

XIPPPY

User Manual

Version 1.06
March 2025

Ripple Neuro
2056 East 1100 South
Salt Lake City, UT 84106 USA

1-800-380-5800
1-801-413-0139

CONTENTS

Contents 2

Overview 4

Installation 4

Functionality 4

- Start Xipppy 5
- Close Xipppy 5
- Xipppy Context Manager 5
- Time 6
- Add operator 6
- Trial 7
- List Elec 8
- Informational Functions 8
  - Front End Info Functions 8
  - Ripple processor info Functions 9
  - XIPP Info Function 9
- Signal Functions 10
  - Signal Status 10
  - Signal Set 10
  - Signal Save Functions – *Upcoming Feature* 11
  - Additional Signal Functions 11
- Filter 12
  - List 12
  - Set 13
  - Set Custom 13
  - Get description 14
- Cont 15
  - Status 16
- Data Functions 17
  - Spike Data 18
  - Stim Data 18
- Digin 18
- Digout 19
- Spike-Thresh 19
- Stim 20

Stim Enable 20
Stim Res 21
StimSegment 22
StimSeq 23
Stim Exhaust – *Upcoming Feature* 29
Fast Settle 30
Get choices 30
Set 30
Get durations 31
Impedance 31
Transceiver Enable 31
Sensor 32
Sensor Class 32
Battery 32
Processor 32
Button get 32
Example Scripts 32
Legacy functions 33

OVERVIEW

Xipppy is the Python API for interfacing with Ripple processors. The name is derived from the communication protocol used by Ripple processors and Trellis known as the eXtensible Instrument Processing Protocol (XIPP). Using Xipppy, most of the functionality available in Trellis can be performed programmatically by writing simple Python scripts. Xipppy then expands beyond the capabilities of Trellis, allowing for more precise control of the data streams, stimulation, and recording. Having these capabilities in Python allows for simple integration of post-processing, visualization, and custom programs for automation or other experiment requirements.

INSTALLATION

Installation files for Windows, Mac, and Ubuntu are included with the Trellis application files. These will have a ".whl" extension, and the applicable operating system and python version are noted in the filename. (e.g., 'xipppy-0.16.19-cp38-cp38-win.whl' is the installation file for Windows operating systems running python 3.8) If a different version than what is included is required, please reach out to support@rppl.com.

To install Xipppy, follow the instructions below:

1. Open a console (Windows) or terminal (Mac or Linux).
2. Change the operating directory to the folder containing the installation file.
   For example, "`cd C:\Program Files (x86)\Ripple\Trellis\Tools\Xipppy\`"
3. Install the file using pip.
   For example, "`pip install xipppy-0.16.10-cp38-cp38-win.whl`"

Note that as written above, Xipppy will be installed into your default python environment. Make sure to use the python environment that will be used with Xipppy.

FUNCTIONALITY

Xipppy is a library containing a variety of functions to control the processor and Front Ends. This includes opening and closing the connection to the processor, retrieving data from specific streams, delivering stimulation, and more. Each function is detailed in the following sections with information on syntax and examples of how to call the function.

Notes:
Before using any other library feature, call open() and call close() when done. There can only be one connection open at a time; it may sometimes be useful to close and re-open the network connection.

Most functions will raise a XippPyException on error. The underlying library does not offer useful information on the cause of an error in most cases, so these (unfortunately) tend to lack additional information.

Property setting functions, like ‘stim_enable_set’, ‘signal_set’, or ‘filter_set’, are asynchronous- the change will not take effect immediately since it must be sent across the XIPP connection. If it is important that a property change be applied before the program continues, you should poll for the desired value.

All following examples in this manual assume the following code has been ran:

```
>>> import xipppy as xp
```

Start Xipppy

The ‘xipppy._open’ function opens a connection between the Ripple processor and Xipppy. This connection between the Ripple processor and Xipppy is necessary for all subsequent Xipppy functionality.

Once the connection is established, multiple calls to ._open() will have no effect. By default, the connection is opened in UDP mode. If your processor is configured for TCP mode, use ‘xipppy._open(use_tcp=True)’.

Usage

```
xipppy._open()
```

Examples

```
try:
    udpMode = xp._open()
    print('Connected through UDP')
except:
    print('Failed to connect through UDP \nAttempting TCP')
    try:
        tcpMode = xp._open(use_tcp=True)
        print('Connected through TCP')
    except:
        raise Exception('Could not connect to processor. \nMake sure power LED is
connected and try again')
```

Close Xipppy

The ‘xipppy._close’ function immediately closes any open connections to the Ripple processor. In addition, all cached data and other buffers are cleared.

Usage

```
xipppy._close()
```

Xipppy Context Manager

Instead of using ‘xipppy._open’ and ‘xipppy._close’, a context manager for Xipppy is also available. ‘with xipppy.xipppy_open():’.

'xipppy._close()' should not be called from within a xipppy_open context. If your processor is configured for TCP mode, use 'with xipppy.xipppy_open(use_tcp=True):'.

For example, to simply test the connection to the NIP:

```
>>> with xp.xipppy_open():
...     pass
```

This context is reentrant, so the following is valid (exactly one call to open and close is made).

```
>>> def print_nip_time():
...     with xp.xipppy_open():
...         print(xp.time())
>>> with xp.xipppy_open():
...     print_nip_time()
```

The reentrant property of this context is useful if you are building complex libraries on top of Xipppy. You can safely wrap function calls that make Xipppy calls in this context without having to detect if Xipppy is already open.

Time

The 'xipppy.time' function retrieves the most recently received Ripple processor time, which is the number of clock cycles as 30 kHz (33.3 µs per cycle) that have occurred since the Ripple processor was started.

Usage

```
xipppy.time()
```

Examples

```
print('Time elapsed after Ripple processor start up: %s min '%(xp.time()/30000/60))
```

After connecting to Ripple processor, in this example, the running time of the Ripple processor is calculated in minutes.

Add operator

The 'xipppy.add_operator' function adds a Trellis operator ID. This function must be called with a valid Trellis operator before the 'xipppy.trial' function can be used if the Ripple processor is in TCP mode. If the processor is in UDP mode, 'xipppy.add_operator' does not need to be used and will throw an error.

Usage

```
xipppy.add_operator(oper_addr, connection_policy=None, timeout=None)
```

Input	oper_addr	The last octet of the IPv4 address of the computer running Trellis, which can be read from the Network Info panel in the main Trellis window.
Input	connection_policy	Optional. Set to integer 0-3 for the following connection policies or set to None to use the default.

		ConnectionPolicy.DEFAULT (None), ConnectionPolicy.PREFER_WIRED (0), ConnectionPolicy.PREFER_WIRELESS (1), ConnectionPolicy.WIRED_ONLY (2), ConnectionPolicy.WIRELESS_ONLY (3).
Input	timeout	Optional. TCP connection timeout in milliseconds.

Examples

```
xp.add_operator(129)
```

After opening Trellis and connecting to the Ripple processor in TCP mode, in this example, the Trellis operator ID 129 is added with default settings.

