SIGLENT Oscilloscope Multi-Instrument Synchronization Application Guide

SIGLENT Oscilloscope Multi-Instrument Synchronization Application Guide

Introduction

The need for high channel count analog signal capture systems continues to increase. High speed, high throughput applications that drive key research are continually pushing the envelope for parallel processing, data bandwidth, and latency.

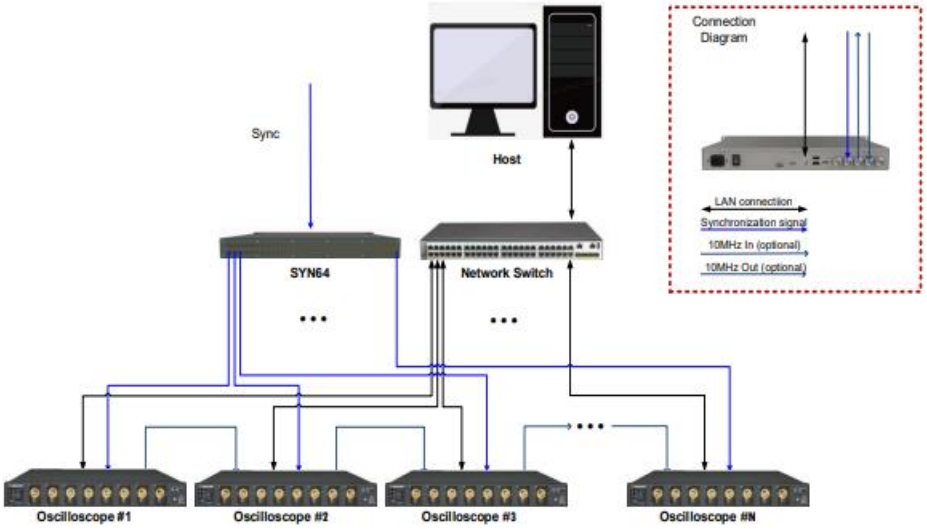
Digital oscilloscopes that combine high channel count, deep memory, and high sample rate can be deployed to make these measurements. High resolution front ends and low jitter are keys to getting the most out of a particular system design.

SIGLENT's new SDS5000X HD and SDS5000L series oscilloscopes have been designed with these system capabilities in mind. Each channel can sample up to 2.5 GSa/sec providing all channel bandwidth of up to 1 GHz. The SDS5000Ls are suited to rack and high channel applications with their headless, blade design. The SDS5000X HD provides the same capability with a built-in touchscreen for direct control.

Methodologies

There are several different system configurations to consider when designing a large channel count application. Let's discuss advantages and challenges of each approach.

Flexible Multi-channel High-speed Acquisition for very large Systems



Very large systems are really defined by use of the SYN64 trigger box. This device makes it possible to trigger up to 64 oscilloscopes with low jitter. This requires a clean trigger event incoming for parallel capture. Using this box, the trigger can be scaled up

to 512 channels (64 8-channel oscilloscopes). We will discuss timing requirements and data adjustments that can also apply to this system depending on how it is constructed. If built with matching cable lengths, this system can be very precise, but that becomes more difficult to do as the system becomes bigger.

Data management also becomes an issue at scale. It is recommended for any of these systems that all instruments be connected over LAN and parallelism in the software architecture can also help improve total system throughput. A system of any size that needs to archive multichannel data is best constructed with a *capture, pause, and transfer data* strategy. Sequence mode can be used to capture memory in segments, but once you have optimized the memory allocation, the design should be to fill or partially fill the memory across 1 or more trigger events and then pause to transfer the dataset. These instruments do not acquire data continuously while transferring it.

Designs for moderately sized systems

Moderately sized systems are defined here as being small enough to not require a trigger buffer like the SYN64. The actual size depends on cable quality, length, and layout. Here, we will demonstrate automating a 24 channel system with 3 instruments. It is clear this setup works well beyond that with proper design and timing expectations.

