TDTPy Documentation

Brad Buran

Jan 30, 2023

Contents

CHAPTER 1

Python interface for TDT equipment

Contributors

- Brad Buran (New York University; Oregon Health & Science University)
- Eric Larson (University of Washington)
- Decibel Therapeutics, Inc.

Acknowledgements

Work on TDTPy was supported by grant DC009237 from the National Institute on Deafness and other Communication Disorders.

Note: If you use the server provided by TDTPy to communicate with your hardware your client code should be able to run on any platform including Unix, Linux and OSX). The server, however, requires the proprietary ActiveX drivers provided by TDT which only run on Windows.

CHAPTER 2

Contents

2.1 Installing

You can use Python's easy_install tool:

```
c:\\> easy_install tdtpy
```

Or, if you have pip installed, that works as well:

```
c:\\> pip install tdtpy
```

The source code is hosted as a Mercurial repository at http://bitbucket.org/bburan/tdtpy.

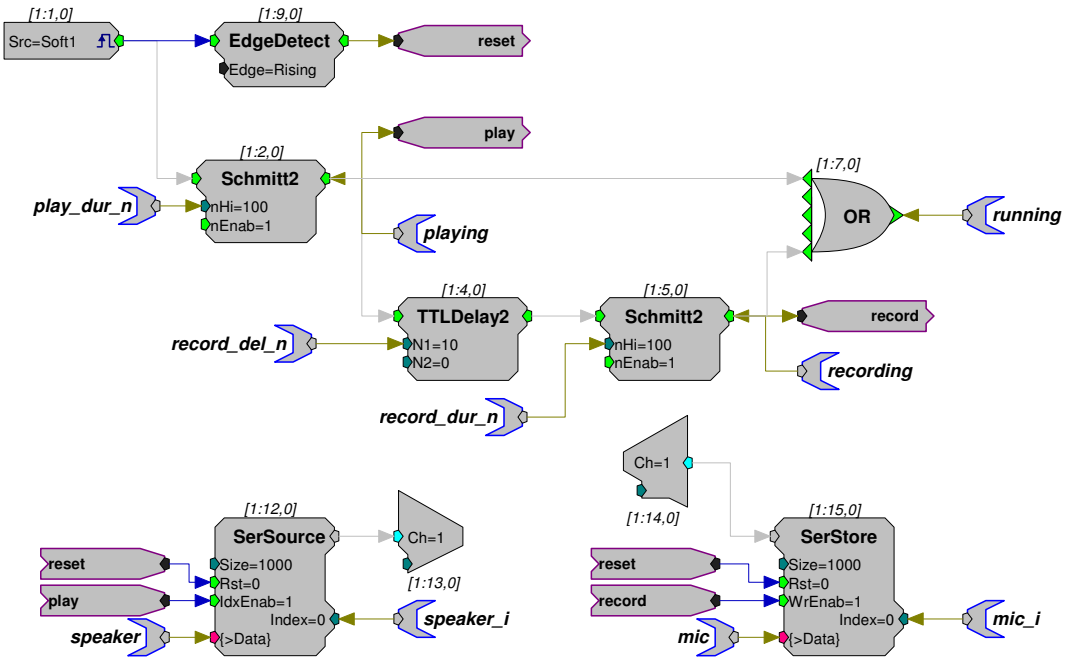
Note: If you want to build a local copy of the documentation, you'll need to install the Python modules *sphinx* and *numpydoc*. Keep in mind that the most recent version is always hosted at readthedocs.org.

2.2 Code examples

Note: Familiarity with TDT's real-time processor visual design studio (RPvds) is required to follow the examples below. See the RPvds manual for more information.

2.2.1 Walkthrough of a simple play/record circuit

The following example is based on this RPvds circuit (`download circuit`). If you wish to test the circuit you may need to adapt it for your specific device (e.g. on the RX6 the correct input channel for the microphone would be 128 and on the RZ6 you would use the AudioIn and AudioOut macros). The specifics for each device are described in TDT's System 3 manual.



Let's start with a simple code example, using TDTPy, that loads a circuit and reads data from a buffer:

```
from numpy import arange, sin, pi
from tdt import DSPProject, DSPError
try:
    # Load the circuit
    project = DSPProject()
    circuit = project.load_circuit('record_microphone.rcx', 'RZ6')
    circuit.start()

    # Configure the data tags
    circuit.cset_tag('record_del_n', 25, 'ms', 'n')
    circuit.cset_tag('record_dur_n', 500, 'ms', 'n')

    # Compute and upload the waveform
    t = arange(0, 1, circuit.fs**-1)
    waveform = sin(2*pi*1e3*t)
    speaker_buffer = circuit.get_buffer('speaker', 'w')
    speaker_buffer.write(waveform)

    # Acquire the microphone data
    microphone_buffer = circuit.get_buffer('mic', 'r')
    data = microphone_buffer.acquire(1, 'running', False)
except DSPError, e:
    print("Error acquiring data: {}".format(e))
```

If you were to do the same thing using TDT's ActiveX driver directly, the code would be much more verbose:

```
from win32com.client import Dispatch
try:
    # Load the circuit
    RX6 = Dispatch('RPco.X')
    if RX6.ConnectRX6('GB', 1) == 0:
        raise SystemError, "Cannot connect to hardware"
    if RX6.ClearCOF() == 0:
        raise SystemError, "Cannot connect clear device"
    if RX6.LoadCOF('record_microphone.rcx') == 0:
        raise SystemError, "Cannot load circuit"

    # Configure the data tags
    fs = RX6.GetSFreq()
    if RX6.SetTagVal('record_del_n', int(25e-3*fs)) == 0:
        raise SystemError, "Cannot set tag"
    if RX6.SetTagVal('record_dur_n', int(500e-3*fs)) == 0:
        raise SystemError, "Cannot set tag"
    if RX6.Start() == 0:
        raise SystemError, "Cannot start circuit"

    # Compute and upload the waveform
    t = arange(0, int(1*fs))/fs
    waveform = sin(2*pi*1e3*t)
    RX6.WriteTagV('speaker', 0, waveform)

    # Acquire the microphone data
    if RX6.SoftTrg(1) == 0:
        raise SystemError, "Cannot send trigger"
    last_read_index = 0
    acquired_data = []
    while True:
```

(continues on next page)

(continued from previous page)

```
        if RX6.GetTagV('running') == 0:
            last_loop = True
        else:
            last_loop = False
        next_index = RX6.GetTagVal('mic_i')
        if next_index > last_read_index:
            length = next_index - last_read_index
            data = RX6.ReadTagV('mic', last_read_index, length)
        elif next_index < last_read_index:
            length_a = RX6.GetTagSize('mic') - last_read_index
            data_a = RX6.ReadTagV('mic', last_read_index, length_a)
            data_b = RX6.ReadTagV('mic', 0, next_index)
            data = np.concatenate(data_a, data_b)
        acquired_data.append(data)
        last_read_index = next_index
        if last_loop:
            break
    data = np.concatenate(acquired_data)
except SystemError, e:
    print("Error acquiring data: {}".format(e))
```

Compared with the code using the TDTPy module, code working with the ActiveX object directly requires a lot more boilerplate code.

Warning: Due to non-standard implementation of ActiveX in the TDT libraries, win32com defaults to an inefficient approach when calling certain methods in the ActiveX library. This results in a significant data transfer bottleneck. For more detail, and a description of how TDTPy solves this problem, see Brad Buran's post.