Trial

The 'xipppy.trial' function allows for the control of data recording through open instances of Trellis. **Note: 'Enable Remote Control' must be selected in Trellis File Save before Xipppy can control Trellis File Save options.**

Usage

```
trial_descriptor = xipppy.trial(oper=None, status=None, file_name_base=None,
auto_stop_time=None, auto_incr=None, incr_num=None)
```

Input	Oper	Required. Operator ID. By default, oper should be set to 129
Input	status	Optional. Recording status, must be one of 'recording', 'stopped', or 'paused'. If not specified, this parameter will not be changed.
Input	file_name_base	Optional. Full path of base file name for recording. Recorded files will be named file_name_base.nev, file_name_base.ns2, etc. If not specified, this parameter will not be changed.
Input	auto_stop_time	Optional. Time to automatically stop recording(s) in seconds. If not specified this parameter will not be changed. If set to 0, auto stop will be disabled.
Input	auto_incr	Optional. Boolean. If set to True, output file name will be appended by a recording index. The appended string will always be four digits long, padded to the left with zeroes. If set to False auto increment will be disabled.
Input	incr_num	Optional. Current value of increment number. If not specified, this parameter will not be changed.
Output	trial_descriptor	Tuple describing Trellis file recording. The parameters returned are (status, file_name_base, auto_stop_time, auto_incr, incr_num)

Examples

```
>>> print(xp.trial(oper=129))
('stopped', 'C:\\Users\\richard\\Trellis\\dataFiles\\20240729\\datafile', 0, True, 7)
```

This example polls the current recording parameters of the Trellis instance with ID 129. The processor is currently not recording.

```
>>> xp.trial(oper=129, status='recording', file_name_base=None, auto_stop_time=30,
auto_incr=None, incr_num=None)
('recording', 'C:\\Users\\richard\\Trellis\\dataFiles\\20240801\\datafile', 30, True,
7)
```
This example starts a recording that will stop automatically after 30 seconds without modifying any of the other recording parameters.

List Elec

The 'xipppy.list_elec' function returns a sequence of electrode IDs for the current Ripple processor setup. Channels of Front Ends attached to port A of the Ripple processor are numbered 0-127, then 128-255 for port B, etc. Analog input channels are numbered 10240-10269 in the following order: SMA inputs 1-4, micro-D grouped inputs 1-23, then the 2 line-level audio inputs.

Note: The 'max_elecs' input is set to 256 by default.

Usage

```
output=xipppy.list_elec(fe_type, max_elecs)
```

Input	fe_type	String corresponding to Front End type. Possible types are 'all', 'stim', 'micro', 'nano', 'surf', 'EMG', 'ecog', 'eeg', or 'analog'. Note: 'micro' is used for Micro and Micro HV Front Ends, and 'surf' is used for Surf S and Surf D Front Ends.
Input	max_elecs	Optional. Maximum number of electrodes to return. If there are more electrodes than specified in this input, the remaining will be omitted.
Output	output	Array of zero-indexed electrode IDs.

Example

```
>>> xp.list_elec('micro', 16)
array('I', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
```
This returns the first 16 electrode IDs for a 32 channel Micro2 Front End.

Informational Functions

Front End Info Functions

The functions 'xipppy.get_fe' and 'xipppy.get_fe_streams' return information regarding the current Ripple processor setup. 'xipppy.get_fe' returns the Front End index of the requested electrode. 'xipppy.get_fe_streams' returns a list of stream types supported by a given electrode per Front End. 'xipppy.get_fe_version' returns the R number for the Front End for a given electrode.

Usage

```
output=xipppy.get_fe(elec)
```

Input	elec	Zero-indexed electrode ID

Output	output	Number corresponding to Front End index for requested elec. If no Front End is using electrode, will raise exception 'no front end found for electrode (requested electrode index)'.

```
output=xipppy.get_fe_streams(elec, max_streams=32)
```

Input	elec	Zero-indexed electrode ID
Input	max_streams	Maximum number of strings to return.
Output	output	List of stream types electrode is capable of utilizing. Types include 'raw', 'stim', 'hi-res', 'lfp', and 'spk'

```
output=xipppy.get_fe_version(elec, max_size=1024)
```

Input	elec	Zero-indexed electrode ID
Input	max_size	Maximum size of string
Output	output	String corresponding to the product number, serial number, hardware, and software versions of the Front End. Eg. 'R02003-0226v6.5'

Ripple processor info Functions

The functions 'xipppy.get_nip_serial' and 'xipppy.get_nipexec_version' provide information on the Ripple processor and the software version it is currently running.

Usage

```
output=xipppy.get_nip_serial(max_size=1024)
```

Input	max_size	Optional. Maximum size of string.
Output	output	String corresponding to Ripple processor serial number, e.g., 'R04500-0014'.

```
output=xipppy.get_nipexec_version(max_size=1024)
```

Input	max_size	Optional. Maximum size of string
Output	output	String corresponding to Ripple processorexec version, e.g., '1.14.5+144'.

XIPP Info Function

The function 'xipppy.get_version' provides information on the XIPP protocol version, the XIPPLib version, and the Xipppy version.

Usage

```
output=xipppy.get_version()
```

Output	output	Dictionary of software and protocol versions. Keys are 'xipp' – XIPP protocol version e.g. '0.9', 'xipplib' – underlying C library version

		e.g. '0.9.1+8X', ‘xipppy’ – xipppy Python module version e.g., '0.16.8'.

Signal Functions

Signal Status

The ‘xipppy.signal’ function retrieves information about which streams are available for a given electrode and which streams are enabled.

Note: Only the ‘stim’ and ‘spk’ data streams can be enabled and disabled on a per-channel basis. All other data streams are enabled and disabled on a per-Front End basis when any channel is used for the ‘elec’ input.

Usage

```
output=xipppy.signal(elec, stream_ty)
```

Input	elec	Zero-indexed electrode ID
Input	stream_ty	Type of stream to get selection for (as from ‘xipppy.get_fe_streams’). Examples: ‘raw’, ‘stim’, ‘hi-res’, ‘lfp’, ‘spk’.
Output	output	Boolean indicating whether the stream of the given type on the given electrode is selected

Signal Set

The ‘xipppy.signal_set’ function enables data streams on the Ripple processor.

Usage

```
xipppy.signal_set(elec,  stream_ty, val)
```

Input	elec	Zero-indexed electrode ID
Input	stream_ty	Type of stream to set selection for (as from ‘xipppy.get_fe_streams’). Examples: ‘raw’, ‘stim’, ‘hi-res’, ‘lfp’, ‘spk’.
Input	val	Required. Possible options: True (activates stream), False (deactivates stream).

Examples

```
>>> xp.signal(1,’spk’)
```

Check spike data streas for electrode 2.

```
>>> xp.signal_set(1,’spk’,False)
```

Turn off spike data streams for electrodes 2.

```
>>> xp.signal(1,’spk’)
```

Check spike data streams for electrode 2.

Signal Save Functions – *Upcoming Feature*

The 'xipppy.signal_save' function returns a bool indicating whether the given signal and stream will be saved on the processor.

Usage

```
output=xipppy.signal_save(elec, stream_ty)
```

Input	elec	Zero-indexed electrode ID
Input	stream_ty	Type of stream to set selection for (as from 'xipppy.get_fe_streams'). Examples: 'raw', 'stim', 'hi-res', 'lfp', 'spk'.
Output	output	Boolean indicating whether the given signal and stream will be saved on the processor

The 'xipppy.signal_save_set 'function sets the file save selection on the processor for the electrode and stream.

