

## **Introduction**

The Python Native Client API for OmniPlex (PyOPXClient) is a set of classes and 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. It is compatible with OmniPlex R18 and above.

PyOPXClient is compatible with Python 3 in Windows environments. There is a separate version of this SDK for Python 2.

PyOPXClient contains a set of sample client programs that demonstrate how to read spike, continuous, and digital event data from OmniPlex. Some samples are run in a command window, and other samples demonstrate the client API being used with a GUI (Graphical User Interface).

Each class, as well as each sample client, has extensive code comments describing how the classes/functions are used.

The first sample client, `timestamp_client.py`, only reads spike and digital event timestamps, not spike waveforms or continuous data. This is close to the minimum amount of code needed to read data from OmniPlex. This sample runs in a command window.

The second sample client, `read_all_sources.py`, shows how to work with OmniPlex sources, how to read spike waveforms and continuous sample data, and query voltage scaling (gain) information so that spike waveforms and continuous data are scaled correctly. This sample runs in a command window.

The third example client, `strobemonitor.py`, is a GUI program that displays the timestamps and values of incoming strobed events.

The fourth example client, `gridmon.py`, is a GUI program that uses colored boxes to show the average ISI of the first unit of 16 or 32 channels.

## **Unpacking**

PyOPXClient's folder and file structure look like this (folders in bold):

### **PyOPXClient**

- bin**
  - OPXClient.dll
  - OPXClient64.dll
- pyopxclient**
  - \_\_init\_\_.py
  - pyopxclientapi.py
  - pyopxclientlib.py
- example\_libs**
  - gridmon\_lib**
  - strobemonitor\_lib**
- read\_all\_sources.py
- timestamp\_client.py

strobemonitor.py  
gridmon.py

The **bin** folder holds the dll files that the libraries in the **pyopxclient** folder use to access data in OmniPlex. The **example\_libs** folder contains supporting files for the examples gridmon.py and strobemonitor.py.

Only the **bin** and **pyopxclient** folders are needed to write your own clients. Your client program should have those folders in the same folder as your client program.

### **Running the sample clients**

To run the sample clients, first start OmniPlex Server and PlexControl with a valid configuration, and run the test .wav file through the system with the headstage test board. In a command prompt window, navigate to the PyOPXClient folder and run a sample client. For example:

```
C:\PyOPXClient>python timestamp_client.py
```

Implementation details of sample clients will be touched on later in this document, but there are also detailed comments in the source code.

### **Sources and Channels**

It's helpful to have an understanding of how sources and channels work in OmniPlex when using any of the Plexon client SDKs.

OmniPlex digitizes what we call a "wideband" signal. Highpass (spike) and lowpass (LFP) filtering of the wideband are done in the OmniPlex software. The wideband, spike, and LFP signals are separate "sources". The thresholded spike waveforms, digital events, and auxiliary analog input are also each separate sources. Each source has a unique number and unique name. Source numbers aren't guaranteed to be identical between different configurations. It's best to identify the source by name to get data from it, or use the source name to associate a source number. This is done in the example clients.

Each source has a channel range from 1 to the number of channels on the system. For example, on a 32 channel system, the wideband source will by default be called "WB", might have source number 6, and will have channels 1 – 32. The spike waveform source will by default be called "SPK", might have source number 2, and will have channels 1 – 32.

Although this arrangement might seem obvious to a new user, this is not how the older Client SDKs addressed channels. The original MAP system and the old .plx file format didn't use sources. They instead used one long range of channel numbers. For example, on a 32 channel system, the wideband data would be channels 1 – 32, the spike filtered continuous data would be channels 33 – 64, and the continuous field potential channels would be 65 – 96.

