Streaming Mode

Streaming mode reads data constantly from the device. When doing streaming, the PicoScope must be set to stream and then a callback must be defined to move the data from the scope to the computer. This is done through a number of function calls, and uses the following operational flow:

- Connect the device
- Set up the reading channels
- Set the PicoScope to fast streaming mode
- Poll the PicoScope for data and collect it in computer memory
- Stop and close the PicoScope
- Process the gathered data

With this setup, the amount of data to be collected is only restricted by the computer you are using. It is also possible to process the data as it is received. This makes streaming mode ideal for collecting data across a certain time period, collecting a large amount of samples, or processing data on the fly.

Connect the Device

Connecting the device is done as before. We will use the stub code from the Setup document.

```
from picosdk.ps2000 import ps2000

with ps2000.open_unit() as device:
    print('Device info: {}'.format(device.info))
```

Create Reading Channels

Setting up the reading channels is done similarly to in the block mode. A breakdown of this process can be found in Block Read

```
from picosdk.ps2000 import ps2000
from picosdk.functions import assert_pico2000_ok
from picosdk.PicoDeviceEnums import picoEnum


with ps2000.open_unit() as device:
    print('Device info: {}'.format(device.info))

    res = ps2000.ps2000_set_channel(
        device.handle,
        picoEnum.PICO_CHANNEL['PICO_CHANNEL_A'],
        True,
        picoEnum.PICO_COUPLING['PICO_DC'],
        ps2000.PS2000_VOLTAGE_RANGE['PS2000_50MV'],
    )
    assert_pico2000_ok(res)
```

Initialize Fast Stream Mode

In fast streaming mode, the PicoScope can take recordings and transmit them back to the computer at very high rates of up to millions per second. To set the PicoScope to fast streaming mode, the function `ps2000_run_streaming_ns` is to be used. This function takes 7 parameters.

- The device handle
- The sampling interval
- The sampling time unit
- The maximum samples to collect
- Whether to stop collecting once the maximum samples is hit
- The number of samples to aggregate together
- The maximum samples to store, before overwriting older values

For the time unit, a value of 2 will be used to signify nanoseconds. Since processing will be handled on the computer, aggregation will be set to 1, maximum samples will be set to `100_000` and the auto-stop will be set as `False`. The maximum samples to store just needs to be set high enough that the computer can copy all the values out before any get overwritten. Using a value of `50_000` will be more than sufficient for this.

```
res = ps2000.ps2000_run_streaming_ns(
    device.handle,
    500,
    2,
    100_000,
    False,
    1,
    50_000
)
assert_pico2000_ok(res)
```

Now, the function `ps2000_get_streaming_last_values` can be used to extract the values from the PicoScope's internal buffers into your computer's memory.

This function only takes 2 parameters, and the second parameter is a function that will be executed. This function should serve the sole purpose of copying the values into memory.

```
ps2000.ps2000_get_streaming_last_values(
    device.handle,
    callback
)
```

To create the `callback` function, the `ctypes` module can be used to convert a Python function into a C callable. `ctypes` exposes a factory function that is necessary here: `POINTER`. These can be used to convert a Python function into a C function. Another required meta-factory is the C function type factory, which is wrapped by the PicoSDK since the type is different on *nix systems to Windows systems.

```
# this should all be placed at the top of the python file
from ctypes import POINTER, c_int16, c_uint32
from picosdk.ctypes_wrapper import C_CALLBACK_FUNCTION_FACTORY
```

```
CALLBACK = C_CALLBACK_FUNCTION_FACTORY(
    None,
    POINTER(POINTER(c_int16)),
    c_int16,
    c_uint32,
    c_int16,
    c_int16,
    c_uint32
)

adc_values = []

def get_overview_buffers(
    buffers,
    _overflow,
    _triggered_at,
    _triggered,
    _auto_stop,
    n_values):

    adc_values.extend(buffers[0][0:n_values])

callback = CALLBACK(get_overview_buffers)
```