Usage

```
xipppy.signal_save_set(elec, stream_ty, val)
```

Input	elec	Zero-indexed electrode ID
Input	stream_ty	Type of stream to set selection for (as from 'xipppy.get_fe_streams'). Examples: 'raw', 'stim', 'hi-res', 'lfp', 'spk'.
Input	val	Boolean indicating whether the given signal and stream will be saved on the processor

Additional Signal Functions

There also exists a subcategory of functions that are specific to the signal stream of interest. These functions work as the 'xipppy.signal' and 'xipppy.signal_set' functions work but specify the stream of interest (and therefore remove one of the inputs required to run). A comprehensive list of these functions can be found below:

Additional Status Functions

```
xipppy.signal_raw(elec)
xipppy.signal_lfp(elec)
xipppy.signal_spk(elec)
xipppy.signal_stim(elec)
```

Additional Set Functions

```
xipppy.signal_set_raw(elec, val)
xipppy.signal_set_lfp(elec, val)
xipppy.signal_set_spk(elec, val)
xipppy.signal_set_stim(elec, val)
```

Filter

The 'xipppy.filter' function displays the status of all the filtered streams and allows the selection of filters as well as the application of custom filters designed in Python. Each data stream, aside from the 'raw stream, is filtered online on the Ripple processor before being published on the instrument network. Each filtered stream may have one band pass style filter, as well as an independent notch or comb style filter to target 60 Hz noise and harmonics (or 50 Hz for EU customers).

The filter command has a number of subfunctions including 'xipppy.filter_list_names', 'xipppy.filter_list_selection', 'xipppy.filter_set', and 'xipppy.filter_set_custom'. Each of these is discussed in the following sections.

In Python, the preset filters are represented by a filter index that matches the 'sel' output from the xipppy.filter_list_selection. The following table shows the preset filter index/sel value and the high and low cutoff values (Hz) for each filter type.

sel	hires	hires notch	lfp	lfp notch	spike
0	1-125	Off	1-125	Off	250-7500
1	1-175	60	1-175	60	500-7500
2	1-250	60/120/180	1-250	60/120/180	750-7500
3	1-500	50	15-375 EMG	50	Custom
4	1-500 (HP	50/100/150	Custom	50/100/150	
5	15-375 EMG	Custom		Custom	
6	Custom				

List

The 'xipppy.filter_list' subfunctions return the types of filters available, the details of the preset filters for each type of filter, and which filter is currently selected by the Ripple processor. 'xipppy.filter_list_names' returns a list of available filter typers for a specified electrode. 'xipppy.filter_list_selection' returns a tuple (sel,nfilt) where sel is the index of the selected filter and nfilt is the number of filters matching the input parameters.

Usage

```
filt_names = xipppy.filter_list_names(elec, max_names =32)
```

Input	elec	Required. Zero-indexed electrode ID
Input	max_names	Optional. Maximum number of type strings to output. Any more than this number will be truncated.
Output	filt_names	List of strings returning possible filter types.

```
(sel, nfilt) = xipppy.filter_list_selection(elec, filter_type)
```

Input	elec	Required. Zero-indexed electrode ID
Input	filter_type	Required. Name of filter to list. Options include 'hires', 'hires notch', 'lfp', 'lfp notch', or 'spike'.

Output	sel	An integer reflecting the currently selected index of the selected filter for the specified 'filter_type'.
Output	nfilt	An integer reflecting the number of filters matching the input parameters

Examples

```
>>> xp.filter_list_names(2)
['hires', 'hires notch', 'lfp', 'lfp notch', 'spike']
```

This example returns the possible filters for electrode 3.

```
>>> xp.filter_list_selection(2,'spike')
(0, 4)
```

In this example, the currently selected index of the 'spike' filter and the number of available 'spike' filters are retrieved for electrode 3.

Set

The 'xipppy.filter_set' subfunction allows the end user to choose among the present filters on the Ripple processor for each filter type.

Usage

```
xipppy.filter_set(elec, filter_typ, filter_index)
```

Input	elec	Required. Zero-indexed list of desired electrodes.
Input	filter_type	Required. Name of filter to list. Options include 'hires', 'hires notch', 'lfp', 'lfp notch', or 'spike'.
Input	filter_index	An integer reflecting the index of the desired filter for the specified 'filter_type'.

Examples

```
>>> xp.filter_set(2, 'hires', 1)
```

In this example, the hires stream's filter for electrode 3 is changed to sel 1, or 1-175 Hz.

```
>>> xp.filter_set(129, 'lfp', 3)
```

In this example, the lfp stream's filter for electrode 130 is changed to sel 3 or 15-275 Hz EMG.

Set Custom

The 'xipppy.filter_set_custom' subfunction allows for custom filters designed by an end user to be added to any of the currently available filters. The custom filter is always the last filter of the selectable filters. To enable it, first the custom filter must be saved, with the 'xipppy.filter_set_custom' command, then selected, with the 'xipppy.filter_set' command. A filter structure must be created by the user that contains fields matching those returned in the filter structures returned from the 'xipppy.filter_get_desc' subcommand. The actual filter coefficients must be specified using second order sections (SOS) matrices. See the below examples for more information.

Usage

```
xipppy.filter_set_custom(elec, filter_type, filt_desc)
```

Input	elec	Required. Zero-indexed electrode ID
Input	filter_type	Required. Name of filter to modify.
Input	filt_desc	SosFilterDesc class object describing filter selection for given filter type. The filter_struct must have the following fields: 'label', 'center', 'lowCutoff', 'highCutoff', 'centerOrder', 'centerFlags', 'lowOrder', 'lowFlags', 'highOrder', 'highFlags', 'maxStages', 'numStages', 'stages'. NOTE: 'numStages' must match the number of SosStage objects in 'stages' and must not exceed 'maxStages'.

Get description

The 'xipppy.filter_get_desc' will return the SosFilterDesc object describing the requested filter, or None.

Input	elec	Required. Zero-indexed electrode ID
Input	filter_type	Required. Name of filter. Options include 'hires', 'hires notch', 'lfp', 'lfp notch', or 'spike'.
Input	filter_index	An integer reflecting the index of the desired filter for the specified 'filter_type'.
Output	filt_desc	SosFilterDesc class object describing filter selection for given filter type. The filter_struct must have the following fields: 'label', 'center', 'lowCutoff', 'highCutoff', 'centerOrder', 'centerFlags', 'lowOrder', 'lowFlags', 'highOrder', 'highFlags', 'maxStages', 'numStages', 'stages'. NOTE: 'numStages' must match the number of SosStage objects in 'stages' and must not exceed 'maxStages'.

Examples

```
>>> filter = xp.filter_get_desc(0, 'hires', 3)
>>> xp.filter_set_custom(0, 'hires', filter)
>>> custom_filter = xp.filter_get_desc(0, 'hires', 6)
>>> custom_filter.label
'0.3-250 Hz'
>>> xp.filter_set(0, 'hires', 6)
```

In this example, the third filter from the hires stream is taken and set as the custom filter. Then the custom filter is enabled.

```
>>> sos = scipy.signal.butter(
        3, np.array([30, 100])/(2000/2), 'bandpass', output='sos'
    )
>>> stages = []
>>> stages.append(xp.SosStage(sos[0, 0], sos[0, 1], sos[0, 4], sos[0, 5]))
>>> stages.append(xp.SosStage(sos[1, 0], sos[1, 1], sos[1, 4], sos[1, 5]))
>>> stages.append(xp.SosStage(sos[2, 0], sos[2, 1], sos[2, 4], sos[2, 5]))
>>> custom_filter = xp.filter_get_desc(0, 'hires', 6)
>>> custom_filter.label = 'myFilter'
>>> custom_filter.stages = stages
>>> custom_filter.lowCutoff = 30
>>> custom_filter.highCutoff = 100
>>> custom_filter.center = 0
```

```
>>> custom_filter.centerOrder = 0
>>> custom_filter.centerFlags = 0
>>> custom_filter.lowOrder = 3
>>> custom_filter.lowFlags = 0
>>> custom_filter.highOrder = 3
>>> custom_filter.highFlags = 0
>>> xp.filter_set_custom(0, 'hires', custom_filter)
>>> xp.filter_set(0, 'hires', 6)
```

In this example, a filter is designed with SciPy's butterworth function and the SOS coefficients are calculated. The required filter structure is built and then set with the 'custom' command.

