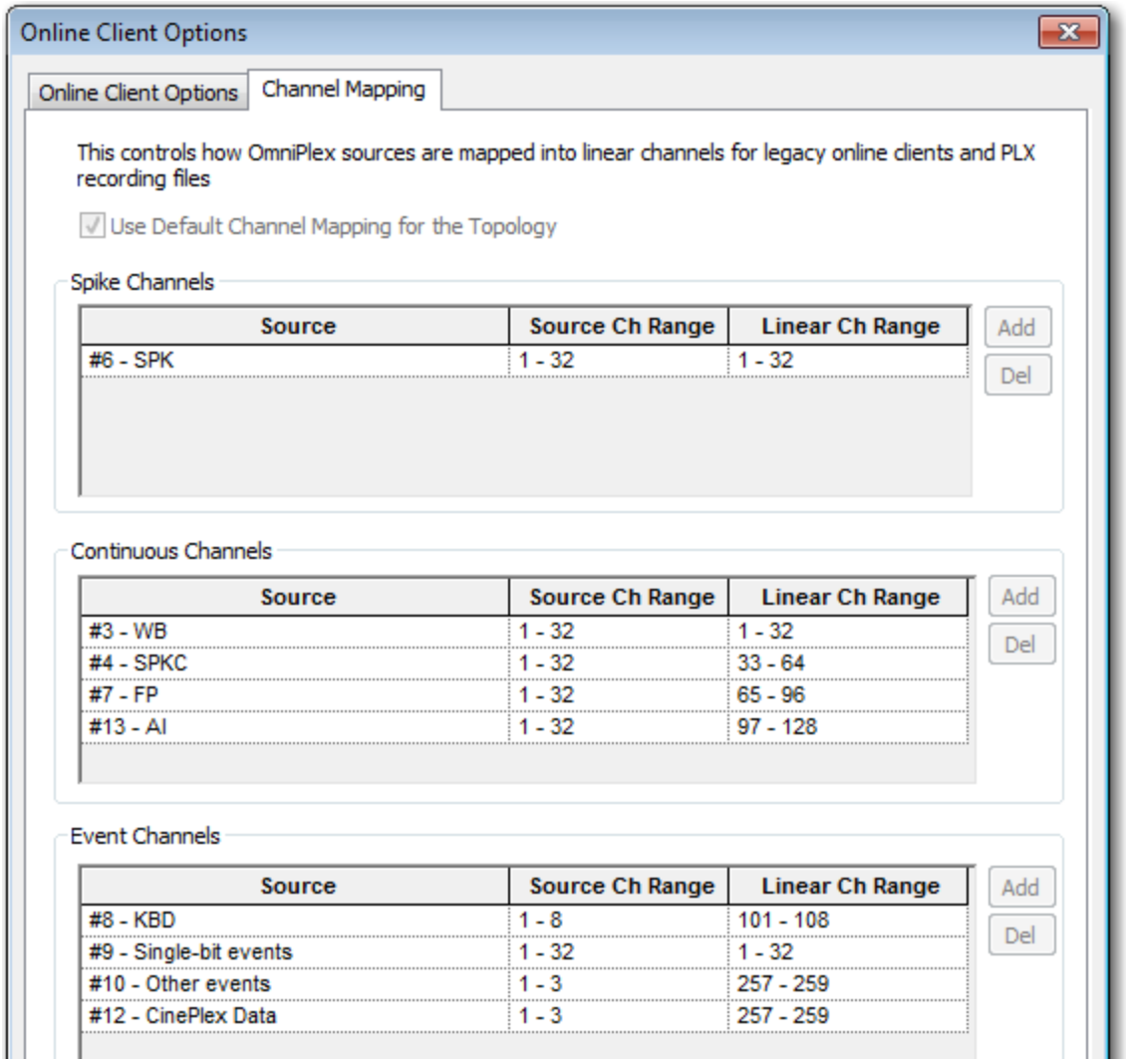
There are two conventions for identifying a channel from a particular source: *linear* channel numbers, as used in TimestampClient, SpikeWaveformClient, LowLatencyClient, the legacy client API, and PLX data files, and *source-relative* channel numbers, as used in ContinuousDataClient, MultiSourceClient and PL2 data files. InfoAndConversionsClient does not display live data, but gives extensive examples of how to perform queries about sources and channels using either linear or source-relative references and how to convert between them.

