Log Your Instrument Test Data Directly to a Network Location

Introduction

Network-connected instruments are common in numerous modern test setups, and there are a great variety of software packages available to enable instrument configuration, test execution, and data collection and management. Additionally, many of the engineers and technicians running these tests will eventually back up their test data to some designated network location. It stands to reason that some of said engineers and technicians would seek out a means of eliminating the PC and software that seems to separate the instrument data from its eventual residence.

In the following sections we will walk the reader through one scenario that shows how to use the Test Script Processor (TSP) command set inherent in a majority of the Keithley Instruments product line to achieve the task at hand but focus on using with the Series 3706A Switch/Multimeter System. We start by reviewing the building blocks available for creating and destroying a TCP/IP sockets connection as well as formatting of our data to write out to the listener. This is followed with some instruction on how to create a simple Python-based server script (which will reside on a network host) to listen for the connecting (instrument) client and save the incoming information locally. We conclude by providing the steps to take in loading the script on to the instrument and running.

Note that we do not go into depth on all the benefits and power of the TSP command set and scripting. For more information, refer to your instrument's Reference Manual or visit tek.com for more information.

Create the TSP Script to Run on Your Instrument

While you can run a script directly on your instrument and perform direct network logging, you will still need a tool that helps you create the script file. Keithley's Test Script Builder is a free software that facilitates TSP scripting project creation, supporting direct instrument connection and interaction allowing you to run full scripts as well as the ability to step through your code in a script debug mode. If your are familiar with programming application development environments (ADE) that use and Eclipse back-end, then many of the features, options, and tools will be easy to adapt to.

In order to maintain focus on the topic at hand, we will refrain from going into the finer details of Test Script Builder installation and usage. You can find the "Using Test Script Builder (TSB)" article as part of the software installation by using the software's menu bar and navigating to *Help->Welcome->Keithley Test Script Builder*. You will also find a wealth of information about using TSP (and Lua) in your instrument's Reference Manual.

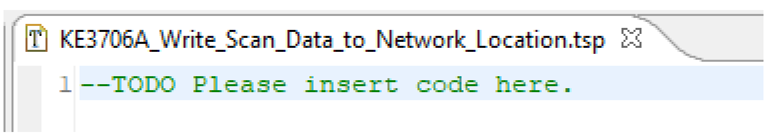
To create your TSP project in Test Script Builder:

1. Open **Test Script Builder** using the desktop icon or the Windows Start menu tools.
2. Navigate to ***File->New->Project***.
3. When the New Project Wizard appears, select **TSP Project** from the list provided then click **Next**.

4. In the Project name field, enter **KE3706A_Write_Scan_Data_to_Netork_Location** then click **Finish**. Note that it is important to use underscores and not blank spaces, or TSB will notify you that you are creating an invalid script name.
5. When presented with the "Open Associated Perspective?" dialog, click **Yes** to accept and establish the TSP perspective. This will enable syntax highlighting while creating your code and format checking each time you save.

At this point you will be presented the main.tsp file that acts as the foundation of your project coding efforts. However, every new project you create will start with a main.tsp file and this could make your work a little difficult to locate in your ADE interface as your scripting skills and files grow. To simplify, we will rename the file to match the project name.

6. In the **Navigator** pane (typically and by default on the left-hand side of the user interface), locate your project in the tree.
7. Click on the folder associated with the project to expose the **main.tsp** file.
8. Right-click on main.tsp to expose the pop-up menu options then select Rename.
9. When the dialog appears, enter the new name of the script file: **KE3706A_Write_Scan_Data_to_Network_Location.tsp**. Note that it is important to end the filename with the *.tsp extension to the Test Script Builder understands that this is a TSP script file.



Your updated script file name shown in the editor pane.

We can see (from the image above) that your next task will be to add your code. We will be using and referencing the code found in Appendix A, not covering the details of the 3706A multiplexer scan setup but focusing on the networking functions and calls.

The first function of interest is tspnet_connect() which is a wrapper function to help bundle some other checking and functionality.

- The `remote_ip` and `remote_port` are the IP address and target port number of the server or host we will be connecting to, respectively.
- The `initialization_string` is the default set of characters to be issued to the host from the instrument via the `tspnet.connect()` TSP command. The `tspnet.connect()` command will attempt to establish a socket connection to the host target.
- If the attempt to connect fails, the function will terminate and return the `host_id` with a nil value.
- If, however, the connection is successful then `tspnet.ipaddress` and `tspnet.termination` attributes are updated, and the `host_id` is returned with a handle to be used with other `tspnet` commands.

```
-- Initialize connection between the instrument and remote host
function tspnet_connect(remote_ip, remote_port, host_id,
initialization_string)
      host_id = tspnet.connect(remote_ip, remote_port, initialization_string)

      if host_id == nil then return nil end

      tspnet_ipaddress = remote_ip
      tspnet.termination(host_id, tspnet.TERM_LF)

      return host_id
end
```