Cont

The 'xipppy.cont' function returns the data from continuous data streams such as lfp, raw, or hi-res. To facilitate this, xipppy maintains a five second circular buffer for all enabled streams. Continuous data may be retrieved directly off the top of the buffer or by a known Ripple processor timestamp. Recording electrode channels are numbered 0-511 (0-127 on port A, 128-255 on port B, etc.) and analog input channels are numbered 10240-10269.

The subfunctions 'cont_raw', 'cont_lfp', 'cont_hires', 'cont_hifreq', 'cont_emg', and 'cont_status' specify the streams from which data is being requested.

Usage

```
[data, timestamp] = xipppy.cont_x(npoints, elecs, start_timestamp)
```

Where 'x' denotes the appropriate stream subfunction of choice

Input	npoints	Number of datapoints to retrieve per electrode.
Input	elecs	Required. Zero-indexed list of desired electrodes.
Input	start_timestamp	Optional. If specified, 'cont' will return data starting from the given Ripple processor timestamp. If empty, the most recent data will be retrieved. The timestamp is based on Ripple processor 30 kHz clock ticks.
Output	data	Array containing requested data (µV for neural data, mV for analog I/O inputs). The returned dimensions will be 1 x npoints*number of elecs. The first npoints will correspond to the first elec, next npoints to the second elec, and so on. Note: LFP streams sample at 1kHz, hi-res streams sample at 2 kHz, and raw streams sample at 30 kHz.
Output	timestamp	Timestamp for the first sample of retrieved data. The timestamp is based on Ripple processor 30 kHz clock ticks. If start_timestamp is not specified, the output timestamp is always 0.

Examples

```
data, ts = xp.cont_raw(5000, [1,2])
```

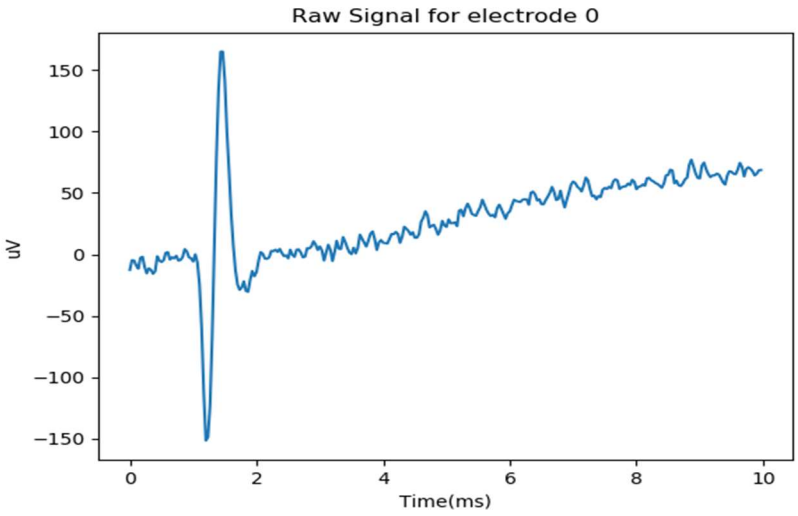
Returns the most recent 10000 points of raw data from Front End channels 2 and 3, along with ts=0. The first 5000 points correspond to channel 2, and the next 5000 to channel 3.

```
data, ts = xipppy.cont_lfp(250, [6, 8, 19], xipppy.time())
```

Returns 750 points of lfp data for Front End channels 7, 9, and 20, along with the Ripple processor timestamp. Data points 0-249 are for channel 7, 250-499 for channel 9, and 500-749 for channel 20.

```
import xipppy as xp
import numpy as np
import matplotlib.pyplot as plt

if __name__ == '__main__':
    with xp.xipppy_open():
        fs_clk = 30000
        xp.signal_set(0, 'raw', True)
        elec_0_raw = xp.cont_raw(300, [0], 0)
        t = np.arange(0, 300000 / fs_clk, 1000/fs_clk, dtype=np.float32)
        plt.plot(t, elec_0_raw[0])
        plt.xlabel('Time(ms)')
        plt.title('Raw Signal for electrode 0')
        plt.show()
```



Status

The sub-function ‘xipppy.cont_status’ retrieves the MIRA status update at a 2 kHz rate. Every clock cycle the MIRA Front End publishes status information concerning the implant connectivity and sensor information as such:

Channel	Name	Units	description
0	counter		8-bit value that increments with each new implant packet [0-255]
1	xcvr_coil_v	V	Transceiver coil voltage. Indicates the level of driven RF wireless power field. [~1.8-3.3]
2	xcvr_coil_a	A	Transceiver coil current. Indicates the power consumed by the transceiver used to power the implant.
3	xcvr_input_v	V	Transceiver input voltage. Indicates the voltage supplied to the transceiver.
4	xcvr_input_a	A	Transceiver input current. Indicates the total power consumed by the transceiver.
5	xcvr_temp	Deg C	Transceiver internal temperature. Meant to approximate the transceiver’s patient-facing surface temperature.

6	xcvr_temp_offboard		(Deprecated
7	servo_state		0 = idle – coil off, waiting for user command or button press. 1 = Comm init – enable communication power supplies and amplifiers. 2 = Coil init – enable coil voltage supply and individual coil drive signals. 3 = Servoing – standard link state where transceiver is modulating coil power output based on implant voltage level. 4 = Searching off – coil off. 5 = searching on – coild on mat max drive
8	impl_serial		Implant Serial number
9	impl_deviceid		Implant model number
10	impl_temp	Deg C	Implant temperature
11	impl_humidity		(deprecated)
12	impl_v	V	Rectified implant voltage. Indicates health of received wireless power.
13	impl_ver_hw		Implant hardware version
14	impl_ver_fw		Implant firmware version

Usage

```
xipppy.cont_status(npoints, elecs, start_timestamp)
```

Input	npoints	Number of datapoints to retrieve.
Input	elecs	Required. Zero-indexed list of desired electrodes. (if it should be channel: a list of channels to retrieve. See table above. All channels are returned as float arrays even if they are fundamentally integer quantitities)
Input	start_timestamp	Optional. If specified, 'cont' will return data starting from the given Ripple processor timestamp. If empty, the most recent data will be retieved. The timestamp is based on Ripple processor 30 kHz clock ticks.

Data Functions

Both the 'xipppy.spk_data' and 'xipppy.stim_data' functions return the count and spike information for requested electrodes. For each electrode, data for up to 1024 spike and stim markers are stored in a circular buffer. When either function is called, the spike/stim marker data is retrieved, and the buffers are flushed. If more than 1024 spike or stim markers are received between consecutive function calls, only the most recent 1024 spikes are returned. However, the returned spike counts will not be capped and will reflect the current number of triggered spike events. Spike and stim marker data can be retrieved for several electrodes at a time, and as such is returned in the SegmentDataPacket class. The class has the following attributes: timestamp (.timestamp), class ID (.class_id), and 52-sample waveform (.wf).