Linear channel numbers

In this scheme, also used in the legacy client API, a channel is uniquely identified by a type:

```
SPIKE_TYPE      1
EVENT_TYPE      4
CONT_TYPE       5
```

plus a channel number within the channels of that type. There is typically only one spike source in an OmniPlex topology (although this could change in future releases), but there are always multiple event and continuous sources. These multiple sources, each with its own range of channels, are mapped onto the three types, by allocating a subrange of channel numbers within each type. For any given OmniPlex topology, these mappings can be viewed in the *Online Client Options* dialog in Server. Here is an example, for a typical 32 channel OmniPlex system:



In the legacy client API, there was no way for a client to determine these mappings programmatically; that information (for example, that the 32 SPKC channels were channels of

CONT_TYPE from channels 33-64) had to be viewed in OmniPlex and hand-coded into the client program. The Native Client API provides functions for querying the mappings, and for translating from linear channel numbers to source-relative channel numbers and vice-versa. Clients can work with either method of channel numbering, but source-relative channel numbers are the OmniPlex native format and don't require you to deal with channel mappings.

Do not confuse channel *mapping* as described above with channel *remapping*, which is an arbitrary channel reordering that can be specified via .cmf remapping files in OmniPlex. Remapping is usually performed in order to normalize unusual channel numbering in electrode arrays or headstages. Client programs always work with channel numbers *after* any remapping has already been performed, and in fact are unaware of whether a remapping is in effect.

Source-relative channel numbers

With source-relative channel numbers, a channel is uniquely identified by its source name or number, plus a channel number within that source. Each source's channel numbers start at 1 and are numbered independently of other sources. As opposed to the somewhat unintuitive linear channel numbering scheme, the source-relative equivalent for our example 32 channel system is more descriptive and straightforward:

```
SPK01 - SPK32
WB01 - WB32
SPKC01 - SPKC32
FP01 - FP32
AI01 - AI32
KBD1 - KBD8
EVT01 - EVT32
CPX1 - CPX3
```

(Note that the Other Events source is a special case, in that its three channels are named Strobed, RSTART, and RSTOP, not shown in the above list. These are dedicated event channels for strobed word data, recording start, and recording stop events respectively, and correspond to channels 1, 2 and 3 on the Other Events source.)

Selecting the channel numbering method (data format)

Clients can call functions to get the number of sources and their properties, as well as functions to convert from source-relative to linear channel numbers and vice versa. These query, conversion, and utility functions can be used at any time and in any sequence; see the InfoAndConversionsClient for examples of usage.

However, when a client connects to Server using OPX_InitClient, the dataFormat parameter sets the format of the live data stream returned by the OPX_GetNewTimestamps and the other OPX_GetNew* functions. If the dataFormat is OPX_CHANNEL_FORMAT_LINEAR, then source numbers are *not* included in the live data, and the channel number within a source type