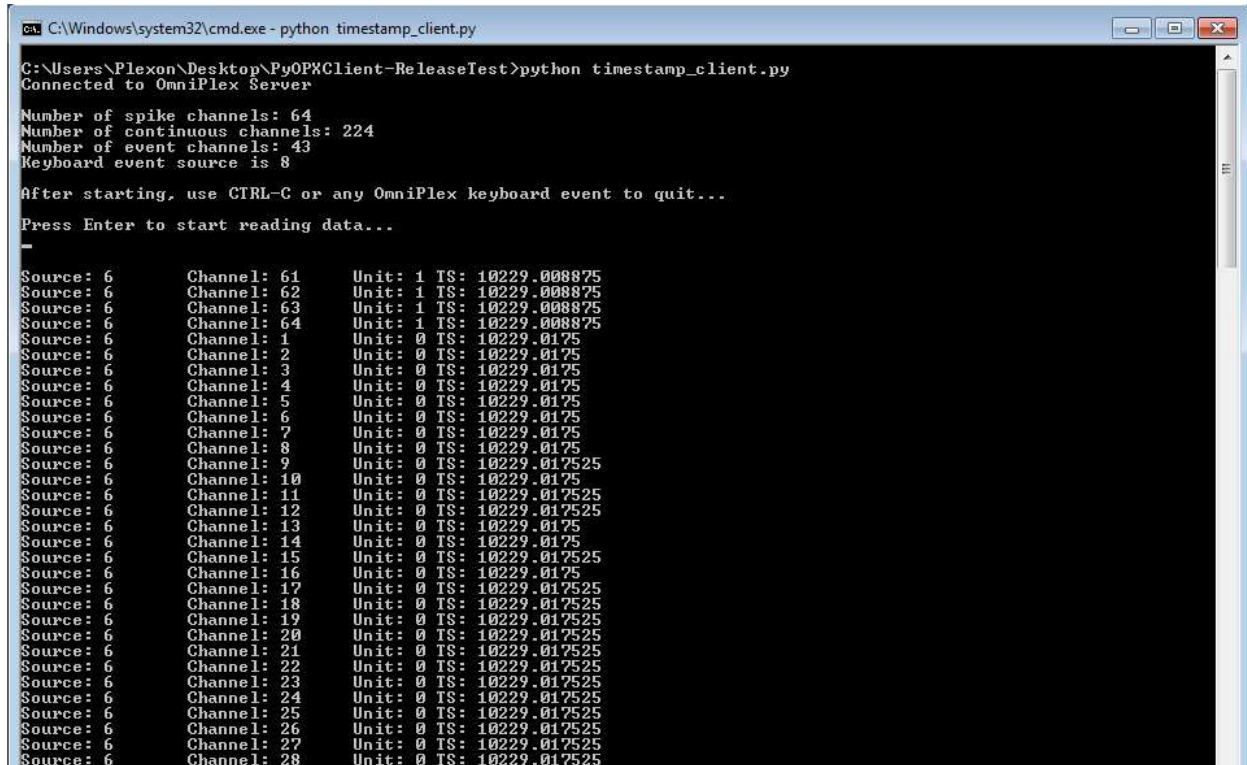
The client SDK by default uses the source-oriented way of addressing channels, but it can technically be configured to use the old "linear channel" style if absolutely necessary (however none of the samples demonstrate this).

For more information on sources, the OmniPlex manual and the documentation for the C/C++ client SDK goes into greater detail.

### Timestamp Client Example

To run the timestamp client example:

C:\PyOPXClient>python timestamp\_client.py



```
C:\Windows\system32\cmd.exe - python timestamp_client.py
C:\Users\Flexon\Desktop\PyOPXClient-ReleaseTest>python timestamp_client.py
Connected to OmniPlex Server

Number of spike channels: 64
Number of continuous channels: 224
Number of event channels: 43
Keyboard event source is 8

After starting, use CTRL-C or any OmniPlex keyboard event to quit...
Press Enter to start reading data...
-
Source: 6      Channel: 61      Unit: 1 TS: 10229.008875
Source: 6      Channel: 62      Unit: 1 TS: 10229.008875
Source: 6      Channel: 63      Unit: 1 TS: 10229.008875
Source: 6      Channel: 64      Unit: 1 TS: 10229.008875
Source: 6      Channel: 1       Unit: 0 TS: 10229.0175
Source: 6      Channel: 2       Unit: 0 TS: 10229.0175
Source: 6      Channel: 3       Unit: 0 TS: 10229.0175
Source: 6      Channel: 4       Unit: 0 TS: 10229.0175
Source: 6      Channel: 5       Unit: 0 TS: 10229.0175
Source: 6      Channel: 6       Unit: 0 TS: 10229.0175
Source: 6      Channel: 7       Unit: 0 TS: 10229.0175
Source: 6      Channel: 8       Unit: 0 TS: 10229.0175
Source: 6      Channel: 9       Unit: 0 TS: 10229.017525
Source: 6      Channel: 10      Unit: 0 TS: 10229.0175
Source: 6      Channel: 11      Unit: 0 TS: 10229.017525
Source: 6      Channel: 12      Unit: 0 TS: 10229.017525
Source: 6      Channel: 13      Unit: 0 TS: 10229.0175
Source: 6      Channel: 14      Unit: 0 TS: 10229.0175
Source: 6      Channel: 15      Unit: 0 TS: 10229.017525
Source: 6      Channel: 16      Unit: 0 TS: 10229.0175
Source: 6      Channel: 17      Unit: 0 TS: 10229.017525
Source: 6      Channel: 18      Unit: 0 TS: 10229.017525
Source: 6      Channel: 19      Unit: 0 TS: 10229.017525
Source: 6      Channel: 20      Unit: 0 TS: 10229.017525
Source: 6      Channel: 21      Unit: 0 TS: 10229.017525
Source: 6      Channel: 22      Unit: 0 TS: 10229.017525
Source: 6      Channel: 23      Unit: 0 TS: 10229.017525
Source: 6      Channel: 24      Unit: 0 TS: 10229.017525
Source: 6      Channel: 25      Unit: 0 TS: 10229.017525
Source: 6      Channel: 26      Unit: 0 TS: 10229.017525
Source: 6      Channel: 27      Unit: 0 TS: 10229.017525
Source: 6      Channel: 28      Unit: 0 TS: 10229.017525
```