Spike Data

The xipppy.spk_data function returns the number of spikes and SegmentDataPacket class objects for each spike.

Usage

```
count, data = xipppy.spk_data(elecs, max_spk=1024)
```

Input	elecs	Required. 1-valued int of desired electrode.
Input	max_spk	Optional. Maximum number of spikes to output. Maximum number allowed is total length of circular buffer (1024), and anything below will be truncated.
Output	count	Array holding the number of spikes received since last call
Output	data	List of SegmentDataPacket classes with attributes: timestamp (.timestamp), class ID (.class_id), and 52-sample waveform (.wf).

Examples

```
Count, data = xp.spk_data(1)
```

Retrieves the number of spikes and most recent SegmentDataPacket classes for electrode 2.

Stim Data

The xipppy.stim_data function returns the number of spikes and SegmentDataPacket class objects for each stimulation waveform.

Usage

```
count, data = xipppy.stim_data(elec, max_spk=1024)
```

Input	elec	Required. Zero indexed int of desired electrode.
Input	max_spk	Optional. Maximum number of spikes to output. Maximum number allowed is total length of circular buffer (1024), and anything below will be truncated.
Output	count	Array holding the number of spikes received since last call
Output	data	List of SegmentDataPacket classes with attributes: timestamp (.timestamp), class ID (.class_id), and 52-sample waveform (.wf).

Examples

```
count, data = xp.stim_data(5, 600)
```

Retrieves the number of spikes and last 600 SegmentDataPacket classes for electrode 6.

Digin

The ‘xipppy.digin’ function returns events triggered on the Digital I/O Front End. Like spike data, Xipppy maintains a circular buffer of 1024 digital events. This buffer is cleared when data is retrieved. If more than 1024 events are received by the Ripple processor between xipppy.digin calls, only the most recent 1024 triggered digital events are returned, though the returned count will always reflect the current number of triggered events. The output ‘events’ is a list of the SegmentEventPacket class. The class has the following attributes: timestamp (.timestamp), reason (.reason), SMA port 1 (.sma1), SMA port 2 (.sma2), SMA port 3 (.sma3), SMA port 4, (.sma4), and parallel (.parallel).

Usage

```
n, events = xipppy.digin(max_events = 1024)
```

Input	max_events	Optional. Maximum number of events to output. Maximum number allowed is total length of circular buffer (1024), and anything below will be truncated
Output	n	The number of digital events received since the last digin call.
Output	events	List of SegmentEventPacket classes containing digital event data. Attributes are 'timestamp', 'reason', 'sma1', 'sma2', 'sma3', 'sma4', and 'parallel'. The fields 'sma1' through 'sma4' and 'parallel' hold unsigned 16-bit integer values for the corresponding input. The 'reason' field is a bit mask showing which input triggered the digital event. 1 corresponds to the parallel port. 2, 4, 8, 16 refer to sma inputs 1, 2, 3, and 4 respectively. 32 refers to a digital output marker created when using the 'digout' command.

Digout

The 'xipppy.digout' function controls digital output ports on the digital I/O Front End. The supply voltage for the digital output pins is 3.3 V. 16-bit unsigned integers may be set on the ouput parallel port.

Usage

```
xipppy.digout(outputs, values)
```

Input	outputs	List of outputs to control. 0-3 controls SMA outputs 1-4. Specifying output 4 controls the output parallel port.
Input	values	List of values to set at ports. SMA Outputs 1-4 are binary. I.e., 0 will set them to 0, any positive value will set the port to the supply voltage. If the output parallel port is specified, the value may be a 16-bit unsigned integer. The length of values must equal the length of outputs.

Examples

```
xipppy.digout([1, 2], [1, 1])
time.sleep (0.001)
xipppy.digout([1, 2,], [0, 0])
```

Assuming output 1 was initially low, this set of commands, will approximate a TTL pulse. The width of the TTL will be determined by the resolution of the time.sleep command and may be system dependent.

Spike-Thresh

The 'xipppy.spk_thresh' function can be used to return or to set the low and high spike thresholds for the desired electrodes.

Usage

```
low_thresh, high _thresh = xipppy.spk_thresh(elec)
xipppy.spk_thresh_set(elecs, lower, upper)
xipppy.spk_thresh_set_lower(elecs, lower)
xipppy.spk_thresh_set_upper(elecs, upper)
```

Input	elec	Integer desired electrode number
Input	elecs	Zero-indexed array of desired electrodes.
Input	lower	Integer lower spike threshold in µV. This field must be negative.
Input	upper	Integer upper spike threshold in µV. This field must be positive.
Output	low_thresh	Integer the current 'low' spike threshold for the electrode.
Output	high_thresh	Integer the current 'high' spike threshold for the electrode.

Examples

```
low_thresh, high_thresh = xp.spk_thresh(3)
```

Retrieves the current low and high spike thresholds for electrode 4.

```
xp.spk_thresh_set_lower([4,10], -40)
```

Sets the low threshold on electrodes 5 and 11 to -40.

```
xp.spk_thresh_set([2], -60, 90)
```

Sets the low and high thresholds on electrode 3 to -60 and 90 respectively.

Stim

The stim module contains a combination of subfunctions with the capability to control the Ripple processor's global stim enable property, set the resolution of the stimulation, and create custom stimulation waveforms.

Stim Enable

Prior to running any stimulation, the global stim enable property state must be set to True. The 'xipppy.stim_enable' command checks the global enable state of stimulation. Depending the boolean state, 'xipppy.stim_enable_set(val)' allows for toggling of the state. This function may be used to immediately stop stimulation. By default, the Ripple processor's stimulation property will be set to false. In normal stimulation usage, Xipppy will set the Ripple processor's stimulation enable property to True (on), and after the stimulation has finished will set the property to False (off).

Usage

```
xipppy.stim_enable()
xipppy.stim_enable_set(val)
```

Input	val	Boolean. True to enable or False to disable. If left empty, Xipppy will return the current global state.

Examples

```
>>> xp.stim_enable()
False
```

Check global enable state. Here, stim is disabled.

```
>>> xp.stim_enable_set(True)
```
Toggle global enable state to on.

```
>>> xipppy.stim_enable()
True
```
Check global enable state. Stim has now been enabled.

Stim Res

For stimulating Front Ends, there is a 'xipppy.stim_set_res' subfunction that sets the stimulation current step size. This command is applied on a per-Front end basis; i.e. all channels on a particular Front End are set to the 'stim_set_res' stimulation step size by setting any electrode on that Front End. The current stimulation step size on an electrode can be checked by the 'xipppy.stim_get_res(elec)' subfunction. Each Front End type has up to 5 different step sizes specified 0-4 (up to 6 different step sizes specified 0-5 if using a Macro2+stim front end).

The following are the available stim resolutions for each Front End type:

Front End	0	1	2	3	4	5
Pico, Nano, Micro	1 uA/step	2 uA/step	5 uA/step	10 uA/step	20 uA/step	
Macro	10 uA/step	20 uA/step	50 uA/step	100 uA/step	200 uA/step	
Macro2	1 uA/step	2 uA/step	10 uA/step	20 uA/step	50 uA/step	120 uA/step

Note: The amplitude of stimulation is specified with unitless values between 0-100. The amount of current produced by each step is determined by the stimulation resolution set with this subfunction.