Ok, let's walk through the first example to illustrate how it works. First, we need to import everything we need:

```
from numpy import arange, sin, pi
from tdt import DSPProject, DSPError
```

Now, initialize the project and load the circuit, saved in a file named 'record_microphone.rcx' to the RZ6 DSP:

```
project = DSPProject()
circuit = project.load_circuit('record_microphone.rcx', 'RZ6')
```

Note that you can leave the default file extension off if desired. If the circuit is not in the current directory, you must provde an absolute or relative path to the circuit.

The circuit has the buffers `mic` and `speaker` as well as the tags `record_dur_n` and `record_del_n`. Note that some tag names end in `_n`. This is a special naming I use to remind myself what units these tags require ('n' indicates number of ticks of the DSP clock while 'ms' indicates milliseconds). Both `mic` and `speaker` have two supporting tags, `speaker_i` and `mic_i`, respectively, that are used by TDTPy to determine how much data is currently in the buffer.

The circuit is configured to deliver the data stored in the speaker buffer to DAC channel 1 (which is connected to a speaker) and record the resulting microphone waveform. The entire process is controlled by a software trigger.

We want to configure the microphone to record for a duration of 500 ms with a 25 ms delay. Remember that `record_del_n` and `record_dur_n` both require the number of samples. Since number of samples depends on the sampling frequency of the DSP, we have to convert our value, which is in millseconds, to the appropriate unit using *`tdt.DSPCircuit.set_tag()`*:

```
circuit.set_tag('record_del_n', int(25e-3*circuit.fs))
circuit.set_tag('record_dur_n', int(500e-3*circuit.fs))
```

Alternatively, we can use a convenience method, `DSPCircuit.cset_tag()`, that handles the unit conversion for us (n is number of samples):

```
circuit.cset_tag('record_del_n', 25, src_unit='ms', dest_unit='n')
circuit.cset_tag('record_dur_n', 500, src_unit='ms', dest_unit='n')
```

Or, if we just rely on positional arguments (which we use in the example above):

```
circuit.cset_tag('record_del_n', 25, 'ms', 'n')
circuit.cset_tag('record_dur_n', 500, 'ms', 'n')
```

All three of the approaches are fine; however, we recommend that you use `DSPCircuit.cset_tag()` whenever possible since this makes the code more readable.

To write a 1 second, 1 kHz tone to the speaker buffer, we first generate the waveform using the sampling frequency of the circuit. The sampling frequency is available as an attribute, *fs* of the `DSPCircuit` class. A method, `DSPCircuit.convert()` facilitates unit conversions that are based on the sampling frequency of the circuit (e.g. *duration*fs* will convert duration, in seconds, to the number of sample required for the waveform):

```
t = arange(0, circuit.convert(1, 's', 'n'))/circuit.fs
waveform = sin(2*pi*1e3*t)
```

Then we open the speaker buffer for writing and write the data to the buffer. The first argument to `DSPCircuit.get_buffer()` is the name of the tag attached to the `{>Data}` port of the buffer component and the second argument indicates whether the buffer should be opened for reading (r) or writing (w):

```
speaker_buffer = circuit.get_buffer('speaker', 'w')
speaker_buffer.write(waveform)
```

Now that you've configured the circuit, you are ready to run it and record the resulting waveform. The `DSPBuffer.acquire()` method will block until the `running` tag becomes False then return the contents of the microphone buffer:

```
microphone_buffer = circuit.get_buffer('microphone', 'r')
data = microphone_buffer.acquire(1, 'running', False)
```

2.2.2 Accessing the raw ActiveX object

Although DSPCircuit and DSPBuffer expose most of the functionality available via the ActiveX object, there may be times when you need to access it directly. You may obtain a handle to the object via *`tdt.util.connect_rpcox()`*:

```
from tdt.util import connect_rpcox
obj = connect_rpcox('RZ6', 'GB')
```

2.3 Converting your code from Matlab or Python to use TDTPy

2.3.1 Connecting to a device and loading a circuit

Matlab:

```
iface = actxserver('RPco.X');
if iface.ConnectRZ6('GB', 1) == 0
    disp 'connect error';
end
if iface.ClearCOF == 0
    disp 'clear error';
end
if iface.LoadCOF('record_microphone.rcx') == 0
    disp 'load error';
end
if iface.Run == 0
    disp 'run error';
end
```

Python:

```
from win32com.client import Dispatch
try:
    pass
    iface = Dispatch('RPco.X')
    if not iface.ConnectRZ6('GB', 1):
        raise SystemError, 'connect error'
    if not iface.ClearCOF():
        raise SystemError, 'clear error'
    if not iface.LoadCOF('record_microphone.rcx'):
        raise SystemError, 'load error'
    if not iface.Run():
        raise SystemError, 'run error'
except SystemError, e:
    print "Error: {}".format(e)
```

TDTPy:

```
from tdt import DSPCircuit
try:
    circuit = DSPCircuit('record_microphone', 'RZ6')
    circuit.start()
    circuit.stop()
except DSPError, e:
    print "Error: {}".format(e)
```

2.3.2 Getting/Setting a tag value

Matlab:

```
iface.SetTagVal('nHi', 5);
fs = iface.GetSFreq();
delay = 25/1000*fs;
iface.SetTagVal('record_del_n', delay);
duration = iface.GetTagVal('record_dur_n')/fs;
```

Python:

```
iface.SetTagVal('nHi', 5)
fs = iface.GetSFreq()
delay = 25e-3*fs
```

(continues on next page)

(continued from previous page)

```
iface.SetTagVal('record_del_n', delay)
duration = iface.GetTagVal('record_dur_n')/fs
```

TDTPy:

```
circuit.set_tag('nHi', 5)
circuit.cset_tag('record_del_n', 25, 's', 'n')
duration = circuit.cget_tag('record_dur_n', 'n', 's')
```

2.3.3 Writing data to a buffer

Matlab:

```
iface.WriteTagV('speaker', 0, data);
```

Python:

```
iface.WriteTagV('speaker', 0, data)
```

TDTPy:

```
speaker = iface.get_buffer('speaker', 'w')
speaker.write(data)
```

2.3.4 Reading data from a buffer

Matlab:

```
size = iface.GetTagV('mic_i');
data = iface.ReadTagV('speaker', 0, size);
```

Python:

```
size = iface.GetTagV('mic_i')
data = iface.ReadTagV('speaker', 0, size)
```

TDTPy:

```
mic = iface.get_buffer('mic', 'r')
data = mic.read()
```

2.4 `tdt.DSPCircuit` – Wrapper for RPvds circuit objects

Wrapper around a RPvds circuit

```
>>> from tdt import DSPCircuit
>>> circuit = DSPCircuit('acquire_neurophysiology.rcx', 'RZ5')
```