The Timestamp Client example demonstrates the basic acquisition loop that every client will have in common.

- 1) Connect to OmniPlex
- 2) Get new data
- 3) Do something with the data
- 4) Wait for new data to accumulate in OmniPlex
- 5) Go to Step 2

The source for timestamp\_client.py has comments explaining each section. Here are a few notes about Timestamp Client that aren't explained in detail in the source.

```
from pyopxclient import PyOPXClientAPI, OPX_ERROR_NOERROR
```

The first line imports the PyOPXClientAPI class, as well as OPX\_ERROR\_NOERROR, which is simply a variable equal to 0. API functions will return 0 upon success. Checking against a named variable simply improves readability. Other error codes have variables, and you can see them at the bottom of pyopxclientlib.py.

```
# Get global parameters, print some information
global_parameters = client.get_global_parameters()
```

```
# Figure out which source is for keyboard events, which is used to end the
program
for source_id in global_parameters.source_ids:
    source_name, _, _ = client.get_source_info(source_id)
    if source_name == 'KBD':
        keyboard_event_source = source_id
        print ("Keyboard event source is {}".format(keyboard_event_source))
```

Timestamp Client collects data and prints out spike or event timestamps until the user presses ctrl-C, or sends a keyboard event. Keyboard events are sent in PlexControl either through the Event Input window, or by pressing alt-1 through alt-8.

The keyboard event source is by default called “KBD”. The block of code above goes through every source ID (which is a number) finds which of those source numbers have the name “KBD”, and associates it to a variable keyboard\_event\_source.

Later in the code, each returned timestamp is checked to see if it’s from the KBD source, and if so, execution is ended. This is a convenient way to end a client program that doesn’t depend on forcing the program to end with ctrl-C.

```
try:
    while(running):
        . . .

except KeyboardInterrupt:
    print ("\nCTRL-C detected; stopping acquisition.")
```

Another way of handling ctrl-C is to wrap the code up in a try/except block that looks for the KeyboardInterrupt exception, as seen above.

```
# Wait up to 1 second for new data to come in
client.opx_wait(1000)
```