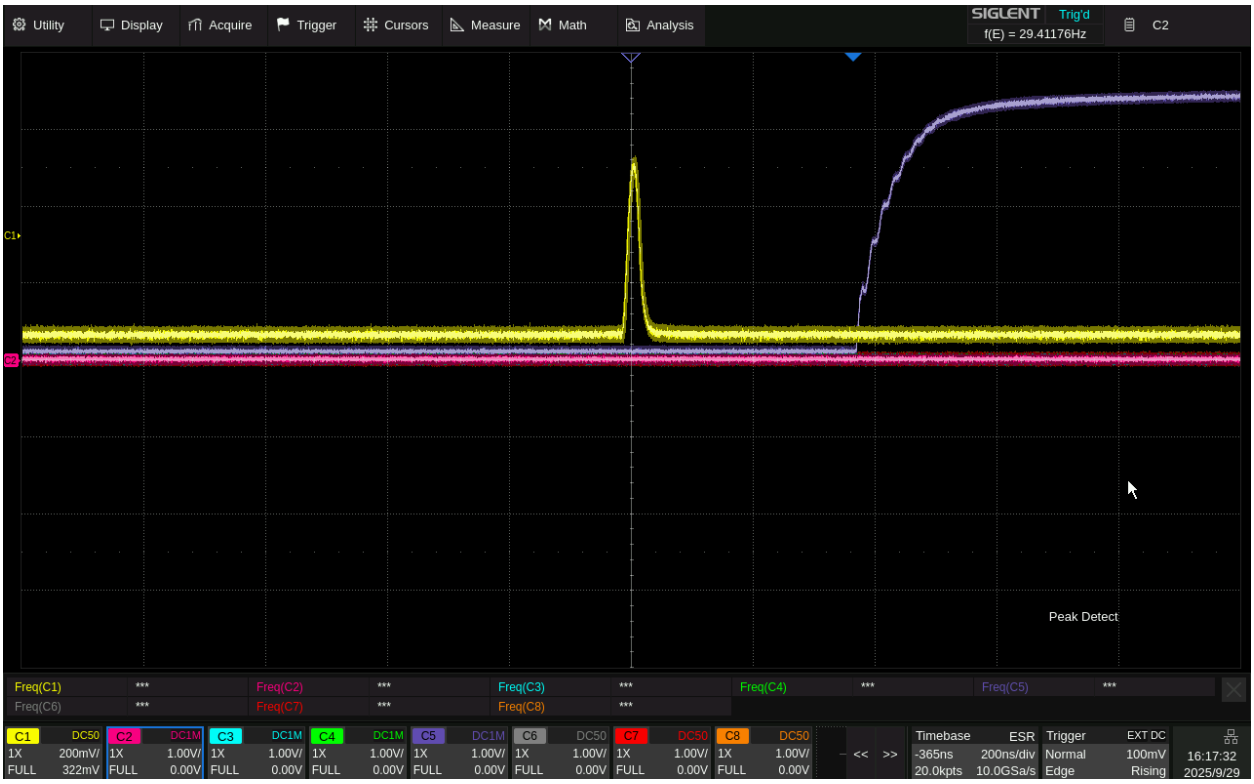
This system uses a SDS5000X HD as the *trigger master*. The instrument's trigger output is then routed to the external trigger inputs for the *slave instruments*. As this trigger signal is multiplexed out to multiple instruments, its signal integrity as a trigger source is essentially the limiting factor for system size. Even in *very large systems*, using one oscilloscope's trigger output is a common source for the trigger source to the SYN64 unless a clean trigger event already exists. When using an oscilloscope as the trigger source, regardless of the system size, the following timing and data manipulation strategies are relevant.

The primary advantage of building a system without the SYN64 trigger buffer is the substantial cost savings of the direct connection method.

In this system design, we used the SDS5000X HD as the master, 1 SDS5000L, 1 SDS6000L, and routed the trigger signal to an analog channel for monitoring purposes.

Again, depending on timing and size, the trigger signal used here can be duplicated to several trigger inputs until signal degradation adds too much trigger jitter. In this case, we used simple BNC tees to make the trigger connections with as short of cables as realistic. The trigger signal is a return to zero pulse. In a standard edge trigger mode for the master, the trigger output signal appears approximately 300-400 nanoseconds after the event. This signal is very low jitter however making it possible to account for the offset in several ways.

Here is a capture from one of the slave units showing a yellow pulse event. This is the same pulse that the master is triggering on. You can also see the purple trigger event that is being routed to the external trigger in on this unit.