2.4.1 Parameters

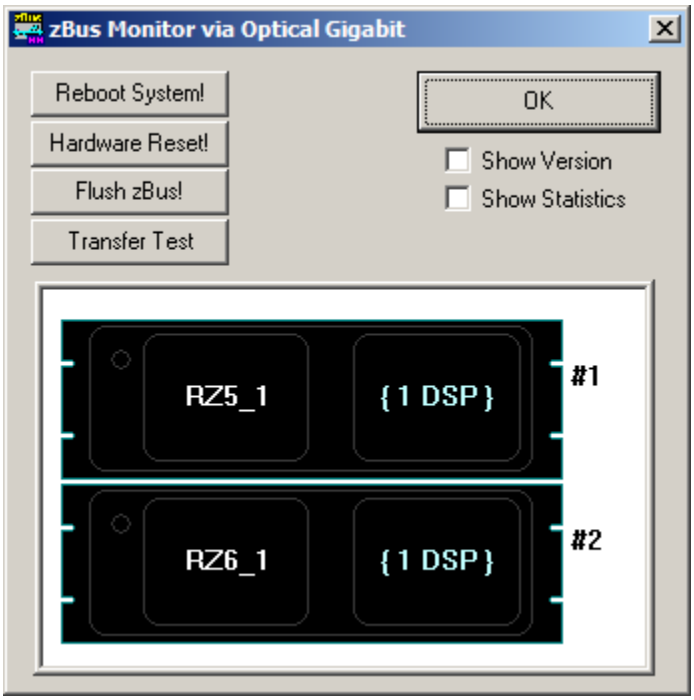
circuit_name [path (required)] Absolute or relative path pointing to the file containing the circuit microcde (e.g. the *.rcx file).

device_name [str (required)] Target device to load the microcode to (the name will be in the format RP2, RX6, RX8, RZ5, RZ6, etc.).

device_id [int (optional, default=1)] Specifies which of the two devices to load the microcode to. Required only if you have more than one of the same device (e.g. two RP2 processors). Use TDT's zBUSmon utility to look up the correct device ID.

load [boolean (optional, default=True)] Load the circuit to the device when class is initialized? True by default. Typically you would set it to False when you want to inspect the DSP microcode without actually running it.

If you're not sure what to enter for device_name and device_id, use TDT's zBUSmon utility to look up the correct information. As shown in the screenshot below, two devices installed in the system are the RZ5_1 and RZ6_1. The device names are RZ5 and RZ6, respectively, while the device ID is 1 for both. If zBUSmon reports that you have a RZ5_1 and RZ5_2, then both device names would be RZ5 while the device ID would be 1 and 2, respectively.



2.4.2 Available public attributes

fs [float] Sampling frequency of the circuit

tags [dictionary] Keys are the tag names (i.e. variables) present in the DSP microcode. Values are a tuple of tag size and tag type. Note that *tdt.constants* defines the available tag types. For simple types (e.g. integer, float and boolean), the tag size will always be 1. For buffer types, the size will indicate the number of 32 bit words in the buffer.

scalar_tags [list] List of tag names present in the DSP microcode that have a tag size of 1 (i.e. a scalar value such as an integer, float or boolean).

vector_tags [list] List of tag tag names (i.e. variables) present in the DSP microcode that have a tag size >= 1 (i.e. buffer or coefficient tag).

name [str] Name of circuit currently loaded

path [str] Full path of circuit on disk

Brief example of the public attributes available for the example circuit, record_microphone.rcx shown in the introduction:

```
>>> print circuit.fs
97656.25
>>> print circuit.scalar_tags
['mic_i', 'speaker_i', 'play_dur_n', 'record_del_n',
 'record_dur_n', 'recording', 'playing', 'running']
>>> print circuit.vector_tags
['speaker', 'mic']
>>> print circuit.name
example_circuit.rcx
>>> print circuit.tags
{'mic': (100000, 68),
 'mic_i': (1, 73),
 'play_dur_n': (1, 73),
 'playing': (1, 76),
 'record_del_n': (1, 73),
 'record_dur_n': (1, 73),
 'recording': (1, 76),
 'running': (1, 76),
 'speaker': (100000, 68),
 'speaker_i': (1, 73)}
```

2.4.3 Error handling

Attempting to get/set the value of a nonexistent tag in the circuit will raise a *DSPError*:

```
>>> circuit.get_tag('nonexistent_tag')
DSPError: 'nonexistent_tag' not found in circuit
```

Note: If you have a tag linked to a *static* datatype the DSPCircuit class will raise an exception. Since the ActiveX driver cannot read from (or write to) this tag, this typically indicates a design error in the RPvds circuit.

2.4.4 Suggested code conventions

Sharing code across circuits

The current version of TDT's real-time processor visual design studio (RPvds) does not facilitate code reuse. The macro system is undocumented and clearly not meant for general use. For example, a macro embedded into a circut has the *absolute* path to the macro hard-coded. This makes it extremely difficult to place circuits using macros under revision control and maintain multiple branches on the same computer. The copy of the circuit in each branch will insist on loading the macro stored in the directory of the original branch where the commit was made, not the location of the macro in the new branch. Furthermore, if you decide to move your code to a new folder, you must manually update the reference to the macros in each circuit you use (even if the relative path between the macro and circuit remains unchanged).

Instead, create a page in your circuit file that contains *only* the shared code that you would like each circuit to use. Whenever you update the code on this page, it's easy to cut and paste the modified code to the other circuits that also

use it. Just be sure to keep the same naming conventions for whatever tags and hops you use in the common portion of the code.

Tag naming

Use right-pointing tags to indicate that they are meant to be written and left-pointing tags to indicate they are meant to be read. Although a tag can be used for both purposes, it makes it much easier for a new programmer to ascertain the purpose of the tag. Is it meant to be a setting that can be modified via the software, or does it hold data that is meant for the software?

If the output of the tag reflects an epoch boundary, use the '/' suffix to indicate the start and '' to indicate the end. If it is simply a point in time (i.e. a timestamp), use the 'I' suffix.

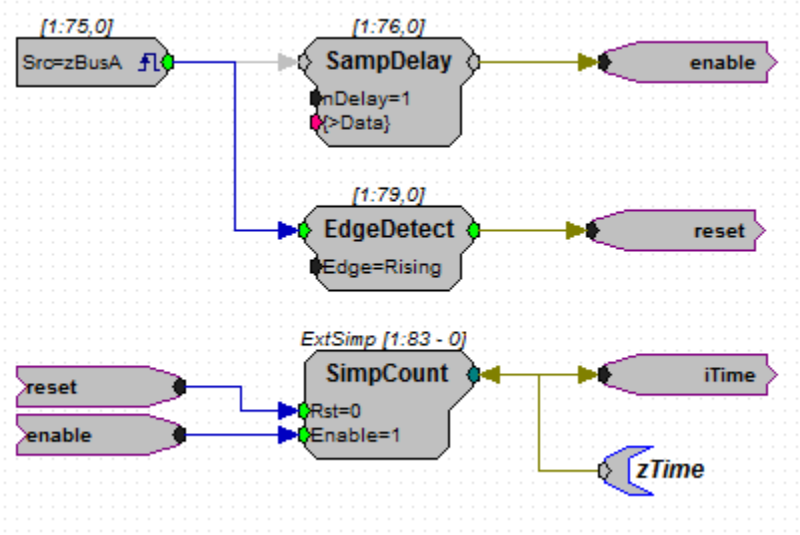
If the tag requires a certain unit (e.g. msec or number of samples), be sure to indicate the unit in the tag name using the appropriate suffix. For example, tags requiring a value in msec should have the suffix '_ms' and tags requiring the number of samples should have the suffix '_n'.

Hop naming

Use the '_start' and '_end' suffix to indicate the hop reflects a logical value that is true for only one cycle of the sample clock (i.e. the output of an EdgeDetect component). Use the '_TTL' or '_window' suffix to indicate that the hop reflects a logical value that is true for some duration of time.

zBUS trigger A

In many cases it's a good idea to put most of the circuit under control of zBUS trigger A using the following circuit construct.