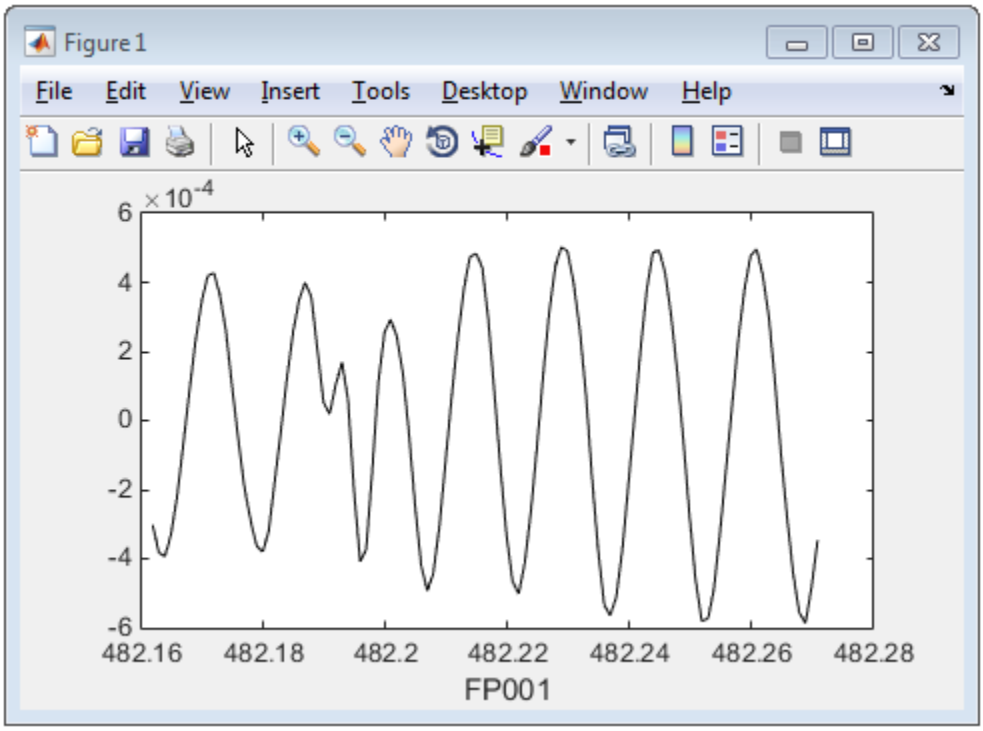
determines which source it belongs to. You would typically use linear format only if your client is not working with continuous data.

If the `dataFormat` is `OPX_CHANNEL_FORMAT_SOURCE_RELATIVE`, then channel numbers will begin at 1 for each source, and the source number *is* included as part of the online data. When working with continuous data, this means that you can refer to channels by their source name and source channel number or source number and source channel number, which is usually more intuitive than working with subranges of channel numbers.

See the comments in the `OPX_Get*.m` files for the organization of the data they return, depending on whether you are using linear or source-relative data format. In particular, note that whether the source number is included or not affects the indexing of items following the source number. The sample clients give examples of correct indexing for both linear and source-relative formats.

ContinuousDataClient

ContinuousDataClient draws the signal from a single channel within a continuous source. The sample code uses the first channel on the FP source, but the same approach will work for any continuous source and channel number.



Since this client works with source-relative channel numbers, it calls OPX_InitClient with the value `OPX_CHANNEL_FORMAT_SOURCE_RELATIVE` for the `dataFormat` parameter.

The calls to `OPX_ExcludeAllSourcesOfType` show how a client can reduce the amount of data

that is sent to it by Server; in this case, we eliminate spikes, digital events, and all continuous data except the FP source. Excluding and including sources can be done by source type, source name, or source number, in any combination; the order of exclusions and inclusions determines the net result. Source exclusion is totally optional but can make client code simpler and more efficient in many cases.

The client calls `OPX_GetContSourceInfoByName` in order to obtain the sampling rate for the FP source, which will be used to draw the continuous data with the correct time scaling. If we were working with more than one continuous source, we could use the `sourceNum` returned by this call to look for a particular continuous source in the continuous data returned by `OPX_GetNewData`.

Within the main loop, only the "cont" return values from `OPX_GetNewData` are used. `numCont` tells us the number of continuous channels which have data. `contHeaders` is an array of size `numCont` which contains the source numbers, source-relative channel numbers, and number of samples for the channels which have data. We iterate through the channel headers, looking for the one with the desired channel number, then draw that channel's samples. We don't need to examine the source number in the channel header, since we have excluded all the continuous sources except the FP source.