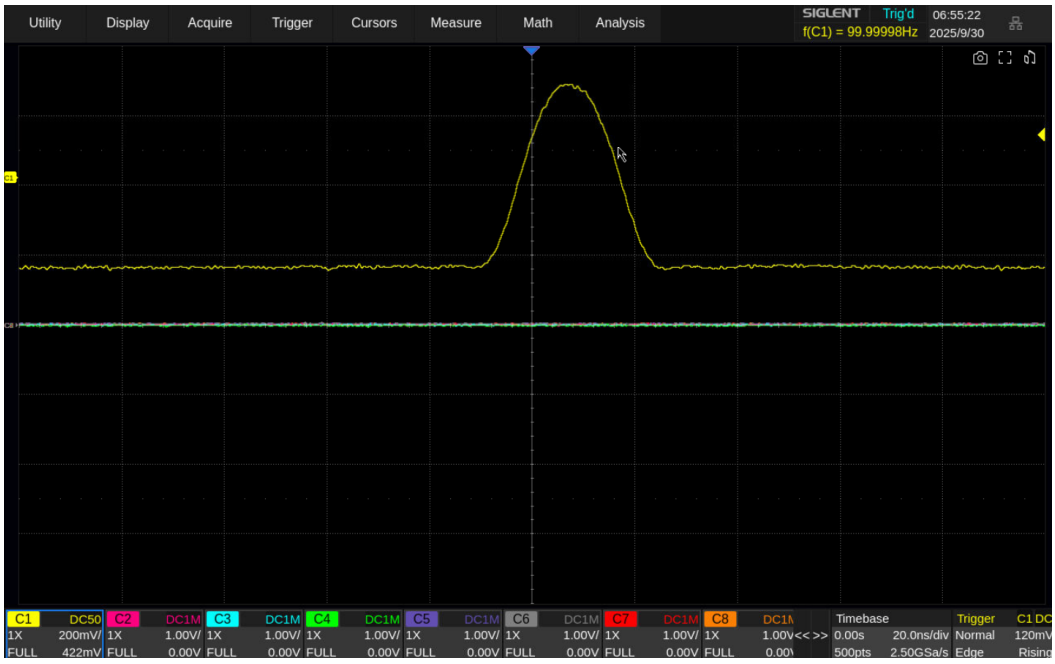


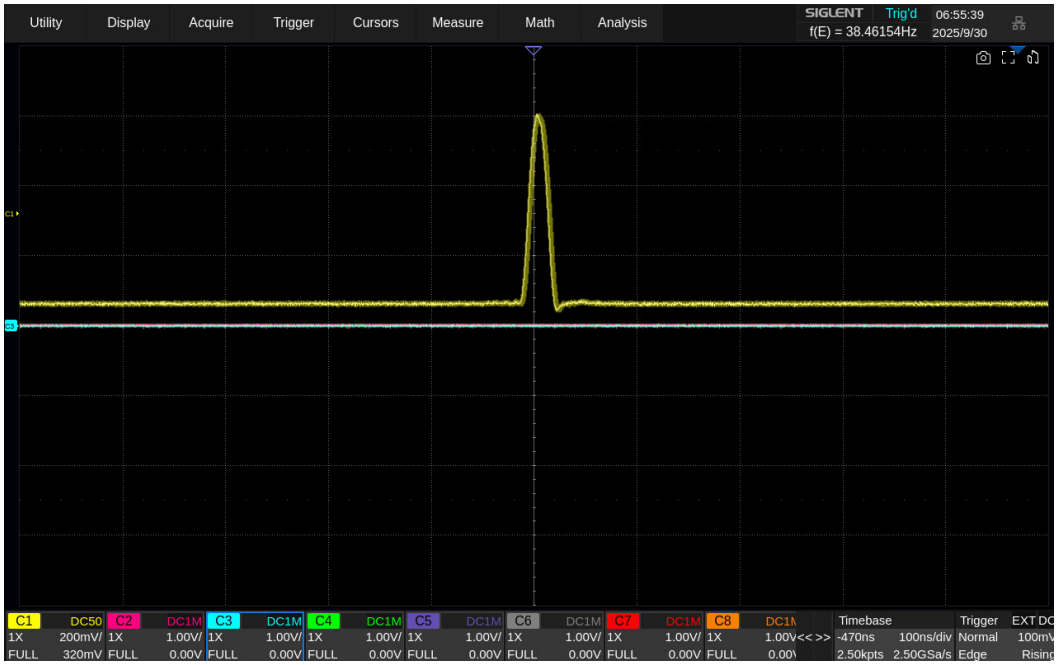
Persistence is set to infinite here, so you can see the trigger jitter on the yellow pulse is very small, on the order of ~10 nanoseconds. We also see that the trigger signal in purple arrives about 370 nanoseconds after the pulse in this setup. This is heavily dependent on the cable lengths, probing, and wiring scheme but in this case, most of that is within the master instrument.

The purple is what the trigger signal looks like when connected to the trigger output, 2 trigger inputs in HiZ mode, and one channel (5) in HiZ mode.