2.5 `tdt.DSPBuffer` – Wrapper for RPvds buffer objects

Each buffer requires tags linked to the data and index parameters of the buffer component. All other tags described below are optional, but will be used if present. The data tag and supporting tags can have any name; however, the recommended approach is to use the data tag plus one of the following prefixes indicating the purpose of the supporting tag.

i index tag (idx_tag)

n size tag (size_tag)

sf scaling factor tag (sf_tag)

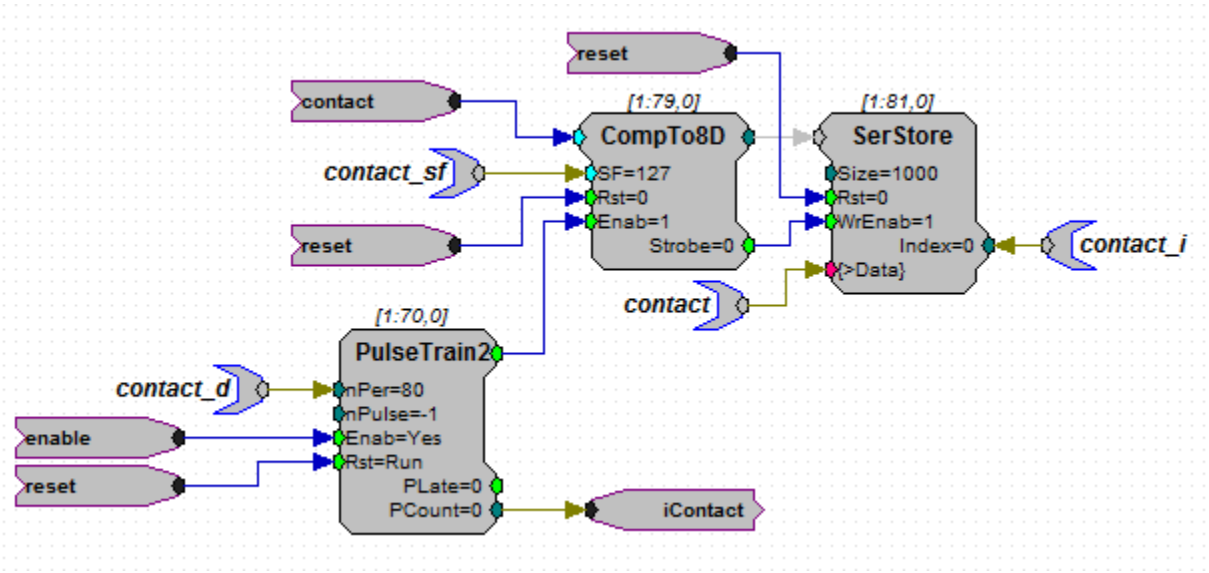
c cycle tag (cycle_tag)

d downsampling tag (dec_tag)

If no value is provided for a tag, the default extension is added to the value for the data_tag and the circuit is checked to see if the tag exists. For example, if you have *spikes*, *spikes_i*, and *spikes_n* tags in your RPvds circuit, you can simply initialize the class by passing only the name of the data tag (*spikes*) and it will automatically use *spikes_i* and *spikes_n* as the index and size tags, respectively:

```
>>> buffer = circuit.get_buffer('spikes', 'r')
```

If a required tag cannot be found (either by explicitly defining the tag name or automatically by adding the default extension to the data tag name), an error is raised.



In the above code, there is a singlue buffer named *contact* with three supporting tags (*contact_d*, *contact_sf* and *contact_i*) that assist the `tdt.DSPBuffer` class in reading data stored in *contact*. For example, we know that, due to the fact that we are applying a scaling factor of 127 to the floating-point data stored in the contact buffer, we are only saving the data with a resolution of 0.00787:

```
>>> contact_buffer = circuit.get_buffer('contact', 'r', src_type='int8')
>>> print contact_buffer.resolution
0.00787
```

Because we specified that the data is stored in 8-bit format, four samples are being compressed into a single 32-bit slot:

```
>>> print contact_buffer.compression
4
```

Since *contact_d* is set to 80 (i.e. acquire and save a sample every 80 cycles), we know the sampling frequency of the contact data is only 1/80th of the sampling rate of the DSP:

```
>>> print circuit.fs
97656.25
>>> print contact_buffer.fs
1220.703125
```

Note: This buffer uses the enable and reset hops to control data acquisition, consistent with the coding guidelines described in dsp_buffer.rst.

Note: Currently TDTPy does not support changing sampling rate on-the-fly (you can do it, but you need to reload the buffer).

2.5.1 Writing single channel data

If you are using epoch-based outputs (where you upload a waveform of fixed size and halt playout once the buffer is complete), then you can use the *WriteableDSPBuffer.set* method:

```
>>> speaker_buffer = circuit.get_buffer('speaker', 'w')
>>> speaker_buffer.set(tone_pip)
```

If you are using continuous output (e.g., where you need to update the stream as the experiment progresses):

TODO

TDTPy has not been tested with writing multi-channel data (mainly because we currently do not have a use-case for it).

2.5.2 Reading single and multichannel data

TODO

2.5.3 Tags

data_tag [string (required)] Tag to read data from

idx_tag [defaults to data_tag_i (required)] Tag indicating current index of buffer. For buffer reads this tag serves as a "handshake" (i.e. when the index changes, new data is available).

size_tag [defaults to data_tag_n (optional)] Tag indicating current size of buffer.

sf_tag [defaults to data_tag_sf (optional)] Tag indicating scaling factor applied to data before it is stored in the buffer.

cycle_tag [defaults to data_tag_c (optional)] Tag indicating number of times buffer has wrapped around to beginning. Used to ensure no data is lost.

dec_tag [defaults to data_tag_d (optional)] Tag indicating decimation factor. Used to compute sampling frequency of data stored in buffer: e.g. if circuit runs at 100 kHz, but you only sample every 25 cycles, the actual sampling frequency is 4 kHz.

latch_trigger [{None, 1, 2, 3, 4}] The _c and _i (buffer cycle and buffer index) tags should be passed through a latch to avoid race conditions when reading these values as each read is a separate call. This indicates which software trigger is connected to the latch in the RPvdsEx circuit. If None, no trigger will be fired.