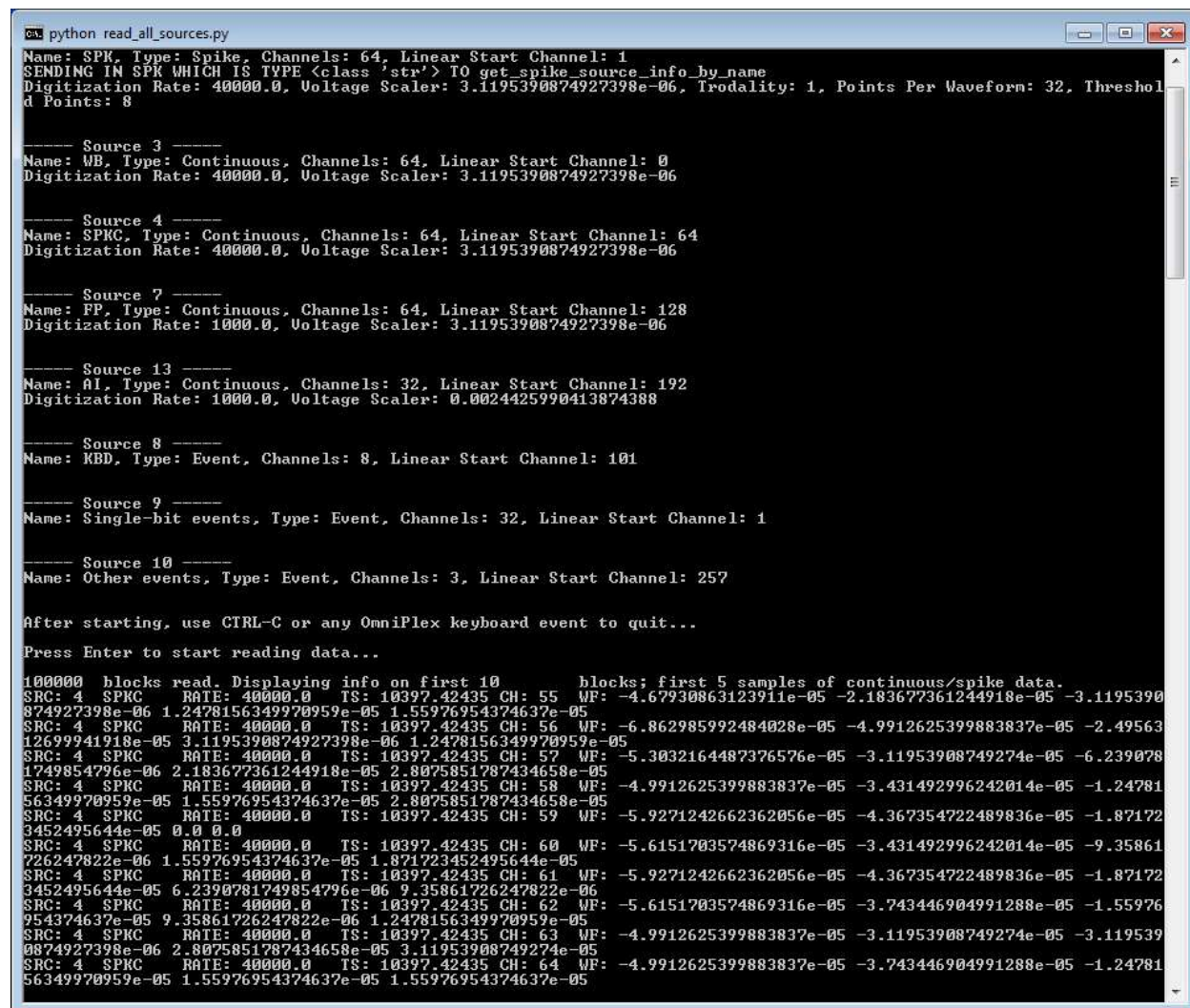
opx\_wait() is a PyOPXClientAPI class method that blocks execution until new data comes into the OmniPlex server buffer. The value passed is the max time to wait, in milliseconds. It is very highly

recommended to use the `opx_wait` function, or some other delay function. Requesting new data in a very tight loop can cause problems.

### Read All Sources Example

To run the read all sources client example:

C:\PyOPXClient>python read\_all\_sources.py



```
python read_all_sources.py
Name: SPK, Type: Spike, Channels: 64, Linear Start Channel: 1
SENDING IN SPK WHICH IS TYPE <class 'str'> TO get_spike_source_info_by_name
Digitization Rate: 40000.0, Voltage Scaler: 3.1195390874927398e-06, Trodality: 1, Points Per Waveform: 32, Threshold Points: 8

----- Source 3 -----
Name: WB, Type: Continuous, Channels: 64, Linear Start Channel: 0
Digitization Rate: 40000.0, Voltage Scaler: 3.1195390874927398e-06

----- Source 4 -----
Name: SPKC, Type: Continuous, Channels: 64, Linear Start Channel: 64
Digitization Rate: 40000.0, Voltage Scaler: 3.1195390874927398e-06

----- Source 7 -----
Name: FP, Type: Continuous, Channels: 64, Linear Start Channel: 128
Digitization Rate: 1000.0, Voltage Scaler: 3.1195390874927398e-06

----- Source 13 -----
Name: AI, Type: Continuous, Channels: 32, Linear Start Channel: 192
Digitization Rate: 1000.0, Voltage Scaler: 0.0024425990413874388

----- Source 8 -----
Name: KBD, Type: Event, Channels: 8, Linear Start Channel: 101

----- Source 9 -----
Name: Single-bit events, Type: Event, Channels: 32, Linear Start Channel: 1

----- Source 10 -----
Name: Other events, Type: Event, Channels: 3, Linear Start Channel: 257

After starting, use CTRL-C or any OmniPlex keyboard event to quit...
Press Enter to start reading data...

100000 blocks read. Displaying info on first 10 blocks; first 5 samples of continuous/spike data.
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 55 WF: -4.67930863123911e-05 -2.183677361244918e-05 -3.1195390874927398e-06 1.2478156349970959e-05 1.55976954374637e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 56 WF: -6.862985992484028e-05 -4.9912625399883837e-05 -2.4956312699941918e-05 3.1195390874927398e-06 1.2478156349970959e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 57 WF: -5.3032164487376576e-05 -3.11953908749274e-05 -6.2390781749854796e-06 2.183677361244918e-05 2.8075851787434658e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 58 WF: -4.9912625399883837e-05 -3.431492996242014e-05 -1.2478156349970959e-05 1.55976954374637e-05 2.8075851787434658e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 59 WF: -5.9271242662362056e-05 -4.367354722489836e-05 -1.871723452495644e-05 0.0 0.0
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 60 WF: -5.6151703574869316e-05 -3.431492996242014e-05 -9.35861726247822e-06 1.55976954374637e-05 1.871723452495644e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 61 WF: -5.9271242662362056e-05 -4.367354722489836e-05 -1.871723452495644e-05 6.2390781749854796e-06 9.35861726247822e-06
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 62 WF: -5.6151703574869316e-05 -3.743446904991288e-05 -1.55976954374637e-05 9.35861726247822e-06 1.2478156349970959e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 63 WF: -4.9912625399883837e-05 -3.11953908749274e-05 -3.1195390874927398e-06 2.8075851787434658e-05 3.11953908749274e-05
SRC: 4 SPKC RATE: 40000.0 TS: 10397.42435 CH: 64 WF: -4.9912625399883837e-05 -3.743446904991288e-05 -1.2478156349970959e-05 1.55976954374637e-05 1.55976954374637e-05
```