So, it is able to drive > 3 loads (more than 4 instruments total) for simultaneous capture. You can also see that it is beginning to degrade in quality and that careful cabling and trigger setup is important to success especially as the system size increases. You can likely further optimize that with setup specific trigger levels and gain settings. Over long periods of time you may see additional uncertainty with temperature or cable selection. Doing an occasional timing verification will help minimize this risk.

A similar view of the master instrument and the other slave instrument are shown here:





Looking carefully at the trigger delay settings that align the yellow pulse in center of the capture we can see that the slave devices have a timing offset of 370 and 470 nanoseconds respectively both with low jitter. The difference between these 2 values is essentially a matter of cable length. We will use those values for data correlation.

Given the delay timing of the trigger output, there are 2 recommended setups for cross-correlating data between instruments:

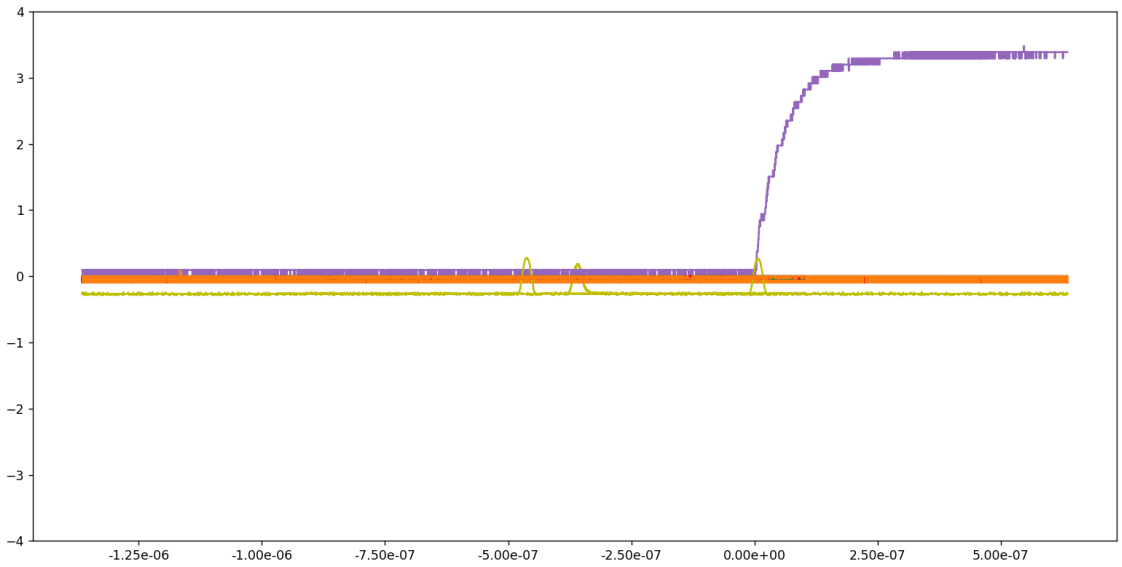
1. Post Processing Data Adjustment
    a. Simply, capture all of the data relative to scope trigger event and then subtract the time value from each point that is predetermined by the setup. On the scope shown above I would subtract the trigger delay from each reading to cross-correlate with the master channel data.

2. Standardized signal
    a. Alternately, you could dedicate a channel on each scope to a correlation marker. Ideally, this might be the trigger signal for the events, but as long as it appears in the record it can be any signal. Either triggering on these events or still triggering externally, instead of using the external trigger timing to make the adjustment you line up the marker events in each record for correct timing.
        i. This gives you increased accuracy in time due to lower jitter and offset as well as a visual marker.
        ii. This reduces the need for cross-correlation calibration or remembering the value for each scope given a certain cable arrangement.
        iii. This does likely require a little more care with wiring. To be precise, you would want equal path lengths from the source to each channel for the event marker.
        iv. Of course, this does force the system to dedicate more channels to synchronization, so there would be less for measurements.

In this example, we are using method 1, but we are showing the signals as you would see in method 2 in order to verify functionality. These are the yellow pulses that appear in each scope's record. Ultimately, if you were using method 1 you

would note the trigger delays and then rewire to your signals of interest with the same cable lengths.