2.5.4 Additional Parameters

circuit [instance of *tdt.DSPCircuit*] Circuit object the buffer is attached to

block_size [int] Coerce data read/write to multiple of the block size. Must be a multiple of the channel number.

src_type [str or numpy dtype] Type of data in buffer (can be a string or numpy dtype). Valid data formats are float32, int32, int16 and int8.

dest_type [str or numpy dtype] Type to convert data to

channels [int] Number of channels stored in buffer

2.5.5 Available attributes

When the buffer is first loaded, there is some "introspection" of the circuit to determine key properties of the buffer (e.g. what is the format of the data stored in the DSP buffer, how much data can be stored before the buffer fills up, etc.).

data_tag, idx_tag, size_tag, sf_tag, cycle_tag, dec_tag [str] Names of supporting tags present in the circuit (both the names provided when the b uffer was loaded as well as the ones automatically discovered when the buffer is created. None if the tag is not present.

src_type Numpy dtype of the data stored on the device. Defaults to float32.

dest_type Numpy dtype of array returned when data is read from the device

compression Number of samples stored in a single 32-bit "slot" on the device. For example, if you are using the MCFloat2Int8 component to convert four samples of data into 8-bit integers and storing these four samples as a single 32-bit work, the compression factor is 4.

sf Scaling factor of the data. If you are not using compression, the scaling factor is almost certainly one.

resolution If data is being compressed, computes the actual resolution of the acquired data given the scaling factor. For example, if you are compressing data into an 8-bit integer using a scaling factor of 10, then the resolution of the acquired data will be 0.1 since numbers will get rounded to the nearest tenth (e.g. 0.183 will get rounded to 0.2).

dec_factor Also called the "downsampling rate". Indicates the number of device cycles before a sample is stored in the buffer. If 1 (default), a sample is acquired on every cycle. If 2, a sample is acquired on every other cycle.

fs Sampling frequency of data stored in buffer. This is basically the sampling frequency of the device divided by the decimation factor (dec_factor): e.g. if a sample is acquired only on every other cycle, then the sampling frequency of the buffer is effectively half of the device clock rate.

channels Number of channels

block_size Coerce read size to multiples of this value (can be overridden if needed)

2.5.6 Buffer size attributes

There are three ways to think about the buffer size. First, how many 32-bit words can the buffer hold? All buffer components in a RPvds circuit store data in 32-bit word segments. However, we can store two 16-bit values or four 8-bit values into a single word. Even if a buffer can only hold 1000 32-bit words, it may actually hold 2000 or 4000 samples if we are compressing two or four samples of data into a single buffer "slot". Now, if we are storing multiple channels of data in a single buffer, then the buffer will fill up more quickly than an identically-sized buffer storing only a single channel of data. By reporting buffer size as the number of samples per channel, we can get a sense for how quickly the buffer will fill up.

```
>>> buffer = circuit.get_buffer('spikes', 'r', channels=16)
>>> print buffer.compression    # number of samples in each buffer slot
2
>>> print buffer.n_slots        # number of slots
4000
>>> print buffer.n_samples      # number of samples
8000
>>> print buffer.size           # number of samples per channel
500
>>> print buffer.fs             # sampling frequency of buffer data
12207.03125
>>> print buffer.sample_time    # time (in seconds) to fill up the buffer
0.04096
```

In the above example, we know that even though the buffer can hold 8,000 samples of data, it will fill up after only 500 samples of 16-channel data are collected. At a sampling frequency of 12 kHz, this means the buffer can only hold 41 msec of 16-channel data. This provides a useful metric for knowing whether we have set the buffer size appropriately.

n_slots Size in number of 32-bit words (the buffer's atomic unit of of storage)

n_samples Size in number of samples (data points) that can be stored in the buffer. The size will be either 1x, 2x or 4x the size of n_slots depending on how many samples are stored in each slot.

size Size in number of samples (data points) per channel.

sample_time How many seconds before the buffer is full?

It is also possible to resize buffers in the RPvds circuit if a size_tag is present. The above attributes reflect the current size of the buffer, which may be smaller than the maximum possible size allocated.

n_slots_max Maximum size in number of 32-bit words

n_samples_max Maximum size in number of samples

size_max Maximum size in number of channels

2.5.7 Acquiring segments of data

Two utility methods, *DSPBuffer.acquire* and *DSPBuffer.acquire_samples* are provided to facilitate the common task of acquiring a segment of data in response to some stimulus. They both fire a trigger then continuously download data from the buffer until a certain end condition is met. This end condition can either be the number of samples acquired or the value of a tag in th RPvds circuit.

The *DSPBuffer.acquire* method takes three arguments:

- The trigger to fire, initiating data acquisition. If None, no trigger is fired and acquire begins spooling data immediately.
- The tag on the DSP to monitor.
- The value of the monitor tag that indicates data acquisition is done. If not provided, the initial value of the tag will be retrieved before firing the trigger. In this situation, the end condition is met when the value of the tag changes from its initial value.

Fire trigger 1 and continuously acquire data until `running` tag is False:

```
microphone_buffer.acquire(1, 'recording', False)
```

Fire trigger 1 and continuously acquire data until `complete` tag is True:

```
microphone_buffer.acquire(1, 'complete', True)
```

Get the initial value of `toggle`, fire trigger 1, then continuously acquire data until the value of `toggle` changes:

```
microphone_buffer.acquire(1, 'toggle')
```

Continuously acquire until the value of the trial end timestamp, `trial_end|` changes:

```
microphone_buffer.acquire(1, 'trial_end|')
```

Fire trigger 1 and continuously acquire data until `index` tag is greater or equal to 10000:

```
microphone_buffer.acquire(1, 'index', lambda x: x >= 1000)
```

Fire trigger 2 and acquire 100000 samples of data:

```
microphone_buffer.acquire_samples(2, 100000)
```

Note: The acquire method continuously downloads data while monitoring the end condition. This allows you to acquire sets of data larger than the buffer size without losing any data. Just be sure that the poll interval is short enough to grab new data before it gets overwritten. To determine how quickly your buffer will fill, check its *sample_time* attribute.

Note: A very common mistake to make is setting the block size for the buffer to a number that is not an integer divisor of the number of samples to be acquired. If you are acquiring 10000 samples of data and set the block size to 1048, then both *DSPBuffer.acquire* and *DSPBuffer.acquire_samples* will hang after acquiring 9432 samples since they are waiting for another 1048 samples to be acquired, but only 568 new samples are in the buffer. If you don't know in advance what the final length of the data will be, just leave the block size at its default value of 1.

To prevent this from happening, a ValueError will be raised if you attempt to acquire a number of samples that is not a multiple of block size.

2.6 API documentation

2.6.1 tdt.util

Functions for loading the zBUS, PA5 and RPcoX drivers and connecting to the specified device. In addition to loading the appropriate ActiveX driver, some minimal configuration is done.

Network-aware proxies of the zBUS and RPcoX drivers have been written for TDTPy. To connect to TDT hardware that is running on a remote computer, both the *connect_zbus()* and *connect_rpcox()* functions take the address of the server via a tuple (hostname, port):

```
connect_rpcox('RZ6', address=(tdt_server.cns.nyu.edu, 3333))
```

`tdt.util.`**`connect_zbus`**(*interface='GB'*, *address=None*)

Connect to the zBUS interface and set the zBUS A and zBUS B triggers to low

Parameters

- **interface** (*{'GB', 'USB'}*) – Type of interface (depends on the card that you have from TDT). See the TDT ActiveX documentation for clarification on which interface you would be using if you are still unsure.
- **address** (*{None, (hostname, port)}*) – If None, loads the ActiveX drivers directly, otherwise connects to the remote server specified by the hostname, port tuple.

tdt.util.**connect_rpcox**(*name*, *interface='GB'*, *device_id=1*, *address=None*)
Connect to the specifed device using the RPcoX driver

Note that the appropriate RPcoX.Connect method is called so you do not need to perform that step in your code.

Parameters

- **name** (*{'RZ6', 'RZ5', 'RP2', .. (any valid device string) }*) – Name of device (as defined by the corresponding RPcoX.Connect* method).
- **interface** (*{'GB', 'USB'}*) – Type of interface (depends on the card that you have from TDT). See the TDT ActiveX documentation for clarification on which interface you would be using if you are still unsure.
- **device_id** (*int (default 1)*) – Id of device in the rack. Only applicable if you have more than one of the same device (e.g. two RX6 devices).
- **address** (*{None, (hostname, port)}*) – If None, loads the ActiveX drivers directly, otherwise connects to the remote server specified by the hostname, port tuple.

tdt.util.**connect_pa5**(*interface='GB'*, *device_id=1*, *address=None*)
Connect to the PA5

Note: The network-aware proxy code should be considered alpha stage. Although it appears to work in our tests, we have not deployed this in our data aqcuisition experiments.