This creates `callback` as a C-compatible reference to the Python `get_overview_buffers` function. In turn, `get_overview_buffers` will copy the ADC values from behind the C pointer into a Python list. Then, the `ps2000_get_streaming_last_values` function needs to be called in a loop until the close condition is met. In this example, I'll be setting up to use a time-based condition, where the PicoScope will be set to collect data for 4 seconds before stopping. To do this, the built-in `time` module can be used.

```
import time

start_time = time.time()

while time.time() - start_time < 4.0:
    ps2000.ps2000_get_streaming_last_values(
        device.handle,
        callback
    )
```

This will collect values for 4 seconds total. Once collecting is finished, the PicoScope is no longer needed. Therefore, it should be stopped with `ps2000.ps2000_stop(device.handle)`.

```
from ctypes import CFUNCTYPE, POINTER, c_int16, c_uint32

from picosdk.ps2000 import ps2000
from picosdk.functions import assert_pico2000_ok
```

```
from picosdk.ctypes_wrapper import C_CALLBACK_FUNCTION_FACTORY

import time

CALLBACK = C_CALLBACK_FUNCTION_FACTORY(
    None,
    POINTER(POINTER(c_int16)),
    c_int16,
    c_uint32,
    c_int16,
    c_int16,
    c_uint32
)

adc_values = []


def get_overview_buffers(
    buffers,
    _overflow,
    _triggered_at,
    _triggered,
    _auto_stop,
    n_values):

    adc_values.extend(buffers[0][0:n_values])


callback = CALLBACK(get_overview_buffers)


with ps2000.open_unit() as device:
    print('Device info: {}'.format(device.info))

    res = ps2000.ps2000_set_channel(
        device.handle,
        picoEnum.PICO_CHANNEL['PICO_CHANNEL_A'],
        True,
        picoEnum.PICO_COUPLING['PICO_DC'],
        ps2000.PS2000_VOLTAGE_RANGE['PS2000_50MV'],
    )
    assert_pico2000_ok(res)

    res = ps2000.ps2000_run_streaming_ns(
        device.handle,
```

```
        500,
        2,
        100_000,
        False,
        1,
        50_000
    )
    assert_pico2000_ok(res)

    start_time = time.time()

    while time.time() - start_time < 4.0:
        ps2000.ps2000_get_streaming_last_values(
            device.handle,
            callback
        )

    ps2000.ps2000_stop(device.handle)
```

Now, the ADC values need converting into mV values, as before. However, since we are using a Python list and not a `ctypes` array, this function will be reimplemented. Converting an ADC value to an mV value involves linearly mapping the ADC space (which is a 16-bit integer) into the potential range (which in this case is ±500mV). This function is translated from `picosdk.functions`, with one change being that the bitness is preset since on the PicoScope 2000 Series, the bitness is 16.

```
def adc_to_mv(values, range_, bitness=16):
    v_ranges = [10, 20, 50, 100, 200, 500, 1_000, 2_000, 5_000, 10_000, 20_000]

    return [(x * v_ranges[range_]) / (2**(bitness - 1) - 1) for x in values]
```

Then, this function can be used to create a list of mV values. Using Matplotlib as before, a plot can be created. The difference is that instead of using the time values provided from the oscilloscope, time values must be estimated using a linear space.

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

mv_values = adc_to_mv(
    adc_values,
    ps2000.PS2000_VOLTAGE_RANGE['PS2000_50MV'])

fig, ax = plt.subplots()

ax.set_xlabel('time/ms')
ax.set_ylabel('voltage/mV')
ax.plot(np.linspace(0, 4000, len(mv_values)), mv_values)
```

```
plt.show()
```

`np.linspace` constructs a linear space across the x-axis. The idea here is that every reading from the oscilloscope should be equally spaced out, from the time that readings began to the time that the last reading was taken. `np.linspace(start, end, divisions)` takes the range of numbers from `start` to `end` and splits it `divisions` times. For example, `linspace(1, 4, 4)` would make `{ 1, 2, 3, 4 }`, and `linspace(1, 4, 7)` would make `{ 1, 1.5, 2, 2.5, 3, 3.5, 4 }`

The final code for this setup looks like so:

```
from ctypes import CFUNCTYPE, POINTER, c_int16, c_uint32

import matplotlib.pyplot as plt
import numpy as np

from picosdk.ps2000 import ps2000
from picosdk.functions import assert_pico2000_ok
from picosdk.ctypes_wrapper import C_CALLBACK_FUNCTION_FACTORY

import time

CALLBACK = C_CALLBACK_FUNCTION_FACTORY(
    None,
    POINTER(POINTER(c_int16)),
    c_int16,
    c_uint32,
    c_int16,
    c_int16,
    c_uint32
)

adc_values = []


def get_overview_buffers(
    buffers,
    _overflow,
    _triggered_at,
    _triggered,
    _auto_stop,
    n_values):

    adc_values.extend(buffers[0][0:n_values])


callback = CALLBACK(get_overview_buffers)
```

```
def adc_to_mv(values, range_, bitness=16):
    v_ranges = [10, 20, 50, 100, 200, 500, 1_000, 2_000, 5_000, 10_000, 20_000]

    return [(x * v_ranges[range_]) / (2**(bitness - 1) - 1) for x in values]


with ps2000.open_unit() as device:
    print('Device info: {}'.format(device.info))

    res = ps2000.ps2000_set_channel(
        device.handle,
        picoEnum.PICO_CHANNEL['PICO_CHANNEL_A'],
        True,
        picoEnum.PICO_COUPLING['PICO_DC'],
        ps2000.PS2000_VOLTAGE_RANGE['PS2000_50MV'],
    )
    assert_pico2000_ok(res)

    res = ps2000.ps2000_run_streaming_ns(
        device.handle,
        500,
        2,
        100_000,
        False,
        1,
        50_000
    )
    assert_pico2000_ok(res)

    start_time = time.time()

    while time.time() - start_time < 4.0:
        ps2000.ps2000_get_streaming_last_values(
            device.handle,
            callback
        )

    ps2000.ps2000_stop(device.handle)

    mv_values = adc_to_mv(
        adc_values,
        ps2000.PS2000_VOLTAGE_RANGE['PS2000_50MV'])

    fig, ax = plt.subplots()
```

```
    ax.set_xlabel('time/ms')
    ax.set_ylabel('voltage/mV')
    ax.plot(np.linspace(0, 4000, len(mv_values)), mv_values)

    plt.show()
```

The advantage of this approach is that the recording time can be easily increased to any number of seconds, or even smaller timescales by using the `time.time_ns()` function to instead measure the system time in nanoseconds. It can also be easily adapted to take a set number of samples:

```
...
    res = ps2000.ps2000_run_streaming_ns(
        device.handle,
        500,
        2,
        100_000,
        False,
        1,
        50_000
    )
    assert_pico2000_ok(res)

    # define the number of samples we want
    target_samples = 1_000_000

    start_time = time.time_ns()

    # repeatedly take samples until we have 1 000 000
    while len(adc_values) < target_samples:
        ps2000.ps2000_get_streaming_last_values(
            device.handle,
            callback
        )

    end_time = time.time_ns()

    ps2000.ps2000_stop(device.handle)

    mv_values = adc_to_mv(
        adc_values,
        ps2000.PS2000_VOLTAGE_RANGE['PS2000_50MV'])

    fig, ax = plt.subplots()

    # update the time unit from ms to ns since
    # the measured time is in nanoseconds
    ax.set_xlabel('time/ns')
```

```
    ax.set_ylabel('voltage/mV')
    ax.plot(np.linspace(0, end_time - start_time, len(mv_values)), mv_values)

    plt.show()
```

This example takes a minimum of 1 million samples. More samples will likely be taken, due to how the `ps2000_get_streaming_last_values` function returns blocks of readings. However, if you need a very specific number of samples rather than an approximate amount, the `adc_values` can be cropped to fit.