At this point, we have the instruments set up and actively triggering. We can now use some programming automation to transfer the data to a computer. Using VISA over ethernet and SCPI in python this process is fairly straightforward using the included code (see appendix).
First, lets capture all of the channel data without any timing compensation:



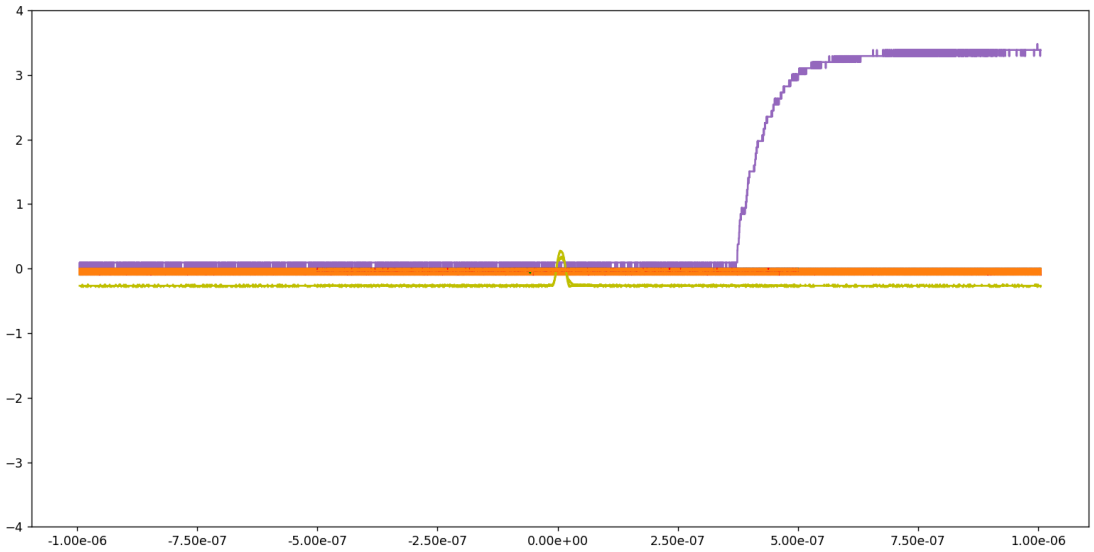
You can see the 3 pulses (that are the same pulse) all showing on CH1 in yellow on the 3 instruments are spread across time.

Then, we can add the time offsets:
In this program, lines 38 – 40 set the default time offset for the 3 instruments. Instrument 2 is the master, so it is set to zero:

```
37 #set time offsets
38 sds1time = 370E-9
39 sds2time = 0
40 sds3time = 470E-9
```

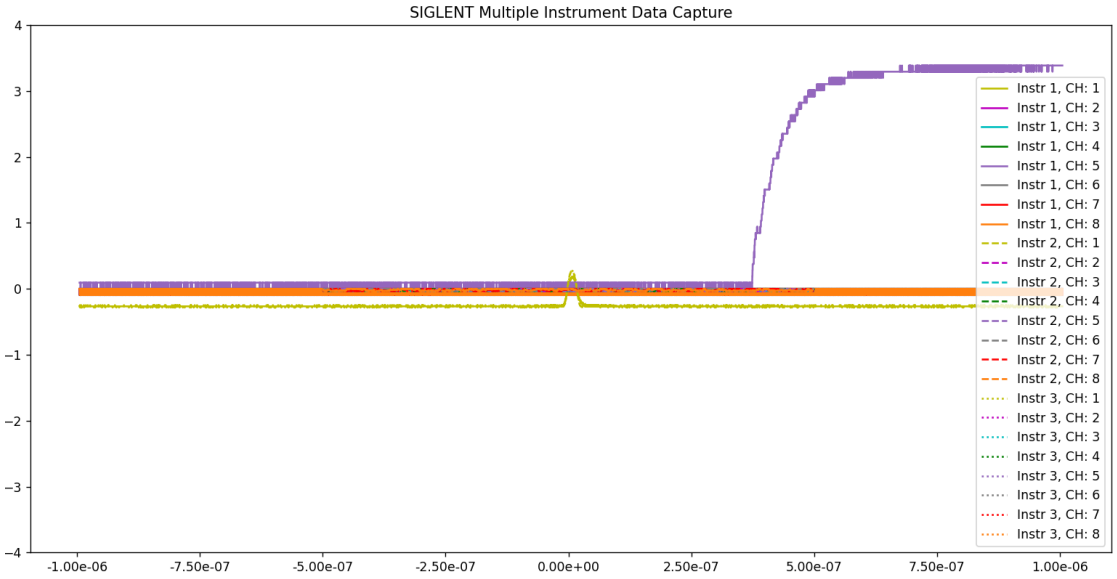
The others are set based on the trigger delays we found with the cables as used for correlation.