To draw the samples, we obtain the number of samples from the channel header and the time of the first sample from the `contTimes` array. We use the variable `dtFP`, which is the interval between samples (e.g. 1 ms at a sampling rate of 1 kHz), to construct the array `x` of sample times.

Note that by default, the client sample rate limit in Server only enables continuous channels with a sampling rate of 1 kHz or less to be sent to clients, in order to avoid wasting bandwidth and processing resources for the majority of clients which are not interested in higher-rate continuous data. If your client wishes to read WB or SPKC data, or FP data where the FP rate is greater than 1 kHz, you will need to increase the client sample rate limit in Server accordingly.

MultiSourceClient

MultiSourceClient can be seen as a combination and extension of SpikeWaveformClient and ContinuousDataClient. It displays a selected channel on "parallel" sources, i.e. sources derived from the same wideband channel, by default WB001, SPKC001, FP001 and SPK001. This sample client also introduces a few additional techniques as described below.

The function `OPX_GetGlobalParameters` returns many items of information about the current OmniPlex configuration, but in this client we are only interested in the client rate limit. If the rate limit is 40 kHz, we will display channels from the WB, SPKC and FP continuous sources, but if the rate is less than this only FP will be shown.

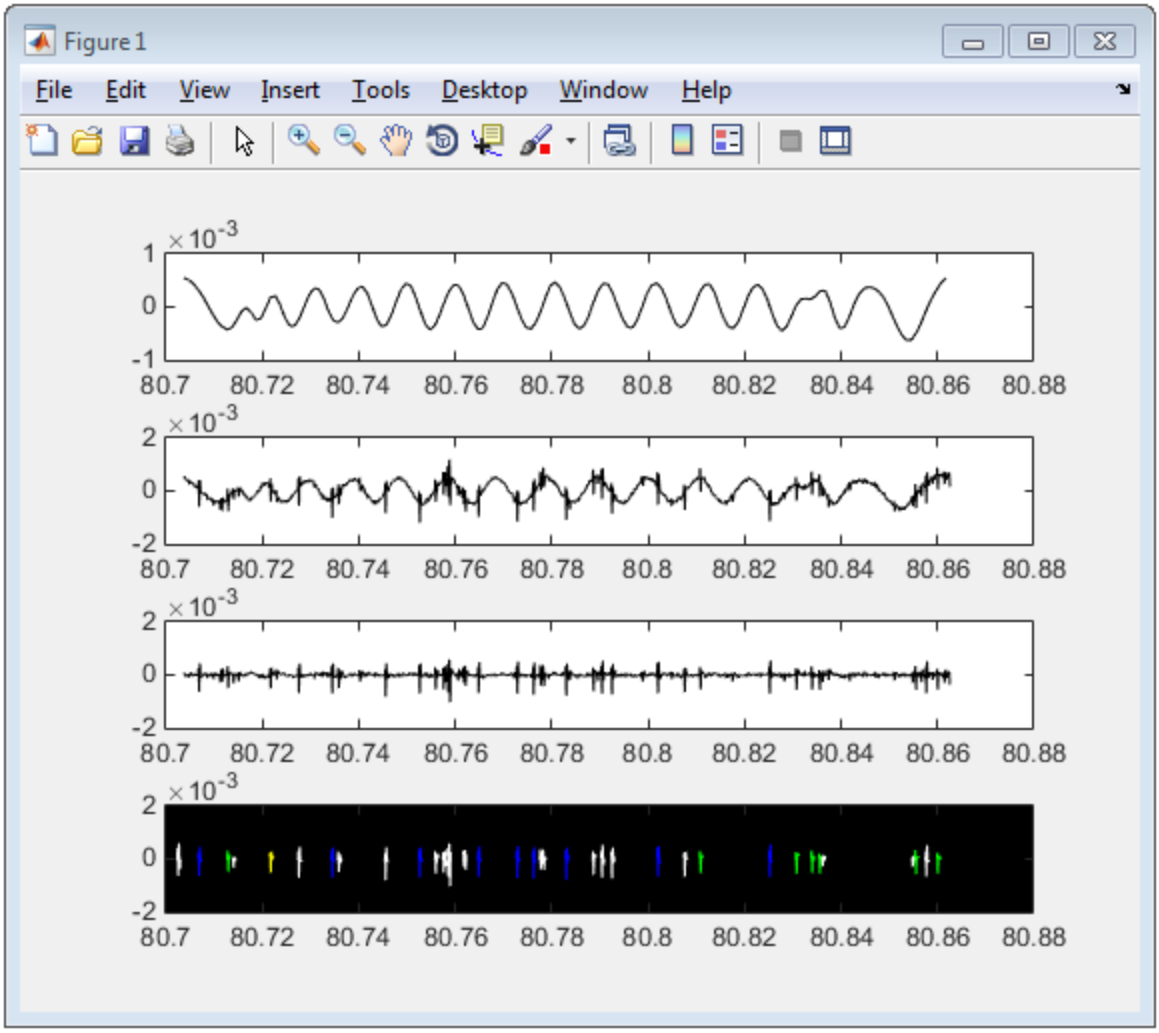
`OPX_GetContSourceInfoByName` is called to obtain the sampling rate for the continuous sources, and `OPX_GetSpikeSourceInfoByName` is called to get the spike sampling rate. For the SPK source we also need the number of pre and post threshold points in order to draw spikes with