The Read All Sources example extends the timestamp client example by acquiring continuous data and spike waveforms. The source for `read_all_sources.py` has comments explaining each section. Here are a few notes about Read All Sources that aren't explained in detail in the source.

```
# This will be filled in later. Better to store these once rather than have
# to call the functions
# to get this information on every returned data block
source_numbers_types = {}
source_numbers_names = {}
source_numbers_rates = {}
```

```
source_numbers_voltage_scalers = {}
```

These dicts start off empty, but are later filled with information gathered by `get_global_parameters()`. The keys are the source numbers, and the values are the types (spike/event/etc), names, digitization rates, and voltage scalers.

The spike waveform and continuous waveform values are returned in units of A/D values, which are whole numbers. To convert to voltage, the values are multiplied by the respective source voltage scaler.

```
if source_numbers_types[new_data.source_num_or_type[i]] == SPIKE_TYPE:
    print ...

if source_numbers_types[new_data.source_num_or_type[i]] == CONTINUOUS_TYPE:
    print ...

if source_numbers_types[new_data.source_num_or_type[i]] == EVENT_TYPE:
    print ...
```

Spike, continuous, and event data have different information to be presented (digital events don't have a digitization rate, for example). The data type associated with the source number stored in the `source_numbers_types` dict is used to determine how the information is printed out in the client.

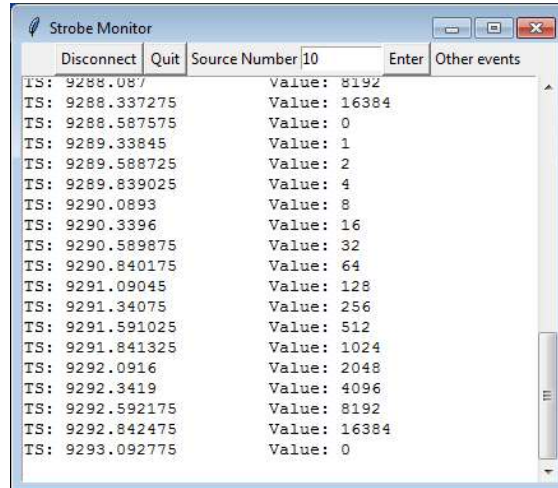
```
# Pause execution, allowing time for more data to accumulate in OmniPlex Server
sleep(poll_time_s)
```

The Timestamp Client example used `opx_wait()` to pause execution until new data is collected in OmniPlex. Read All Sources uses the `sleep` function in Python's `time` module.

## Strobe Monitor Example

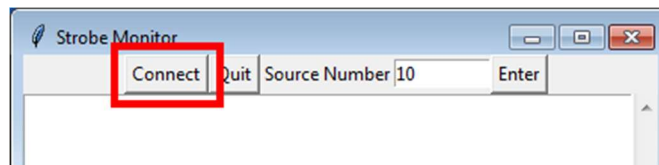
To run the Strobe Monitor client example:

```
C:\PyOPXClient>python read_all_sources.py
```

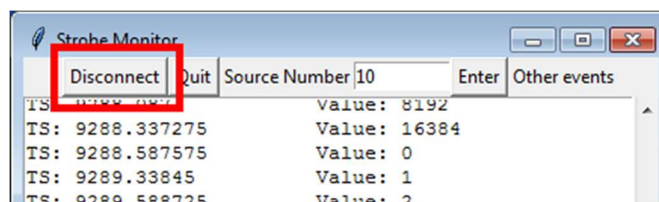


Strobe Monitor prints the timestamp and value of incoming strobed values on the OmniPlex digital input board. It's meant to demonstrate how to integrate PyOPXClient with a user interface framework. This example uses Tkinter, the GUI framework built into Python.