With our connection function defined, let us review how it is used in the main script area. Below we find not just the `tspnet_connect()` wrapper function at play, but some other `tspnet`-specifc tools as well:

- `tspnet.reset()` is used to terminate any/all active `tspnet` sockets that the instrument may already be using. If you have the instrument that this script is running on physically connected to (vis the TSP bus) another TSP-enabled instrument, then that instrument's active `tspnet` sockets will also be terminated.
- The sockets response timeout value is updated to a given value; in this case the timeout is defined to be 5 seconds.
- Variables for the server's IP address and port number are defined, as well as a variable which will hold the handle to the server that will be reused in other `tspnet`-specific functions and wrappers.
- Note from earlier that our `tspnet_connect()` wrapper function includes a parameter where the user can pass a string that is used during the initial connection to the instrument. What we pass – because we will define the server socket code to expect it – is the name of the file that will be created on the server and used to hold all the measurement data we send. The `os.date()` function is built into the Lua-based embedded TSP scripting engine and allows us to format a string with specifiers identical to those used by [`strftime()`]() of the C programming language. We will be defining the end file in *.csv format so that all comma-separated strings sent to the instrument will be write-ready when they reach the server.
- Finally, we call our wrapper function, expecting successful connection and a handle that will be used for other communication actions with respect to the target server host.

```
-- Intitialize overall tsp-net configuration...
tspnet.reset()
tspnet.timeout = 5.0

-- Establish variables for remote connection
host_ipaddress = "192.168.1.111"
host_port = 60000
host_connection_id = nil
filename = os.date("data_%Y-%m-%d_%H-%M-%S.csv") -- timestamp the file name
```

```
-- Connect to the server/host
host_connection_id = tspnet_connect(host_ipaddress,
                                    host_port,
                                    host_connection_id,
                                    filename)
```

Our next user-defined wrapper function is `tspnet_write()` which was named to align more with the verbiage commonly used with test instrument communications – for example: “write”, “read”, and “query”. While we can surmise what might happen when we use `tspnet.execute()`, using terms like “write” or “send” remove any guesswork on the user’s part and helps prevent the need to open the Reference Manual to be certain.

```
-- Send command to controlled remote instrument
function tspnet_write(host_id, command)
      tspnet.execute(host_id, command .. "\n")
      if echo_cmd == 1 then
            print(command)
      end
end
```

This function is consumed much as we might expect. However, let us review what is happening just before we call this function in the main script:

- At this point we have already configured our 3706A to perform multiplexed scanning on a series of channels and trigger the scan to start. All channels will be scanned once every ten seconds for an hour, and measurements for each channel will be added to the reading buffer as each scan completes.
- The `for` loop is used to index through the reading buffer for a given scan, populating a string variable – `data_string` – with a series of comma-separated measurements. This loop will be executed for each scan that occurs, and the code that manages monitoring scan progress can be found in the main code body. We do not use the built-in `printbuffer()` command because it only moves buffered readings to the output queue.
- We then send the data to the server using our `tspnet_write()` wrapper, passing in the `host_connection_id` and `data_string` as arguments.

```
-- build the string of channels we want to write to the server
for i = startindex, endindex, 1 do
    if i == startindex then
        data_string = reading_buffer.readings[i]
    else
        data_string = data_string .. "," .. reading_buffer.readings[i]
    end
end

-- Send the scan data to the server
tspnet_write(host_connection_id, data_string)
```

Upon scanning completion, we will need a means to tell the server that this particular data logging instance is at an end and allow for proper client-connection termination on its end. As you will soon see, we simply write "done" to the server (because it is screening for this), then call our `tspnet_disconnect()` wrapper function.

```
-- After the scanning is complete, notify the server that we are done
-- and disconnect
tspnet_write(host_connection_id, "done")
tspnet_disconnect(host_connection_id)
```

There is nothing terribly spectacular to note about our disconnect wrapper other than it does some checking for a valid host ID handle prior to calling the native `tspnet.disconnect()` command. Still, we include it for the sake of being thorough.

```
-- Terminate the connection between the master and subordinate instrument
function tspnet_disconnect(host_id)
      if host_id ~= nil then
            tspnet.disconnect(host_id)
            host_id = nil
      end
end
```

Before transitioning to the next topic, we call attention to a useful way of providing operator feedback at the instrument front panel. Below is the series of commands executed just after our scan is triggered on the 3706A:

```
-- Update the instrument display
display.clear()
display.setcursor(1, 1)
display.settext("KE3706A DCV Scan")
display.setcursor(2, 1)
display.settext("$BSend Data Direct to Server")
```

We customize both the first and second lines of the instrument with the initial call to `display.setcursor()` and `display.settext()` placing the write cursor on line one, index position 1 then followed by writing the desired text, respectively. We see something similar with the follow-on calls to each command but with a focus on the display's second line. Note, however, in this text string that we begin with the "`$B`" character code. This tells the 3706A to cause all characters that follow the character code to blink. For more information on other display character codes, refer to `display.settext()` in the instrument Reference Manual.