With these offsets active in the X axis data, we can graph as:

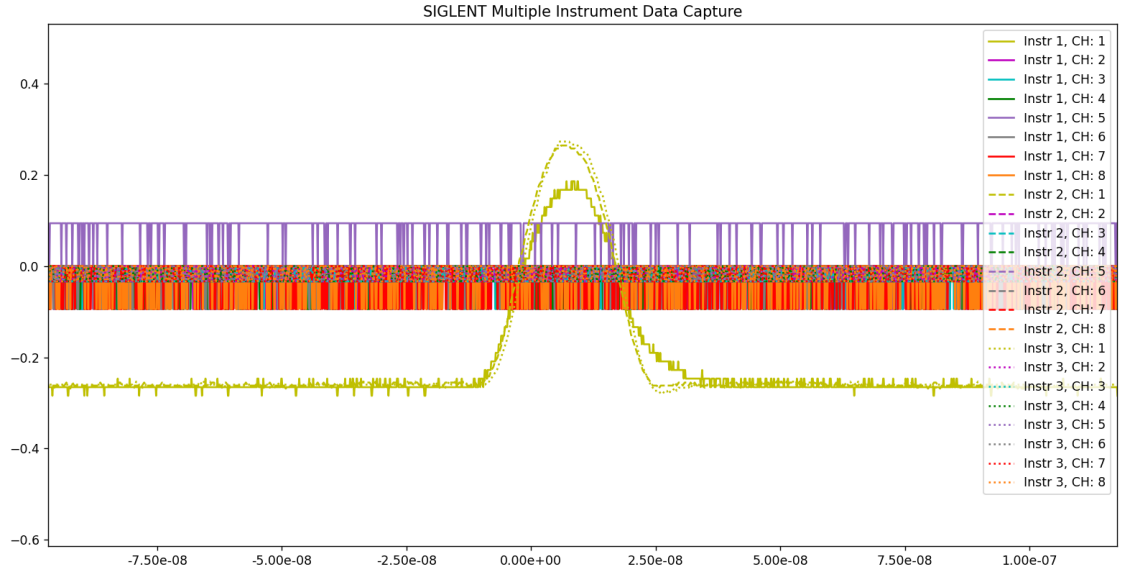


One nice feature of this methodology is that if you look closely at the instrument images above you will see one is using 500 points, one is set to 2500 points, and one set to 20,000 points. Because we are pulling the point data and charting it by absolute correlated time values all the signals are synchronized in time even with different numbers of points, vertical scales, and even sampling or horizontal scaling. This would enable us to chart digital channels as well that often have different sample rates and lengths. 16 logic channels are available on the SDS5000X HD and the SDS6000L series high channel count oscilloscopes.

In the final version of the test code in the appendix, we added some linestyling and a legend for this final result:



And of course, you can use the matplot tools to zoom in and check the timing:



These results correlate well from trigger event to trigger event creating a system capability for multichannel correlated signal capture.