2.6.2 tdt.DSPProject

class tdt.**DSPProject**(*address=None*, *interface='GB'*)
Used to manage loading circuits to multiple DSPs. Mainly a convenience method.

load_circuit(*circuit_name*, *device_name*, *device_id=1*, ***kw*)
Load the circuit to the specified device

Parameters

- **circuit_name** (*str*) – Path to circuit to load
- **device_name** (*str*) – Name of TDT System3 device to load circuit to
- **device_id** (*number*) – ID of device

Returns **circuit** – The circuit.

Return type instance of DSPCircuit

start()
Start all circuits that have been loaded

stop()
Stop all circuits that have been loaded

trigger(*trigger*, *mode='pulse'*)
Fire a zBUS trigger

Parameters

- **trigger** (*{'A', 'B'}*) – Fire the specified trigger. If integer, this corresponds to RPco.X.SoftTrg. If 'A' or 'B', this fires the corresponding zBUS trigger.
- **mode** (*{'pulse', 'high', 'low'}*) – Indicates the corresponding mode to set the zBUS trigger to
- **that due to a bug in the TDT ActiveX library for versions greater** (*Note*) –
- **56, we have no way of ensuring that zBUS trigger A or B were** (*than*) –
- **fired.** –

2.6.3 tdt.DSPCircuit

class tdt.**DSPCircuit**(*circuit_name*, *device_name*, *interface='GB'*, *device_id=1*, *load=True*, *start=False*, *fs=None*, *address=None*, *latch_trigger=None*)

Wrapper around the TDT ActiveX object, RPCo.X.

Provides several stringent checks and convenience methods to minimize programming errors and typos.

circuit_name [string] Path to circuit file.

device_name [string] Device to load circuit to.

interface [{'GB', 'USB'}] Interface to use (see TDT's ActiveX documentation on the Connect* methods for more information). You almost certainly want 'GB' (which is the default value).

device_id [number] ID of device

load [boolean (optional)] Load circuit to specified device. Default is True. Set to False if you just want to get a list of the tags available in the circuit.

start [boolean (optional)] Start (i.e. run) the circuit after loading it. Default is False.

address [two-tuple (str, int)] Connect to the address specified as a two-tuple in (host, port) format using the network-aware proxy of TDT's driver. If None, defaults to the TDT implementation of the RPcoX and zBUSx drivers.

latch_trigger [{None, 1, 2, 3, 4}] Trigger used for latching values when we need to capture a snapshot of some tags at a given point in time. This is used by DSPBuffer to eliminate race conditions when reading the value of the cycle and index tags. If the tags are not latched, then it's possible to read the index tag, then by the time the cycle tag is read the index has wrapped around to the beginning of the buffer.

cget_tag(*name*, *tag_unit*, *val_unit*)

Enhanced version of *get_tag* that returns value in requested unit

Parameters

- **name** (*str*) – Tag name
- **tag_unit** (*str*) – Unit of tag
- **val_unit** (*str*) – Requested unit

convert(*value*, *src_unit*, *dest_unit*)

Converts value to desired unit give the sampling frequency of the DSP.

Parameters specified in paradigms are typically expressed as frequency and time while many DSP parameters are expressed in number of samples (referenced to the DSP sampling frequency). This function

provides a convenience method for converting between conventional values and the 'digital' values used by the DSP.

Note that for converting units of time/frequency to n/nPer, we have to coerce the value to a multiple of the DSP period (e.g. the number of 'ticks' of the DSP clock).

Appropriate strings for the unit types:

fs sampling frequency

nPer number of samples per period

n number of samples

s seconds

ms milliseconds

nPow2 number of samples, coerced to the next greater power of 2 (used for ensuring efficient FFT computation)

Given a DSP clock frequency of 10 kHz:

```
>>> circuit.convert(0.5, 's', 'n')
5000
>>> circuit.convert(500, 'fs', 'nPer')
20
```

Given a DSP clock frequency of 97.5 kHz:

```
>>> circuit.convert(5, 's', 'nPow2')
524288
```

Parameters

- **value** (*numerical (e.g. integer or float)*) – Value to be converted
- **src_unit** (*string*) – Unit of the value
- **dest_unit** (*string*) – Destination unit

Returns converted unit

Return type numerical value

cset_tag(*name*, *value*, *val_unit*, *tag_unit*)

Enhanced version of *set_tag* that converts the value

Parameters

- **name** (*str*) – Name of the parameter tag to write the converted value to
- **value** (*int or float*) – Value to convert
- **val_unit** (*str*) – Unit of value provided
- **tag_unit** (*str*) – Unit parameter tag requires

Returns

- *Actual value of the tag (i.e. the converted value)*
- *Value will be converted from val_unit to tag_unit based on the sampling*
- frequency of the device (if needed). See **:module:'convert'** for more

- *information.*

get_tag(*name*)
Analogue of RPco.X.GetTagVal

Parameters

- **name** (*str*) – Name of the parameter tag to read the value from
- **DSPError** (*Raises*) – If the tag does not exist or is not a scalar value (e.g. you cannot use this method with parameter tags linked to a buffer)

inspect()
Determine what tags are available in the microcode

is_connected()
True if connection with hardware is active, False otherwise

is_loaded()
True if microcode is loaded, False otherwise

load()
Clear DSP RAM set all variables to default value

The circuit is reloaded from disk, so any recent edits to the circuit will be reflected in the running program.

print_tag_info()
Prints a list of tags and their current value if they are a scalar (buffer tags are not printed yet)

Used as a convenience method for debugging

set_coefficients(*name*, *data*)
Load data to a coefficient or matrix input

Parameters

- **name** (*str*) – Name of the parameter tag to write the data to
- **data** (*array-like*) – Data to write to tag. Must be 1D format (even for matrices). See RPvds documentation for appropriate ordering of indices for the component.
- **DSPError** (*Raises*) – If the specified parameter tag is not linked to a coefficient input or the length of the data is not equal to the size of the input on the component.
- **that as of 3.10.2011, RPvds' CoefLoad component appears to be** (*Note*) –
- **(per conversation with TDT's tech support -- Mark Hanus and** (*broken*) –
- **Walters) As a workaround, connect a data tag directly to the** (*Chris*) –
- **or >Coef input of the component.** (*>K*) –

set_sort_windows(*name*, *windows*)
Utility function for configuring TDT's SpikeSort component coefficients

Windows should be a list of 3-tuples in the format (time, center volt, half-height)

If the windows overlap in time such that they cannot be converted into a coefficient buffer, and error will be raised.