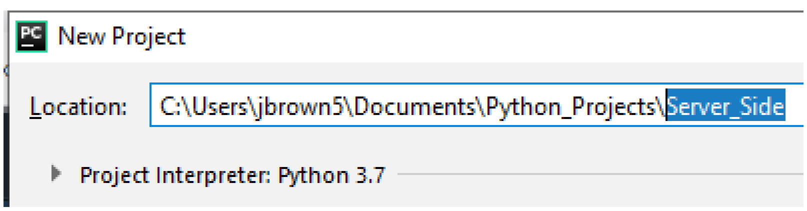
Create a Simple Server to Run on a Remote Host PC

Our simple server is derived from what you might find in many introductory “Computer Networks” college courses. This same sort of information is also freely available on the internet and we have made a link available in the Resources section of this document. Bear in mind that our goal for this piece of the solution is simplicity, which is why you will find that there is not a lot of code necessary to accomplish out task.

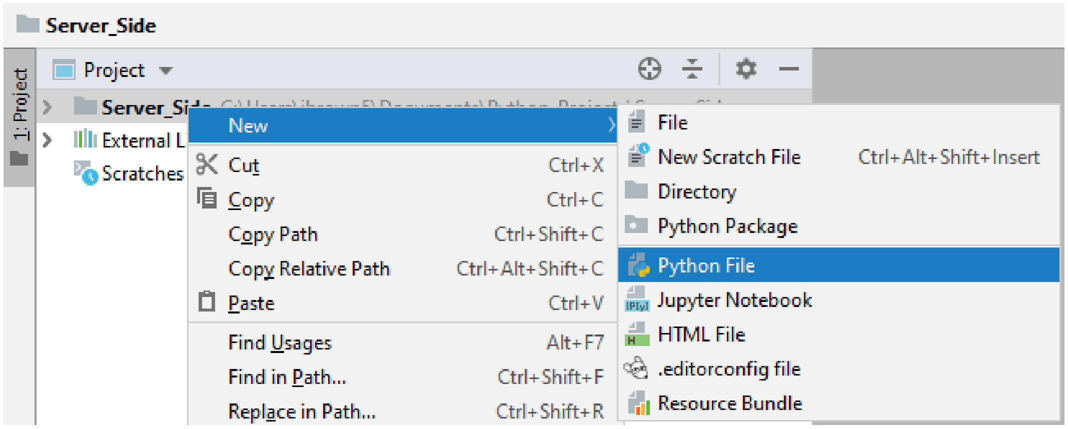
While all the example code we use for this application is available in Appendix B, you will still need to acquire the Python programming language for your PC as well as a powerful source code editor like PyCharm Community which is available free of charge from the team at JetBrains. Because Python is a cross-platform programming language, most of the code we write can be run on Windows or Linux systems. For help in getting started with the installation of both of these tools as well as a brief introduction on remote instrument control using Python, refer to “Getting Started with Instrument Control Using Python 3” which is available via the Tektronix GitHub repository. Note that direct links to all the tools referenced here are available in the Resources section of this document.

Let us start by creating a new project in PyCharm that will hold our code:

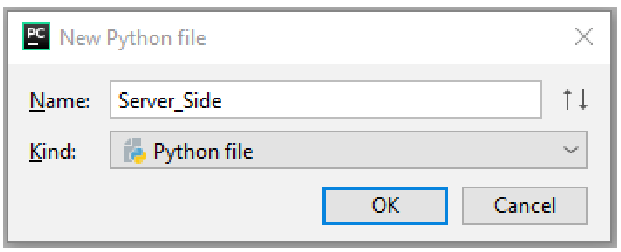
1. Open **PyCharm** using the desktop icon or the Windows Start menu tools. (For launching PyCharm on a Linux system, use the desktop icon or refer to the internet for guidance specific to your Linux distribution.)
2. Click on the **+ Create New Project** option in the default dialog.
3. Add the name of your Python project that defines our objective in the area highlighted in the Location input field. We will set this to **Server_Side**. Also ensure that your Project Interpreter is set to use your choice Python installation. (Note, below, that Python 3.7 is selected.) Click the **Create** button.