Appendix: Python Example Program

```
import pyvisa
import numpy as np
import struct
from matplotlib import pyplot as plt
import matplotlib.ticker as mtick
import time

# ##########################
# Example Program designed for use with combination of SDS5000X
HD, SDS5000L, and SDS6000L.
# If you adjust for channel count, most configurations will work
with any SIGLENT HD Oscilloscopes.
# Some if statements below used for dealing with different
design decisions of the SDS6000L are commented as such.
#
# Before running scopes should be triggering in sequence from
the master to the slave units on their external triggers.
#
# ##########################


rm = pyvisa.ResourceManager()
print(rm.list_resources())
#select 3 scopes for data collection
sds1 = rm.open_resource(rm.list_resources()[4])
idnstr = sds1.query('*IDN?')
print("sds1 is: ",idnstr)
sds2 = rm.open_resource(rm.list_resources()[6])
idnstr = sds2.query('*IDN?')
print("sds2 is: ",idnstr)
sds3 = rm.open_resource(rm.list_resources()[5])
idnstr = sds3.query('*IDN?')
print("sds3 is: ",idnstr)

#identify masterscope
master = sds2
slave1 = sds1
slave2 = sds3

#set time offsets
sds1time = 370E-9
sds2time = 0
sds3time = 470E-9
```

```
#stop master:
#slaves will stop automatically, since their triggers come from
the master trigger out.
master.write(':TRIG:MODE NORM')
time.sleep(0.1)
master.write(':TRIG:MODE SING')
time.sleep(1)
slave1.write(':STOP')
slave2.write(':STOP')



#preconfig chart
# setup matplotlib initial plot
fig = plt.figure()

ax = fig.add_subplot(111)



#Loop for channel data collection
# for loop handles all channel speciailization like chart color,
style, labels, instrument connection, etc.
# loop has dat preamble and data collection per channel
totalchannels = 24
for chancount in range(1,totalchannels+1):
    #selects instrument, label, linestyle, and delaytime by
channel#.
    if chancount <9:
        sds = sds1
        plotlabel = 'Instr 1, CH: '
        delaytime = sds1time
        chlinestyle = '-'
    elif chancount < 17:
        sds = sds2
        delaytime = sds2time
        chlinestyle = '--'
        plotlabel = 'Instr 2, CH: '
    else:
        sds = sds3
        plotlabel = 'Instr 3, CH: '
        delaytime = sds3time
        chlinestyle = ':'
    chan = chancount % 8

    # set color for the mat plot lines based on channel #.
```

```
    if chan == 0:
        chan = 8
        plotcolor = 'tab:orange'
    elif chan ==1:
        plotcolor = 'y'
    elif chan ==2:
        plotcolor = 'm'
    elif chan ==3:
        plotcolor = 'c'
    elif chan ==4:
        plotcolor = 'g'
    elif chan ==5:
        plotcolor = 'tab:purple'
    elif chan ==6:
        plotcolor = 'tab:gray'
    else:
        plotcolor = 'r'

    #set plotlabel
    plotlabel += str(chan)

    #Set Channel
    chanstr = str(int(chan))
    #print(chanstr)

    #get channel info
    idnstr = sds.query('*IDN?')
    memdepth = sds.query(':ACQuire:POIN?')
    captured_points = int(float(memdepth[0:len(memdepth)-1]))

    #print(captured_points)

    #This sets it to 8 bit data capture mode for speed.
    width = 'BYTE'

    #select channel data and setup transfer.
    sds.write(':WAV:SOUR C' + chanstr)
    sds.write(':WAV:WIDT %s' % str(width))
    sds.write(':WAV:STAR 0')
    sds.write(':WAV:POIN %s' % str(captured_points))

    #get data preamble. needed to convert binary data to volts
and time.
    sds.write(':WAV:PRE?')
    recv = sds.read_raw()

    recv=recv[11:]
```

```
    #print(len(recv))

    #These values enumerate the time division from the value in
the data preamble.
    # This list can be found in the programming manual for each
series or model.
    #time base for 6000L
    if "SDS6" in idnstr:
        tdiv_enum = [100e-12,200e-12,500e-12, 1e-9,\
     2e-9, 5e-9, 10e-9, 20e-9, 50e-9, 100e-9, 200e-9, 500e-9, \
     1e-6, 2e-6, 5e-6, 10e-6, 20e-6, 50e-6, 100e-6, 200e-6,
500e-6, \
     1e-3, 2e-3, 5e-3, 10e-3, 20e-3, 50e-3, 100e-3, 200e-3,
500e-3, \
     1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]
    else:
        #use for SDS5L oor XHD or any others. May need other
options added for new series.
        tdiv_enum = [200e-12, 500e-12, 1e-9,\
     2e-9, 5e-9, 10e-9, 20e-9, 50e-9, 100e-9, 200e-9, 500e-9, \
     1e-6, 2e-6, 5e-6, 10e-6, 20e-6, 50e-6, 100e-6, 200e-6,
500e-6, \
     1e-3, 2e-3, 5e-3, 10e-3, 20e-3, 50e-3, 100e-3, 200e-3,
500e-3, \
     1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]

    WAVE_ARRAY_1 = recv[0x3c:0x3f + 1]

    #print('verification string: ',str(recv[0:8]))

    #print('raw code/div: ',str(recv[0x9c:0Xbb + 1]))