correct time scaling, since spike timestamps are defined relative to the threshold crossing (or waveform peak, if aligned extraction is used), not the first sample as with continuous data.

In the main loop, we draw spikes and either one or three continuous source channels, depending on whether the client rate limit is high enough to receive WB and SPKC data. Note that compared to SpikeWaveformClient, where we simply used the sample index (e.g. 1 to 32) for the scaling in x and spikes were drawn on top of each other, here we use the spike timestamps to draw each spike in its correct position in time relative to other spikes that we are drawing within each batch of new data.

Since the x scaling of each of the subplots is independent, events are not guaranteed to exactly line up across different subplots, and in particular the x scaling of the spike subplot will depend on the span of time included by each batch of spikes. For example, if only one spike is seen in a batch of data from `OPX_GetNewData`, that one spike will occupy the entire horizontal extent of the subplot. In effect, its view will be zoomed in (in time) compared to the continuous subplot(s). This could be addressed by pre-scanning all the spike and continuous data to find the maximum time extent across all sources within each batch of new data before displaying it, and then enforcing the same time (x) scaling on all the subplots, but this was omitted in order to keep the sample code straightforward.

Another consequence of each subplot being scaled independently and automatically by MATLAB is that the amplitude scaling is different for each, and for each batch of data. For example, if within one batch of new data, there is nothing but noise in the displayed SPKC channel, this will be displayed with a different amplitude scaling than a segment of the signal where large spikes are present. The optimal "display behavior," e.g. manual versus automatic scaling and independent versus uniform scaling, is application-dependent and beyond the scope of the sample clients; we only mention it so that you are aware of it.



Note that unlike ContinuousDataClient, here the client does need to examine the source numbers in the continuous data, in order to draw the continuous data in the correct subplot, and with the correct time scaling, depending on the source's sampling rate.

InfoAndConversionsClient

Unlike the preceding sample clients, InfoAndConversionsClient does not enter an infinite loop where it periodically reads batches of new data from Server. Instead, it prints out a large amount of information about the current state of OmniPlex in the command window and then exits. It is intended to show the usage of the many functions provided for querying the characteristics of sources and channels, for referring to sources either by name or by number, and for converting between linear channel numbers and source-relative channel numbers. As is, you can use it as a utility to dump the information on all the sources and channels in your systems, in addition to global parameters such as the client sample rate limit, the OmniPlex subsystem type (DHP, DigiAmp, or OPX-A), etc.

Note that in some cases InfoAndConversionsClient performs that same function more than once, using alternate forms of a function, e.g. by source name and then by source number, with the consequence that you may see the same information printed out more than once. You can delete or comment out the redundant code if you wish to reduce the amount of output.