Note: When using the highest step size of 5 (or 6 for Macro2) step sizes, the range of stimulation output is only ±75 steps.

Note: When the Grapevine is initially powered on, the default setting of the lowest step size will be set.

Usage

```
output = xipppy.stim_get_res(elecs)
xipppy.stim_set_res(elecs, level)
```

Input	elec	Zero-indexed value of desired electrode. Gets/Sets resolution for entire Front End.
Input	level	Scalar value representative of the desired stimulation resolution 0-4 dependent on the specific Front End.
Output	output	Current stimulation resolution

Examples

```
>>> xp.stim_get_res(15)
0
```
Check the current stimulation resolution for channel 16 (the first Front End on port A). Here, it is set to 0 (lowest amplitude step).

```
>>> xp.stim_set_res(15, 3)
```
This sets the Front End associated with channel 16 (the first Front End on port A) to have a stimulation step-size equal to the fourth step size.

```
>>> xp.stim_get_res(15)
3
```
Check to see that the current stim resolution for channel 16 changed to 3.

```
>>> xp.stim_get_res(1)
3
```
Check to see the current stim resolution for channel 2 changed to 3. Since channel 2 is also part of the first front end on port A, changing the resolution for channel 16 also changed the resolution for rest of the channels on the Front End.

StimSegment

Stimulation segments define the stimulation waveform using a class structure. Multiple stimulation segments can be assembled to create a stimulation sequence. 'xipppy.StimSegment()' has up to seven parameters that define the waveform: length, amplitude, polarity, current enable, delay, fast settle, and amplifier select.

Usage

```
seg = xipppy.StimSegment(length, amplitude, polarity, enable=True, delay=0,
fast_settle=False, amp_select=1)
```

Stimulation Segment Parameters

A valid segment is a list of parameters separated by commas. For example the segment: 'xipppy.StimSegment(15000, 50, 1, enable=True, delay=0, fast_settle=False, amp_select=1)' creates a monophasic pulse of length 15000 samples (500 ms) with an integer amplitude of 50 steps, and a positive polarity. There will be current output with no delay and fast settle will not be activated. The following table outlines all the required fields in the structure and examples will follow on simple biphasic pulses.

length	Required. Integer length of the stimulation segment in units of Ripple processor clock cycles (33.33 µs). For example, if a length is 10 then this segment would be 333.3 µs long. Must be in range [0,65535].
amplitude	Required. Integer amplitude in stimulation output steps. The current per step depends on the current stimulation resolution (see 'stim_get_res'). The total current is stimulation resolution*integer amplitude.
polarity	Required. If polarity is set to -1, the stimulation segment will be cathodic. If 1, the stimulation will be anodic.
current_enable	Optional. Specifies whether the next transition is to zero output (interphase interval) or positive/negative stimulation (anodic/cathodic). If there is not stimulation in this period, this should be False. If current output is wanted, positive or negative, this should be True (Default).
delay	Optional. Time to delay a stimulation pulse in units 33.33 µs / 32 =1.04 µs. This number must be between 0-31. This parameter allows for the control of

	stimulation at a temporal resolution of almost 1 µs but is only valid for particular waveforms. This parameter may be used for interleaving stimulation.
fast_settle	Optional. Boolean that enables fast settling. Defaults to false (see 'fast_settle')
amp_select	Optional. Select the amplifier that is enabled during this segment. Default to NEURAL_AMP (1), alternative is STIM_LEVEL_AMP (0).

Examples

```
xipppy.StimSegment(10, 100, -1)
xipppy.StimSegment(10, 100, 1)
```

These commands would return the StimSegments needed to produce a simple bipolar pulse with durations of 333.33 µs and an integer amplitude of 100.

StimSeq

The 'xipppy.StimSeq' class is an advanced tool that allows for low-level control of stimulation via the Ripple processor and Stimulation Front Ends (e.g. Micro2+Stim). Careful use of this class can produce almost any stimulation waveform, with temporal resolution as small as 1 µs.

The construction of stimulation trains includes two main control blocks. The overall control word specifies the electrode, period of stimulation, and the number of times that the stimulation waveform will be repeated. The second control word defines the repeated waveform. Figure 1 illustrates the breakdown of the overall language.

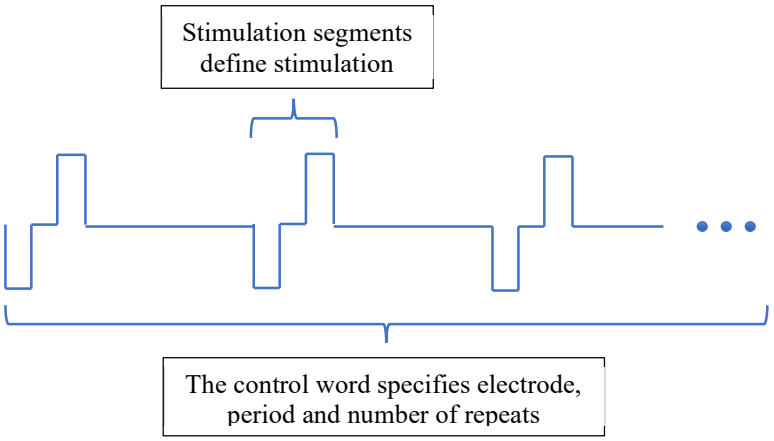


Figure 1. Control word and stimulation diagram

Stimulation will be processed by the Ripple processor as soon as it is received based on the action part of the control word. The handling of time critical pieces must be performed in Python.

In Xipppy, the control word consists of five fields 'elec', 'period', 'repeats', 'segments', and 'action', and is packed in a class structure. Each required field is defined in the following table.

elec	Scalar, zero-indexed electrode ID. Must be in range of [0, 511].
period	The period at which to repeat stimulation in units of 33.3 µs (on clock cycle at 30 kHz).
repeats	The number of repetitions of the stimulation. To perform stimulation this number must be at least 1. The number of repeats cannot be greater than 4095.
segments	List of segments that define the stimulation waveform. The user needs to list them out, this cannot just be a python list.

action	Optional. Command determining how the stimulation is processed on the Ripple processor. Must be an integer in range [0, 4]. 'STIM_IMMED' (0), 'STIM_CURCYC' (1), 'STIM_ALLCYC' (2), 'STIM_TRIGGER' (3), or 'STIM_AT_TIME' (4). If not specified 'STIM_IMMED' is used.

The 'action' field of the stimulation control word tells the Ripple processor how to process the stimulation command. This allows a user to immediately begin stimulation, queue stimulation, or execute stimulation at a defined Ripple processor time.

STIM_IMMED – Process the command immediately
This command will be processed as soon as the Ripple processor receives the command. However, in the case that the new command would interrupt an ongoing stimulation waveform and disrupt charge balancing, the new stimulation train will be placed after the current waveform. Using this command could potentially put two stimulation pulses very close together, causing an abrupt change in stimulation frequency. The STIM_IMMED command will clear any stimulation that is currently queued.

STIM_CURCYC – Process the command after current cycle
This command will interrupt any ongoing stimulation. However, instead of executing the new waveform immediately when it is received, the Ripple processor attempts to blend the two frequencies. The new stimulation pulses will be placed such that the time between successive pulses is never less than the updated frequency. If there is no ongoing stimulation when this command is received, STIM_CURCYC acts identically to STIM_IMMED.