    # Sets the values in the preamble binary package to key
values for converting to volts and times.
    # This definition can be found in the programming manual.

    wave_array_count = recv[0x74:0x77 + 1] #count
    first_point = recv[0x84:0x87 + 1] #first point
    sp = recv[0x88:0x8b + 1] #point interval
    v_scale = recv[0x9c:0x9f + 1] #156 vert gain
    v_offset = recv[0xa0:0xa3 + 1] #160 vert offset
    interval = recv[0xb0:0xb3 + 1] #176 horizontal interval
    code_per_div = recv[0xa4:0Xa7 + 1] #164 code per div
    adc_bit = recv[0xac:0Xad + 1] #172 adc bit
    delay = recv[0xb4:0xbb + 1] #180 horiz offset
    tdiv = recv[0x144:0x145 + 1] #time base (table).
```

```
    probe = recv[0x148:0x14b + 1] #probe atten
    channel = recv[0x158:0x159 + 1] # 0-7 = c1-c8
    data_bytes = struct.unpack('i', WAVE_ARRAY_1)[0]
    point_num = struct.unpack('i', wave_array_count)[0]
    fp = struct.unpack('i', first_point)[0]
    sp = struct.unpack('i', sp)[0]
    interval = struct.unpack('f', interval)[0]
    delay = struct.unpack('d', delay)[0]
    tdiv_index = struct.unpack('h', tdiv)[0]
    probe = struct.unpack('f', probe)[0]
    channel = struct.unpack('h', channel)[0]
    vdiv = struct.unpack('f', v_scale)[0] * probe
    offset = struct.unpack('f', v_offset)[0] * probe
    code = struct.unpack('f', code_per_div)[0]
    adc_bit = struct.unpack('h', adc_bit)[0]
    tdiv = tdiv_enum[tdiv_index]

    # print options for preamble values for debugging:
    #print('raw tdiv: ', str(tdiv_index))
    # print('points: ', str(point_num))
    # print('first point: ', str(fp))
    # print('point interval: ', str(sp))
    # print('probe: ', str(probe))
    # print('V/div: ', str(vdiv))
    # print('offset: ',  str(offset))
    # print('#176 horizontal interval: ',  str(interval))
    # print('#180 horiz offset: ',  str(delay))
    # print('#time base (table): ',  str(tdiv))
    # print('#164 code per div: ',  str(code))
    # print('#172 adc bit: ',  str(adc_bit))
    # print('channel: ',  str(channel))
    # print(idnstr)

    #capture actual data
    sds.write(':WAV:DATA?')
    recv = sds.read_raw()
    time.sleep(1)
    valueslist = []
    valueslist.append(recv)

    # convert bits to volts
    header = int(chr(recv[1]))+2
    # print statements for header debugging:
    #print('length: ',len(recv[header:-2]))
    #print('start: ',valueslist[0][1:100])

    #possible alternate method for some scopes
```

```
    #values =
sds.query_binary_values(':WAV:DATA?',datatype='B',is_big_endian=
False)

    #print('buffer length: ',len(recv[header:-2]))
    convert_data = struct.unpack("%sb" % str(captured_points),
recv[header:-2])

    # data conversion values for debugging:
    #print('values: ',str(len(convert_data)))
    #print(convert_data)
    #print(type(convert_data))
    codes_data = list(convert_data)

    #voltage value (V) = code value *(vdiv /code_per_div) -
voffset.
    volts = []
    codeadjust = code

    if "SDS6" in idnstr or "BYTE" in width:
        codeadjust = (code/(pow(2,(adc_bit-8))))

    if "SDS6" in idnstr:
        #12 x ranges, otherwise screen  # 6L spreads bits across
24 divisions. other instruments just 8.
        # this value is used below to set ylim on the matplotlib
        rangemult = 12
    else:
        rangemult = 4


    #adjustment to correct data on 6L. Other instruments
unaffected.
    volts = [(x*(vdiv/codeadjust) - offset) for x in codes_data]

    #create timelist values
    #adjust timelist for known offset set above [delaytime]
    timelist = []
    for x in range(0,captured_points):
        timelist.append((delay-(tdiv*5)+x*interval)+delaytime)

    #add plot with channel data to main plot
    ax.plot(timelist,volts,markersize=2, label=plotlabel, color
=plotcolor, linestyle = chlinestyle)
    #set plot limits for sizing the view
    plt.ylim(-rangemult*vdiv-offset, rangemult*vdiv-offset)
```

```
    #update shell with channel status:
    print(plotlabel," data collected.")
    # End of FOR LOOP FOR Channel by Channel data collection

# All channels have been added to plot
#post process chart
ax.xaxis.set_major_formatter(mtick.FormatStrFormatter('%.2e'))

#start triggers again:
slave1.write(':TRIG:MODE NORM')
slave2.write(':TRIG:MODE NORM')
time.sleep(0.1)
master.write(':TRIG:MODE NORM')

#Close the VISA sessions with each instrument
sds1.close()
sds2.close()
sds3.close()

#Close VISA backend Resource Manager.
rm.close()

#Set Title and Legend to customize matplot chart. Then, show
chart.
plt.title('SIGLENT Multiple Instrument Data Capture')
plt.legend()
plt.show()
```