To use Strobe Monitor, first click on the "Connect" button.



When connected, you can disconnect by clicking the "Disconnect" button.



When Strobe Monitor is first run, it has a default source number of 10 which on most OmniPlex systems is the source associated with the strobed word events. If this isn't correct for your system, you can change the value and hit the Enter key (or press the "Enter" button).

To keep things simple, there isn't any bounds checking in the Source Number entry. If you set the source number to the spike source (typically 6), it will output the incoming spike timestamps.

Strobe Monitor generally follows the common model-view-controller design pattern. As such, there is a view class that contains all the user interface widgets and methods, a model class that handles getting

data from OmniPlex, and a controller class that manages the relationship between the view and the model.

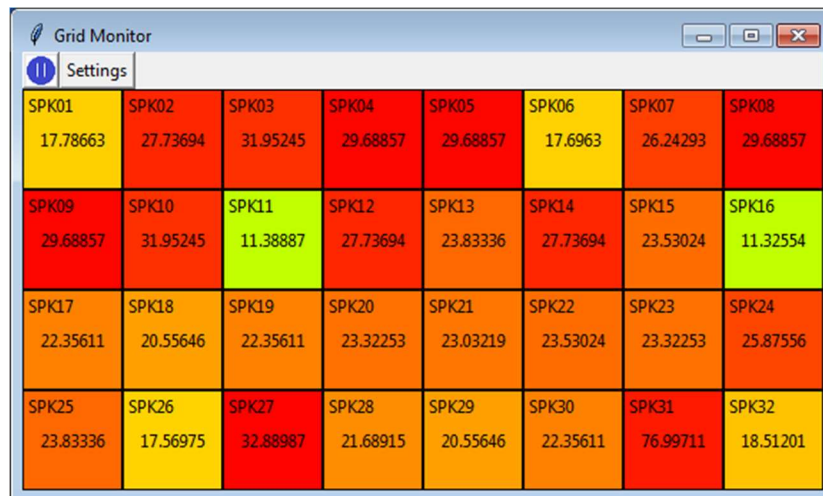
Strobe Monitor checks for new data in OmniPlex every 250 milliseconds. This is accomplished using a timer function called `UITimerFunction` in the view class. This timer function itself runs a timer callback function that is set in the controller class. Tkinter has its own timer methodology (using the `.after()` method), but there should be something similar in any other modern GUI framework, and it can even be done using a separate threading module.

For more information on how Strobe Monitor works, see the source comments.

### **GridMon Example**

To run the Strobe Monitor client example:

```
C:\PyOPXClient>python gridmon.py
```



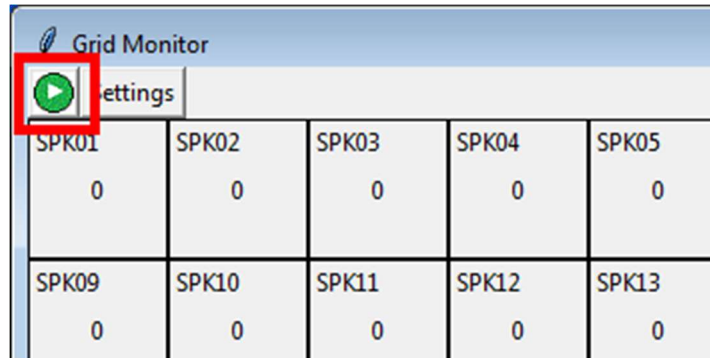
If you happen to be running the demo version of OmniPlex, run this example with an additional command line toggle:

```
C:\PyOPXClient>python gridmon.py -demo
```

Grid Monitor (GridMon) is a client example that displays the average inter-spike interval (ISI) of the first unit of the first 16 or 32 channels in a grid using a number value as well as a color.

After starting, click the "Play" button.