STIM_ALLCYC – Queue stimulation after the current train
This command will queue the new stimulation pattern directly after the current pattern has run through its entire train. If consecutive patterns are sent to the Ripple processor with STIM_ALLCYC they will be queued as they are received. The max queue depth is eight patterns. If more than eight patterns are received for a given electrode, the Ripple processor will disable stimulation on that entire Front End.

STIM_TRIGGER – *Upcoming Feature*

STIM_AT_TIME – Apply stimulation pattern at a specified Ripple processor time
This command allows a user to specify a later Ripple processor time for when a stimulation pattern will be executed. This command utilizes the 'period' field in the stimulation command word differently than the commands above. As such, only a 16-bit unsigned may be used with the 'period' field of this command. The 'repeats' field of the command word is necessary but is ignored using STIM_AT_TIME; thus stimulation trains are not possible with this command outright, though they may be generated using multiple stimulation pulses and pauses in the command sequences or by sending regular STIM_AT_TIME commands and updating the time with regular intervals.

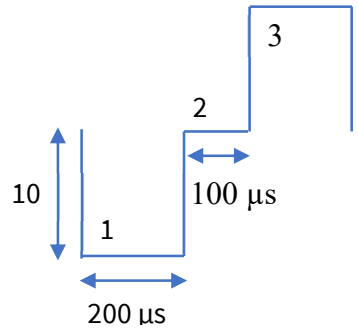
The incoming sequences will be executed when the value in the period field matches the lower 16-bits of the Ripple processor time, or when $t_{Ripple\ processor}$*&0xFFFF==period.*

Note: This requires that the Ripple processor receive a STIM_AT_TIME command no later than two seconds from the current Ripple processor time (see 'time' command on page 6 for reference). As

multiple commands are received, they will be queued similar to using the STIM_CURCYC and STIM_ALLCYC commands. Stimulation will be disabled on a Front End if a command is received too early to execute or if the queue depth is larger than eight.

The 'segments' field specifies the waveform itself with a list of StimSeg classes. Each element in the list specifies a period of the stimulation where the stimulation parameters are constant.

Examples
In this first example, the waveform described in Figure 2 is created with a frequency of 30 Hz and a train length of one second.



A. The first, cathodic phase of the waveform can be created with length set to 6, pol set to -1, and enable set to 1. All other fields may be set to zero.
B. The interphase interval can be created using length of 3 and setting amplitude to zero. All other fields may be set to zero.
C. The last, anodic phase can be produced by matching the word from 'A' only with the pol field set to 1.

Figure 2. A basic biphasic pulse pair can be constructed with three words in this example: A, B, and C.

```
import xipppy as xp
import time

elec = 0
res = 3

with xp.xipppy_open(use_tcp=True):

	stimList = xp.list_elec('stim')

	if xp.stim_get_res(0) <= 1: # change stim res
		xp.stim_enable_set(False)
		xp.stim_set_res(elec,res)

	xp.stim_enable_set(True)

	pseg = xp.StimSegment(6,100,-1)# create first phase (cathodic) segment for
	stimulation
	# duration of 200 us (6 clock cycles at 30 kHz), amplitude of 100, and
	negative polarity

	ipi = xp.StimSegment(3,0,1, enable=False) # creates inter-phase interval
	# duration of 100 us (3 clock cyclesat 30 kHz), amplitude of 0, enable set to
	false

	nseg = xp.StimSegment(6,100,1) #creates second, anodic phase
	# duration 200 us (6 clock cycles at 30 kHz), amplitude of 100, and positive
	polarity

	seq0 = xp.StimSeq(elec,1000,30,pseg,ipi,nseg) # create overall sequence header
	defining electrode, period, repeats
	# This will stimulate on electrode 1 at 30 Hz for one second (1000 ms)

	xp.StimSeq.send(seq0) #send sequence to Ripple processor
	time.sleep(0.1)
```

In the second example, the use of the 'delay' field is demonstrated to produce a biphasic pulse with phases narrower than the 30 kHz clock cycles. Here a symmetric biphasic pulse is created with 50 µs

pulse width. Because this requires a temporal resolution less than 33.3 µs, the 'delay' parameter will need to be used. Figure 3 illustrates the needed words and boundaries on the 30 kHz clock.

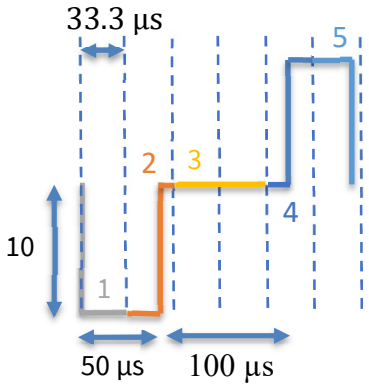


Figure 3. Illustration of the required stimulation words needed to create a biphasic pulse with 50 µs phases. As this requires a temporal resolution finer than 33.3 µs, the delay parameter will be used. The dashed lines represent the 30 kHz clock. Each new color shows where a new stimulation word is used.

```
import xipppy as xp
import time
import math

clock_cycle=1/30*1000 #single clock cycle in microseconds
delay_length=1/30*1000/32 #calculate the length of a single unit of delay in
microseconds
elec=0 #electrode num
res=3 #resolution (10 uA/step)

with xp.xipppy_open(use_tcp=True):

	stimList = xp.list_elec('stim')

	if xp.stim_get_res(0) <= 1: #change stim res
		xp.stim_enable_set(False)
		xp.stim_set_res(elec,res)

	xp.stim_enable_set(True)

	seg1=xp.StimSegment(1,100,-1) #seg1: 1 clock cycle worth of stim for the first
	50 us phase

	cath_remaining=50-clock_cycle #calculate remianing duration of the cathodic
	phase
	cath_delay=math.floor(cath_remaining/delay_length) #actual delay param must be
	integer between 0-31

	seg2=xp.StimSegment(1,100,-1,enable=False,delay=cath_delay) #seg 2: full
	cathodic phase and a bit of interphase interval. 'enable' field set to zero,
	sets pulse to start high and transition to off
```

```
seg3=xp.StimSegment(2,0,1,enable=False) #seg3: interphase interval is 100 us,
so add two more full clock cycles worth of stim for 66.6 us more

ipi_remaining=100-(clock_cycle-cath_delay*delay_length)-2*clock_cycle
#calculate how much more of inter-phase is remaining, being careful to account
for quantization of delay at the end of cathodic
ipi_delay=math.floor(ipi_remaining/delay_length)

seg4=xp.StimSegment(1,100,1,delay=ipi_delay) #seq4: enable field sets the
pulse to start off and transition to on

anod_remaining=50-(clock_cycle-ipi_delay*delay_length) #finish pulse with one
use of delay field
anod_delay=math.floor(anod_remaining/delay_length)

seg5=xp.StimSegment(1,100,1,enable=False, delay=anod_delay) #seg5: last
command word

seq0=xp.StimSeq(elec,1000,30,seg1,seg2,seg3,seg4,seg5)#create overall sequence
header defining electrode, period, repeats, and segments
#This will stimulate on electrode 1 at 30 hz for one second (1000 ms)

xp.StimSeq.send(seq0) #send sequence to Ripple processor

time.sleep(0.1)
```