set_tag(*name*, *value*)
Analogue of RPco.X.SetTagVal

Parameters

- **name** (*str*) – Name of the parameter tag to write the value to
- **value** (*int or float*) – Value to write
- **DSPError** (*Raises*) – If the tag does not exist or is not a scalar value (e.g. you cannot use this method with parameter tags linked to a buffer)

set_tags(***tags*)
Convenience function for setting the value of multiple tags

```
>>> circuit.set_tags(record_duration=5, play_duration=4)
```

start(*pause=0.25*)
Analogue of RPco.X.Run

The circuit sometimes requires a couple hundred msec "settle" before we can commence data acquisition

stop()
Analogue of RPco.X.Halt

trigger(*trigger*, *mode='pulse'*)
Fire a zBUS or software trigger

Parameters

- **trigger** (*{1-9, 'A', 'B'}*) – Fire the specified trigger. If integer, this corresponds to RPco.X.SoftTrg. If 'A' or 'B', this fires the corresponding zBUS trigger.
- **mode** (*{'pulse', 'high', 'low'}*) – Relevant only when trigger is 'A' or 'B'. Indicates the corresponding mode to set the zBUS trigger to
- **that due to a bug in the TDT ActiveX library for versions greater** (*Note*) –
- **56, we have no way of ensuring that zBUS trigger A or B were** (*than*) –
- **fired.** –

2.6.4 tdt.DSPBuffer

class tdt.**DSPBuffer**(*circuit*, *data_tag*, *lock*, *idx_tag=None*, *size_tag=None*, *sf_tag=None*, *cycle_tag=None*, *dec_tag=None*, *block_size=1*, *src_type='float32'*, *dest_type='float32'*, *channels=1*, *dec_factor=None*, *latch_trigger=None*)
Given the circuit object and tag name, return a buffer object that serves as a wrapper around a SerStore or SerSource component. See the TDTPy documentation for more detail on buffers.

acquire(*trigger*, *handshake_tag*, *end_condition=None*, *trials=1*, *intertrial_interval=0*, *poll_interval=0.1*, *reset_read=True*)
Fire trigger and acquire resulting block of data

Data will be continuously spooled while the status of the handshake_tag is being monitored, so a single acquisition block can be larger than the size of the buffer; however, be sure to set poll_interval to a duration that is sufficient to to download data before it is overwritten.

Parameters

- **trigger** – Trigger that starts data acquistion (can be A, B, or 1-9)
- **handshake_tag** – Tag indicating status of data acquisition

- **end_condition** – If None, any change to the value of handshake_tag after trigger is fired indicates data acquisition is complete. Otherwise, data acquisition is done when the value of handshake_tag equals the end_condition. end_condition may be a Python callable that takes the value of the handshake tag and returns a boolean indicating whether acquisition is complete or not.
- **trials** – Number of trials to collect
- **intertrial_interval** – Time to pause in between trials
- **poll_interval** – Time to pause in between polling hardware
- **reset_read** – Should the read index be reset at the beginning of each acquisition sweep? If data is written starting at the first index of the buffer, then this should be True. If data is written continuously to the buffer with no reset of the index in between sweeps, then this should be False.

Returns **acquired_trials** – A 3-dimensional array in the format (trial, channel, sample).

Return type ndarray

Examples

```
>>> buffer.acquire(1, 'sweep_done')
>>> buffer.acquire(1, 'sweep_done', True)
```

acquire_samples(*trigger*, *samples*, *trials=1*, *intertrial_interval=0*, *poll_interval=0.1*, *reset_read=True*)
Fire trigger and acquire n samples

available(*offset=None*)
Number of empty slots available for writing

Parameters **offset** (*{None, int}*) – If specified, return number of samples relative to offset. Offset is relative to beginning of acquisition.

blocks_pending()
Number of filled blocks waiting to be read

clear()
Set buffer to zero

Due to a bug in the TDT ActiveX library, RPco.X.ZeroTag does not work on certain hardware configurations. TDT (per conversation with Chris Walters and Nafi Yasar) have indicated that they will not fix this bug. They have also indicated that they may deprecate ZeroTag in future versions of the ActiveX library.

As a workaround, this method zeros out the buffer by writing a stream of zeros.

find_tag(*tag*, *default_prefix*, *required*, *name*)
Locates tag that tracks a feature of the buffer

Parameters

- **tag** (*{None, str}*) – Name provided by the end-user code
- **default_prefix** (*str*) – Prefix to append to the data tag name to create the default tag name for the feature.
- **required** (*bool*) – If the tag is required and it is missing, raise an error. Otherwise, return None.

- **name** (*str*) – What the tag represents. Used by the logging and exception machinery to create a useful message.

Returns **tag_name** – Name of tag. If no tag found and it is not required, return None.

Return type {None, str}

Raises ValueError – If tag cannot be found and it is required.

get_tag(*tag*, *default*, *name*)
Returns value of tag that tracks a feature of the buffer

Parameters

- **tag** (*{None, str}*) – Name provided by the end-user code
- **default** (*{int, float}*) – Default value of feature if tag is missing.
- **name** (*str*) – What the tag represents. Used by the logging and exception machinery to create a useful message.

Returns **value** – Value of tag. If no tag is present, default is returned.

Return type {int, float}

pending()
Number of filled slots waiting to be read

read(*samples=None*)

Parameters **samples** (*int*) – Number of samples to read. If None, read all samples acquired since last call to read.

reset_read(*index=None*)
Reset the read index

2.6.5 tdt.convert

exception tdt.convert.**SamplingRateError**(*fs*, *requested_fs*)
Indicates that the conversion of frequency to sampling rate could not be performed.

tdt.convert.**convert**(*src_unit*, *dest_unit*, *value*, *dsp_fs*)
Converts value to desired unit give the sampling frequency of the DSP.

Parameters specified in paradigms are typically expressed as frequency and time while many DSP parameters are expressed in number of samples (referenced to the DSP sampling frequency). This function provides a convenience method for converting between conventional values and the 'digital' values used by the DSP.

Note that for converting units of time/frequency to n/nPer, we have to coerce the value to a multiple of the DSP period (e.g. the number of 'ticks' of the DSP clock).

Appropriate strings for the unit types:

fs sampling frequency

nPer number of samples per period

n number of samples

s seconds

ms milliseconds

nPow2 number of samples, coerced to the next greater power of 2 (used for ensuring efficient FFT computation)

```
>>> convert('s', 'n', 0.5, 10000)
5000
>>> convert('fs', 'nPer', 500, 10000)
20
>>> convert('s', 'nPow2', 5, 97.5e3)
524288
```

Parameters

- **src_unit** (*string*) –
- **dest_unit** (*string*) – Destination unit
- **value** (*numerical (e.g. integer or float)*) – Value to be converted

Returns converted unit

Return type numerical value

tdt.convert.**ispow2**(*n*)

True if n is a power of 2, False otherwise

```
>>> ispow2(5)
False
>>> ispow2(4)
True
```

tdt.convert.**nextpow2**(*n*)

Given n, return the nearest power of two that is >= n

```
>>> nextpow2(1)
1
>>> nextpow2(2)
2
>>> nextpow2(5)
8
>>> nextpow2(17)
32
```

2.7 DSP Server API

Since all TDT devices share the same connection with the computer, we can only talk to a single device at a time. Unfortunately, TDT's hardware drivers do not handle the requisite concurrent access issues, meaning that only one program and/or process can safely use the hardware connection (e.g. the optical or USB interface) at a time even if each process communicates with a different device. If you have two separate programs (one for generating the stimulus and one for data acquisition), TDTPy can handle the requisite concurrency issues.

TDTPy provides a server that manges all communication with the TDT hardware, allowing multiple client processes to communicate safely with the hardware via the server. Client processes communicate with the server via network-aware proxies of the RPcoX, PA5 and zBUS drivers. The network-aware drivers will relay all method calls to the server and block until the server returns a response. This is certainly not the most efficient way to handle the system (e.g. if the server is busy handling a request from another process it may take longer to receive a response).