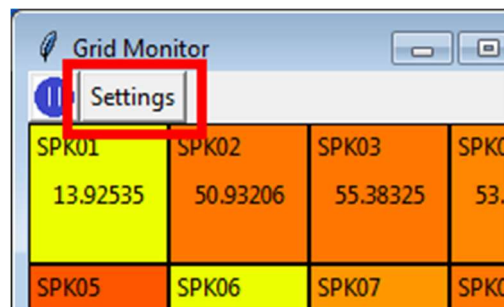




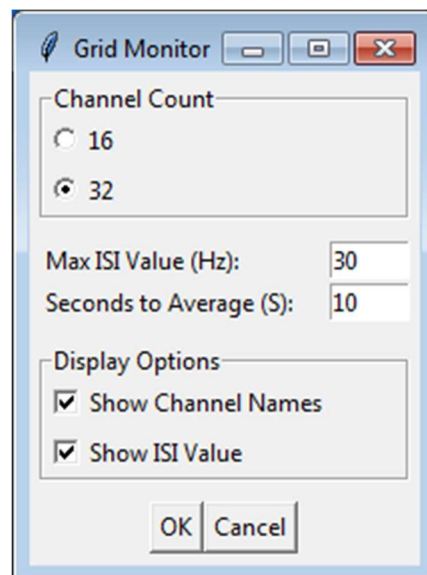
Grid Monitor				
Settings				
SPK01	SPK02	SPK03	SPK04	SPK05
0	0	0	0	0
SPK09	SPK10	SPK11	SPK12	SPK13
0	0	0	0	0

On connecting, by default, GridMon stores for each channel the past ten seconds of timestamps of the first unit, and averages the ISI for each channel. The ISI is displayed as a numeric value, as well as a color that ranges from green (0 Hz) to red (30 Hz).

GridMon has various settings that can be accessed by clicking the “Settings” button.



Grid Monitor			
Settings			
SPK01	SPK02	SPK03	SPK04
13.92535	50.93206	55.38325	53....
SPK05	SPK06	SPK07	SPK08
...	...	...	...



Grid Monitor

Channel Count

☐ 16

☒ 32

Max ISI Value (Hz):

Seconds to Average (S):

Display Options

☒ Show Channel Names

☒ Show ISI Value

OK Cancel

**Channel Count:** 16 or 32 channels.

**Max ISI Value (Hz):** This value or above, in Hz, will display as red.

**Seconds to Average (S):** How many seconds of spike timestamps on each channel to hold.

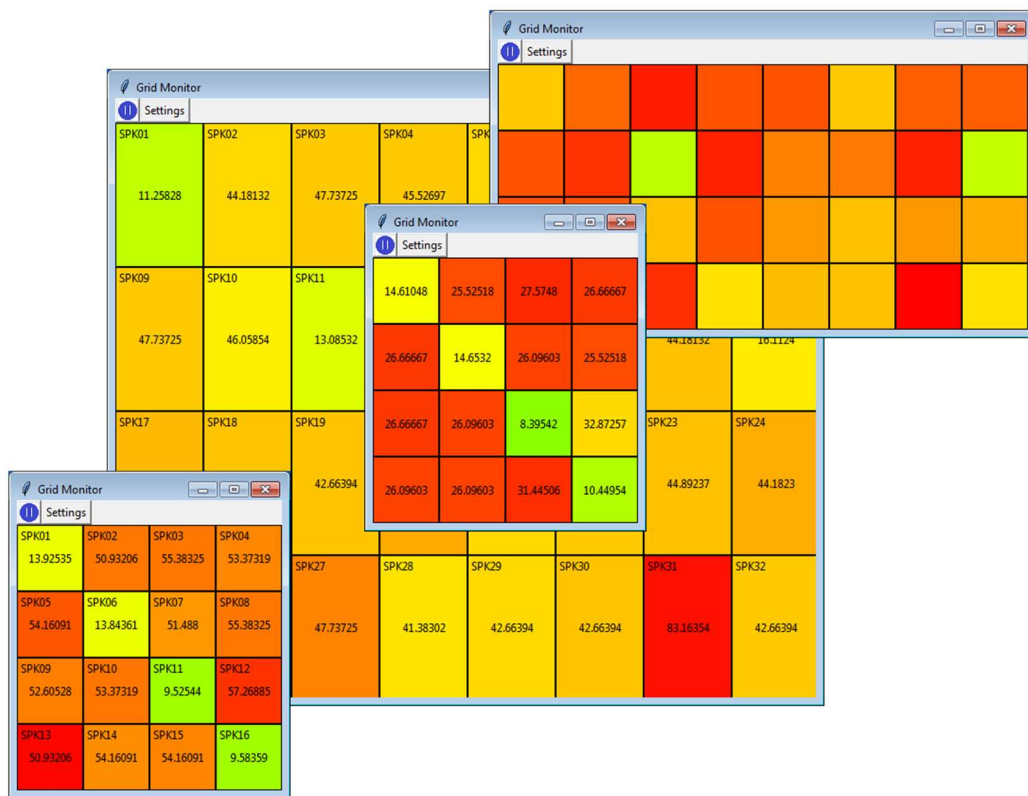
**Show Channel Names:** When enabled shows the channel names in the grid.