In the third example, multi-channel synchronous stimulation is performed using ‘xp.StimSeq.send_stim_seqs’, which takes a list of StimSeq objects and packs them into an array before sending to the Ripple Processor. If the total size of all the stim sequences is less than 1500 bytes, the stim sequences will execute synchronously with precise timing.

```
import xipppy as xp
import time

def stimWaveform(stim_channel, pulse_width, stim_mag_steps, stim_res):
	# stim_channel = channel # to send stimulus to
	# pulse_width = width of each phase (units of clock cycles)
	# magnitude = magnitude of stim pulse (units of steps
	# stim_res = index of resolution of stim desired
	#(e.g.for nano, 1=1uA/step, 2=2uA/step, 3=5uA/step, 4=10uA/step, 5=20uA/step)

	xp.stim_enable_set(False)
	time.sleep(0.001)
	xp.stim_set_res(stim_channel,stim_res)

	# Enable stimulation on NIP
	xp.stim_enable_set(True)

	# Design stimulation waveform
	# Biphasic cathodic-first (single pulse)
	pseg = xp.StimSegment(pulse_width,stim_mag_steps,-1)
	ipi = xp.StimSegment(round(pulse_width/2),0,1, enable=False)
	nseg = xp.StimSegment(pulse_width,stim_mag_steps,1)
	seq0 = xp.StimSeq(stim_channel, 1000,1,pseg,ipi,nseg)

	return seq0

with xp.xipppy_open(use_tcp=True):

	stimList = xp.list_elec('stim')

	seq_arr = []
	elecs = [0, 15]

	for elec in elecs:
		seq_arr.append(stimWaveform(elec, 200, 50, 4))
	# This will stimulate on electrodes 1 and 16 for a single pulse.

	xp.StimSeq.send_stim_seqs(seq_arr) #send array of sequences to Ripple
	processor
	time.sleep(0.1)
```

Stim Exhaust – *Upcoming Feature*

For Macro+stim style FE's, the exhausting functionality allows for pulling stim electrodes through a resistor after stimulation.

Usage

```
ouput = xipppy.stim_get_exhaust(elec)
xipppy.stim_set_exhaust(elec, res_opt)
```

Input	elec	Zero-indexed value of desired electrode. Gets/Sets stim exhaust resistor value for entire Front End.
Input	res_opt	Stim exhaust resistor value.
Output	output	Current stim exhaust resistor value

Fast Settle

Xipppy can control the Ripple processor's fast settle capability, allowing for successful recordings within a millisecond of stimulation on neighboring electrodes. Fast settle may be set to occur directly after stimulation produced by the Ripple processor or triggered from digital events, useful if used with external stimulators. The fast settle period, or the length of time to settle the ADC, can also be set with this command.

The fast settle command includes two subfunctions: 'xipppy.fast_settle_get_choices' and 'xipppy.fast_settle_get_durations'.

Get choices

The subfunction 'xipppy.fast_settle_get_choices' gets options for fast settle for a desired front end.

Usage

```
xipppy.fast_settle_get_choices(fe, fs_type, max_names=32)
```

Input	fe	Required. Desired Front End. Must be in range [0,15].
Input	fs_type	Required. Fast settle type: either 'stim' or 'digin'.
Input	max_names	Optional. Maximum number of type strings to output. Any more thn this number will be truncated.

Examples

```
>>> xp.fast_settle_get_choices(0,'stim')
(1, ['None', 'Any Front End', 'Same Front Port', 'Same Front End'])
```

This example returns the four options for stim type fast settle for the front end in position 0.

Set

The function 'xipppy.fast_settle' sets fast settle triggering to occur from outgoing stimulation by the Ripple processor. Fast settle may be triggered by any stim electrode on the Ripple processor, by any stim electrode connected to the same front end port, or by a stim electrode on the same Front End. All Grapevine Front Ends support fast settle with the exception of the Surf D and EMG Front Ends.

Usage

```
xippy.fast_settle(fe, fs_type, item=None, duration=None)
```

Input	fe	Required. Desired front end. Must be in range [0,15].
Input	fs_type	Required. Fast settle type: either 'stim' or 'digin'.
Input	item	Optional. Index of fast settle option. None for default. If not specified the current source index is returned. These are 0-'none', 1-'any front end', 2-'same front end port, or 3-'same front end'.
Input	duration	Optional. Duration of fast settle in milliseconds. Default is 0.5 ms. Has to be in range [0.5 – 15.0]

Examples

```
>>> xipppy.fast_settle(0,'stim', item=2, duration=2)
```

This example sets the stimulation fast settle for the Front End in position 0 to ‘same front end port’ for a duration of 2 ms.

Get durations

The subfunction ‘xipppy.fast_settle_get_duration’ gets the current duration for fast settle for desired Front End.

Usage

```
output = xippy.fast_settle_get_duration(fe, fs_type)
```

Input	fe	Required. Desired front end. Must be in range [0,15].
Input	fs_type	Required. Fast settle type: either ‘stim’ or ‘digin’.
Output	output	The fast settle duration in milliseconds (ms)

Examples

```
>>> xp.fast_settle_get_duration(0,’stim’)
2.0
```

This example checks the stim fast settle duration for the front end in position 0. The duration is set to 2.0 ms.

Impedance

The ‘xipppy.impedance’ function can be used to return the magnitude of the impedance in Ohms.

Usage

```
output = xipppy.impedance(elecs)
```

Input	elecs	Required. Zero-indexed list of desired electrodes.
Output	output	List of measured impedance on the inputted list of channels

Example

```
>>> xp.impedance([1,2,3])
array('f', [1072831.25, 1037615.625, 1079905.5])
```

Retrieves the magnitude of the Impedance result in Ohms for channels 2, 3, and 4.

Transceiver Enable

The ‘xipppy.transceiver_enable’ function can be used to enable or disable a Link 32 transceiver.

Usage

```
xipppy.transceiver_enable(front_end, enable=True)
```

Input	front_end	Front End location of the Link 32 transceiver. Position A.1 = 0, A.2 = 1, …, D.4 = 15
Input	enable	Boolean value to enable (True) or disable (False) the Link 32 transceiver

Example

```
>>>xp.transceiver_enable(6, True)
```

Enables the transceiver in Port B, position 3 (B.3) and returns a value of 1 if the command was successful.

Sensor

Sensor Class

The 'xipppy.sensors.Sensor' class has three attributes, current, voltage and power. This class is returned in the functions 'xipppy.wall_sensor' and 'xipppy.vdd_sensor'. These functions take no arguments.

Battery

Xipppy has the capability to read the current battery percentage on the internal and external battery (if attached). The function 'xipppy.internal_battery' will return the current internal battery percentage. 'xipppy.sensors.external_battery_detected' will return a Boolean indicating if an external battery is connected to the processor. 'xipppy.sensors.external_battery' will return the current external battery percentage, returning 0 if no external battery is connected.

Processor

Button get

The function 'xipppy.button_get' will return the number of times a specified button has been pressed since the last time the function was called.

Usage

```
presses = xipppy.button_get(button)
```

Input	button	Integer specifying the button of interest. 1 – STOP STIM, 2 – EVENT, 3 – F1, 4 – F2.
Output	presses	Returns the number of times the specified button has been pressed since the last function call for that button.

EXAMPLE SCRIPTS

Example scripts demonstrating various functions and applications of the Xipppy library are available at Projects & Examples - Ripple Neuro. Please contact support@rppl.com for any questions or requests for additional examples.

LEGACY FUNCTIONS

With 'xipppy.xipppy_open()':
‘xipppy._close()’ should not be called from within a xipppy_open context.

© Ripple Neuro 2022. All rights reserved.
Product specifications are subject to change without notice.
Ripple Neuro is not responsible for the misuse of its products.
For more information, email support@rppl.com or call
1-800-380-5800 or 1-801-413-0139.