LowLatencyClient

In some applications it is important to minimize the latency (delay) between the time of acquisition of a neural signal or digital event and the time when the acquired data is delivered to a client program. For example, in neural prosthetics and other closed-loop applications, neural activity is decoded in real time and used to control an prosthetic device, an electrical or optical stimulator, or other hardware. In neural prosthetics, closed loop delays of more than a few tens of milliseconds are generally not acceptable. OmniPlex clients can achieve closed loop latencies of under 10 ms under most conditions, as long as the correct techniques are used, and the client-specific processing (e.g. a decoder) does not add appreciable latency.

LowLatencyClient is similar to TimestampClient, but rather than pausing for a fixed interval before reading new data, i.e. polling, it uses a Server wait event. This is a Windows synchronization event which is signaled as soon as new data is available. By waiting on this event, a client can minimize unnecessary delays in acquiring each batch of new data.

To do this, a client obtains a wait handle by a call to `OPX_GetWaitHandle`, before entering its acquisition loop. Then within the main loop, instead of using a fixed duration pause, the function `OPX_WaitForNewData` is called. You can think of this as "pause until new data is available." The sample code simply prints out the number of spikes and digital events in each batch of data, along with the current time, to indicate the approximate time between reads from the Server. Note that unless you un-comment the indicated `fprintf` statement, output will be generated only when there are spikes and/or events, not on "empty reads."

For the very lowest latency, the client synchronization event in combination with the *Lowest Latency* and *Minimize Client Latency* options in Server can be used (see the section on low latency operation in the OmniPlex User Guide for details on these options). However, be aware that enabling both of these options will result in a client waking up to read new data at a rate of approximately 2000 Hz. Any non-trivial MATLAB code cannot keep up with this update rate unless it is highly optimized and does not attempt to redraw graphical displays frequently. If your client has performance problems, you can use the MATLAB "Run and Time" function or other profiling tools to determine which sections of your code are in need of optimization. In any case, it is recommended that you write and debug your client with normal latency settings in Server to begin with, and then only enable *Lowest Latency* and *Minimize Client Latency* in Server if necessary to achieve acceptable latencies.

Additional considerations in client programming

Sending data to OmniPlex from a client

In some cases a client may wish to send its own data to Server, to be timestamped and recorded along with spikes, events, and continuous signals. This can be done using the `OPX_SendClientDataWords` function, whose parameters are an array of 16 bit data words and the count of the number of data words, with a maximum of 32 words in each call. In itself this is straightforward, but there are some limitations that you should be aware of. If you are not using hardware strobed data input with the DI card, and you are not using CinePlex at the same time as OmniPlex, the issues described in the remainder of this section will not apply.

In OmniPlex, there is the concept of a "strobed event channel," which accepts timestamped 16 bit word values. In current OmniPlex systems, this channel is in effect shared by (i.e. is an alias for) several potential generators of timestamped event word data:

1 Hardware-strobed event words from the DI card
2 Software-generated coordinate data words from CinePlex
3 Client data words sent via `OPX_SendClientDataWords`

In OmniPlex, and in PL2 data files, both of which operate in terms of sources, there are *two* strobed event channels: channel 1 on the Other Events source, which is used exclusively by hardware-strobed DI data (#1 above), and channel 1 on the CinePlex source, which is shared by CinePlex data (#2) and client data words (#3). For example, in the Activity View in PlexControl, you will see separate traces for Strobed (channel 1 on Other Events) and CPX1 (channel 1 on the CinePlex source). *If both CinePlex and a client are sending data words, ticks for both will be seen on the same CPX1 line in the Activity View.* In summary, unless you are using PLX files, the only conflict to be aware of is CinePlex data and client data. If you are not using CinePlex, client data has the exclusive use of the CPX1 channel.