This server is also useful for people who wish to run their code on a separate computer (e.g. Linux or Mac OSX) while maintaining a Windows computer to run the DSP server.

Note: This code definitely works, and is reasonably fast on a localhost connection. When I tested it via a client connecting using the wireless network there were some latency issues. There are likely many speedups that can be implemented, but I don't have the time to do so right now.

2.7.1 Running the server

To launch the server, go to the host computer and run the following command:

```
c:\\> python -m tdt.dsp_server :3333
```

The string *:3333* specifies which port the server listens on.

2.7.2 Code example

Running a client process is as simple as providing an address argument to DSPCircuit:

```
from tdt import DSPCircuit
address = ('localhost', 3333)
circuit = DSPCircuit('record_microphone.rcx', 'RZ6', address=address)
circuit.start()
```

Note: The circuit files must be stored on the client. The network proxy, RPcoXNET will handle transferring the circuit files to the server for you.

2.7.3 Converting existing code

Assuming your code uses *win32com.client* directly rather than using TDTPy's abstraction layer, the following code:

```
from win32com.client import Dispatch
iface = Dispatch('RPco.X')
iface.ConnectRZ6('GB', 1)
zbus = Dispatch('ZBUSx')
```

can simply be converted to a network-aware version via:

```
from tdt.dsp_server import RPcoXNET, zBUSNET
host, port = 'localhost', 3333
iface = RPcoXNET(address=(host, port))
iface.ConnectRZ6('GB', 1)
zbus = zBUSNET(address=(host, port))
```

Even if you prefer not to use the TDTPy abstraction layer (e.g. `DSPProject`, `DSPCircuit` and `DSPBuffer`), I highly recommend using TDTPy to obtain a handle to the ActiveX drivers since we have patched the win32com connection to speed up certain calls to the ActiveX drivers. To rewrite the code above that utilizes the patched version of the ActiveX drivers using TDTPy:

```
from tdt.util import connect_rpcox, connect_zbus
iface = connect_rpcox('RZ6', address=('localhost', 3333))
zbus = connect_zbus(address=('localhost', 3333))
```

If you're using the TDTPy abstraction layer, simply provide an address argument when initializing the DSPCircuit class:

```
from tdt import DSPCircuit
host, port = 'localhost', 3333
circuit = DSPCircuit('play_record.rcx', 'RZ6', address=(host, port))
```

2.7.4 Server implementation

The simplest way to handle concurrency was to create a remote procedure call (RPC) server. This RPC server will listen for connections from clients (either from the same computer or on a networked computer). Each client will initiate a persistent connection for the lifetime of the program and send requests via a TCP protocol. As these requests come in, the server will process them sequentially (thus handling concurrency issues).

A thread-based design for the server was considered; however, the bottleneck currently is in the optical interface I/O speed so it is unlikely that the additional hassle and overhead of threading will provide any significant performance gain.

The server is meant to be a relatively thin layer around the ActiveX device driver. Requests from clients are essentially passed directly to the ActiveX interface itself. To facilitate using this code we've created a network-aware proxy of the RPcoX client that passes off all RPcoX method calls directly to the RPC server. This allows you to use the server in your code without having to rewrite your code to use `DSPProject` or `DSPCircuit`.

The core client classes for communicating with the server are *RPcoXNET*, *PA5NET* and *zBUSNET* tha serve as a duck-typed proxy of the ActiveX drivers provided by TDT:

```
from tdt.dsp_server import RPcoXNET
client = RPcoXNET('localhost', 3333)
client.ConnectRZ5('GB', 1)
client.LoadCOF(cof_path)
```

Currently all method calls are simply relayed to the server, so what's really going on under the hood is that the call:

```
client.ConnectRZ5('GB', 1)
```

Is translated to:

```
client._send('RZ5', 'ConnectRZ5', ('GB', 1))
return client._recv()
```

Alternatively, the above can be achieved via:

```
from tdt.util import connect_rpcox
client = connect_rpcox('GB', 1, ('localhost', 3333))
```

The *.rcx files need to be stored on the client. They are uploaded to the server when the LoadCOF method is called.

Furthermore, TDTPy was as written as part of an initial progress towards a hardware abstraction layer. Your experiment code should not care whether you're using Tucker Davis' **'System 3'_** hardware, National Instruments DAQ platform, a high-quality audio card, or some combination of different vendors' hardware. A key goal of TDTPy is to begin progress towards an application programming interface (API) that can be implemented by Python wrappers around other hardware vendors' libraries. By building experiment code on top of TDTPy (rather than directly on top of TDT's ActiveX library), switching to another hardware platform should only require the following steps:

- Identifying (or writing) a wrapper around the vendor's library that supports the public API that TDTPy also supports.

- Writing the underlying microcode (e.g. a LabVIEW VI if you are switching to National Instruments' PXI) for the new hardware required to run the experiment.
- Changing your code to import from your new wrapper rather than TDTPy.

We have already built two programs, Neurogen and NeuroBehavior, on top of TDTPy with an eye towards ensuring that we can switch to a different hardware platform if needed.

CHAPTER 3

Key differences between TDTPy and OpenEx

Some people may note a number of similarities between the goals of the TDTPy and TDT's OpenEx platform. Both platforms are designed to streamline the process of setting up and running experiments by providing high-level functionality.

- TDTPy is open-source. OpenEx (despite the name) is not.
- Both OpenEx and TDTPy facilitate handling of buffer reads and writes provided you follow certain conventions in setting up your RPvds circuit. OpenEx requires strict conventions (e.g. you must give your tag a four-letter name with a special prefix). TDTPy allows you to use whatever names you like.
- Both TDTPy and OpenEx support running the hardware communication in a subprocess. However, OpenEx does not make the data immediately available. At best, there is a 10 second lag from the time the data is downloaded from the hardware to the time it is availabile to your script for plotting and analysis. TDTPy makes the downloaded data available immediately.
- OpenEx integrates with other components produced by TDT (OpenController, OpenDeveloper, OpenWorkbench, etc.). TDTPy currently does not offer the functionality provided by these other components.
- OpenEx requires the use of TDT's proprietary data format (TTank). In addition to being a proprietary, binary format, TTank imposes certain constraints on how you can save your data to disk. In contrast, TDTPy allows you to handle saving the data (i.e. you can dump it to a HDF5, XML, ASCII, CSV or MAT container).
- Integrating OpenEx with your custom scripts is somewhat of a hack. You must launch OpenEx then launch your script. TDTPy is part of your script.
- TDTPy comes with robust error-checking that catches many common coding mistakes (e.g. attempting to access a non-existent tag on the device) and a test-suite you can use to ensure your hardware is performing to spec.

CHAPTER 4

Roadmap

- In the write-test-debug routine of developing RPvds circuits, it would be very useful to have a GUI that allows you to interactively monitor and manipulate tag values well as visualize and manipulate data in the RPvds buffers. We can leverage Enthought's powerful Traits, TraitsGUI and Chaco packages for this purpose.
- Support processing pipelines for uploaded and downloaded data. This would be especially useful when running TDTPy as a subprocess to offload much of the processing overhead to a second CPU.
- Support streaming data from RPvds buffers to disk so the main process does not have to handle this step as well (requires a IO library that is thread/process safe).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index