**Show ISI Value:** When enabled shows the ISI value in the grid.

When you click OK, the settings are applied and the settings window closes. The buffers are reset, and it might take a moment to update with new spike ISI data. The settings are also stored to disk in a .json file, and reloaded when GridMon is opened.

GridMon can also be resized by dragging a side or corner of the window.

GridMon in various configurations:



Similar to Strobe Monitor, GridMon generally follows a model-view-controller architecture. There is an additional view and controller class specific to the settings window.

The components of the view are separated into different classes across separate files. For example, the individual boxes of the grid are defined in a class called `Box` in the file `box_view.py`. The view that arranges the boxes into a grid is defined in a class called `Grid` in `grid_view.py`. This separation of concerns sacrifices simplicity in exchange for flexibility.

The `GridMonClient` class in `client.py` acts as the main model, but depends on a few other model classes. `GridMonClient` stores an instance of a `ChannelsSpikeBuffer` class, which contains one instance of a

ChannelBuffer class per channel. ChannelBuffer is where the timestamps are buffered and the ISI is calculated.

In ChannelBuffer, the built-in deque class is used to store spike timestamps. This is a useful class for buffering timestamp data because it automatically will remove elements in the buffer over a pre-determined buffer size. To remove spikes that are older than the “Seconds to Average” value, the client has to know the “current time” in OmniPlex. When you use the `opx_wait()` function to block (pause execution) until new data is available in OmniPlex, you can use the `get_last_wait_event_time()` function to get the time when new data was returned.

### **Technical Details**

PyOPXClient uses the Python ctypes module to wrap (expose) functions in OPXClient.dll, which was released as part of the C/C++ Client Development kit.

**pyopxclientlib.py** contains classes and methods that wrap the functions in OPXClient.dll. The main class is called PyOPXClient (not to be confused with the package name).

**pyopxclientapi.py** is a higher-level API that provides a friendlier (and ideally more “Pythonic”) way of using the functions defined in pyopxclientlib.py. The main class is called PyOPXClientAPI.

PyOPXClientAPI doesn’t improve on every single function in PyOPXClient. For example, there are functions in PyOPXClient for converting a channel in a source number to the old-style linear channel number, but these aren’t in the PyOPXClientAPI class for brevity, and because the old-style channel numbering isn’t recommended.

There is, however, a way to call any function in the PyOPXClient class. An instance of the PyOPXClientAPI class has a class variable called `opx_client`, which is an instance of the PyOPXClient class.

```
client = PyOPXClientAPI()
client.opx_client.linear_chan_and_type_to_source_chan(4, 32)
```

The code above shows how to call a function directly wrapped by the PyOPXClient class, but not exposed by the higher level PyOPXClientAPI class.

### **Additional Considerations in Client Programming**

This section contains more advanced material which can be skipped on a first reading, or until you are comfortable with the basics of writing and running clients.

#### *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.

### **Additional Considerations in Client Programming (Python-specific)**

PyOPXClient has been tested with a 512 channel OmniPlex system (currently the maximum number of channels). It performs well when collecting spike/event timestamps, but can struggle to keep up when trying to gather that many channels of continuous data and spike waveforms. On a 512 channel system, there are 1,568 channels of continuous data (512 wideband, 512 continuous spike, 512 LFP, and 32 auxiliary analog input). If you need continuous data on a high channel count system, it's suggested to use the `exclude_source()` function to prevent sources you don't require from being transferred to the client.

There is a fair bit of data re-organization in the data acquisition functions in the PyOPXClient class in the `pyopxclientlib` module. This isn't strictly necessary and could contribute to slow execution in higher channel count systems. Advanced users that understand how the `ctypes` module works should be able to use the existing wrapper code as an example of how to directly access the functions in `OPXClient.dll`, and write functions more custom to their application that may run faster.

### **Future Plans**

Because the Python Client SDK is the newest Client SDK (preceded by the C/C++ and Matlab SDKs), there are likely many improvements that will be made over time.

More examples, and more API-level functions (covering more functions exposed by the direct wrapper) are planned.

If you have any bugs, comments, or suggestions, please contact us at [support@plexon.com](mailto:support@plexon.com)