In PLX files, all three (#1,#2,#3 above) are aliased to a single PLX channel, event channel 257. In this case, strobed word data, data from CinePlex, and client-injected data words, will all end up on the same channel, and cannot be disambiguated in the recorded data, except by a user-defined convention, such as restricting the range of allowed word values from each.

This facility is intended for relatively low bandwidth client data, typically 100 16 bit words per second or less. Sending more data than this may be possible, but you should perform tests with your OmniPlex system and your client to establish the maximum reliable bandwidth before using it in an actual experiment.

Timestamping versus the order in which data is delivered

All OmniPlex data read by a client has been accurately timestamped by the OmniPlex hardware. Even "soft" events such as keyboard events from PlexControl are timestamped by reading the OmniPlex hardware clock at the time the keypress is detected. The following caveats concern the *order* in which this timestamped data is delivered to clients, *not* the accuracy of the

timestamps themselves.

It is guaranteed that the timestamped data for any individual channel will be received by a client in correct time-increasing order. However, within each batch of data read from Server, timestamps from different sources are not guaranteed to be sorted into a global increasing-timestamp order. An example using only three channels from three sources might be:

```
SPK07       534.188075
EVT14       534.187000
AI03        534.187025
SPK07       534.197150
AI03        534.197025
AI03        534.207025
SPK07       534.219650
EVT14       534.211075
...
```

This is partly due to the fact that OmniPlex collects the hardware-timestamped data from different devices asynchronously and in some cases at different rates.

Per-channel client buffers and multi-source processing

Some clients immediately process the timestamped data from OmniPlex and have no need to save it. However, in some cases a client will maintain its own buffers of data from OmniPlex, for example, a circular buffer (ring buffer) of the most recent data; new data is appended to one end of the buffer and consumed from the other end. In such scenarios, clients will find it more convenient to store data in per-channel buffers; as timestamped blocks of data are received, each block is appended to the buffer for the corresponding channel. Within each channel's buffer, timestamped data will therefore naturally be stored in increasing-time order, without the need to do any kind of global (i.e. cross-source) reordering (time sorting) of the timestamped data. For the above example, the contents of three per-channel buffers after reading the above data would be (each column representing one per-channel buffer):

```
SPK07                AI03             EVT14
----------           ----------       ----------
534.188075           534.187025       534.187000
534.197150           534.197025       534.211075
534.219650           534.207025
```

If each channel's data is processed by the client independently of other channels and sources, then all the data from the per-channel buffers can be completely consumed (processed and deleted from the buffer) for each batch of new data, before the next batch is read.

On the other hand, if a client requires that data from multiple sources be processed together (for example, you wish to perform spike-triggered averaging of continuous data, or calculate peri-event spike histograms), it is not in general possible to consume all the data from all the channels each time, because of the asynchronous delivery of data from different devices at different rates.

Consider the most recent (largest) timestamp on any block of data from any source, within a given batch of new data (in the above example, `SPK07` at `534.219650`). For a different source, the data whose timestamp is at or near to that largest timestamp might not be delivered until the next batch of data that is read (for example, data from `AI03` at `534.217025` through `534.237025`).

The extent to which this affects a client is often minimal, and many clients ignore it. Correcting for it requires incurring one or more "read cycles" of latency, e.g. deferring the processing of the data for batch *k* until batch *k+1* has been read. Note that clients which perform functions such as peri-event histograms or spike-triggered averaging will already need to buffer data corresponding to a post-event interval, and so they will presumably implement a mechanism for deferred processing of buffered data.

Clients and PlexNet

NCA clients can run on the remote end of a PlexNet connection without changes, with the following exceptions. There may be additional latency caused by the network, although if a fast LAN is used without significant non-PlexNet traffic, and wideband data is not sent, this is often negligible. See the PlexNet documentation for more information. Note that sending of event words from client to Server is not currently supported for remote clients, and filter and gain parameters are only updated for remote clients when they initially connect to PlexNetRemote.

For more information or assistance

If you have questions about building, running or understanding the clients, or questions in general about writing clients, please contact Plexon technical support at support@plexon.com.