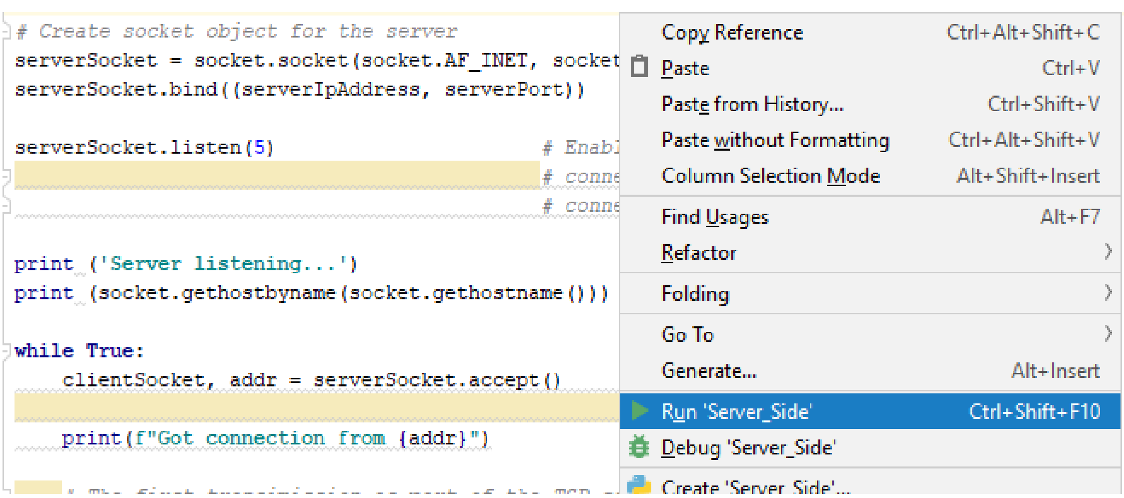
4. In the Project pane (on the left-hand side of the user interface by default), right-click on the Server_Side folder to reveal the pop-up menu and select **New->Python File** from the options provided.



5. Enter **Server_Side** in the New Python file dialog presented.



6. Copy the code found in Appendix B into the editor window. Note that the PyCharm environment may highlight areas in the editor and indicate warnings. While not necessary, you may want to edit the contents to eliminate the warnings and adhere to the PEP 8 coding style guidelines promoted by the PyCharm ADE.
7. To run or debug the code, right-click in the editor window to reveal the pop-up and select either **Run 'Server_Side'** or **Debug 'Server_Side'**.



Our main code area starts off with the instantiation of a socket object, `serverSocket`, then is bound to its IP address and a port through which will receive the incoming data. The server is then configured to permit up to five simultaneous incoming connections, though we will only be using one in our example.

```
# Create socket object for the server
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind((serverIpAddress, serverPort))

serverSocket.listen(5)                        # Enable our server to accept up to 5
                                              # connections before refusing new
                                              # connections
```

The script program then enters a `while()` loop that detects an incoming client connection (which will come from our 3706A scanning script). The local terminal will report that the connection is made, then the server will ready itself for the first incoming message which we defined as the data filename in the 3706A script. This `while()` loop will continue until the client connection is closed.

```
while True:
    clientSocket, addr = serverSocket.accept()     # Establish connection with
                                                   # client.
    print("Got connection from {0}".format(addr))

    # The first transimission as part of the TSP connect will be the instrument
    # file name that will be written to during data transmission.
    filename = clientSocket.recv(1024).decode()
```

We then fall into another `while()` loop that handles the incoming scan data:

- The server is queued up to receive an incoming message from the client, and we remove all trailing line feed characters from the sent message (with `rstrip()`) so the contents can be evaluated.
- The `while()` looks for a `nil` string first to determine if its looping action should be broken. Additionally, if the incoming message reads “done” then the loop will be broken.
- If the incoming message is our reading data then we pass the message string into the `write_data_to_file()` function that logs the data to the file and location defined by `filename`.

```
# Subsequent writes will contain a string of data which will be written to
# the file until the sender issues a "done" string.
data_to_write = clientSocket.recv(1024).decode()
while (data_to_write):
    print(data_to_write.rstrip())
    if(data_to_write.__contains__("done")):
        break
    write_data_to_file(filename, data_to_write.rstrip())
    data_to_write = clientSocket.recv(1024).decode()
```

Once the looping is broken, the client connection is terminated and a notification is sent to the server terminal window.

```
# Close the active client connection to allow for other incoming connections
# to be made.
clientSocket.close()
print("Closed connection...")
```

Load and Run the Script on Your Instrument

Because our goal is to run our network-logging script on our instrument in the absence of the PC, let us now cover the process of migrating and loading the script onto our target instrument. The first step is to move a copy of the script on to a USB flash drive. Locate the script on your hard drive via the default project path established by the TSB software install and in your specific project folder. For instance:

```
C:\Users\username\Keithley Test Script
Builder\Workspaces\workspace\KE3706A_Write_Scan_Data_To_Network_Location
```

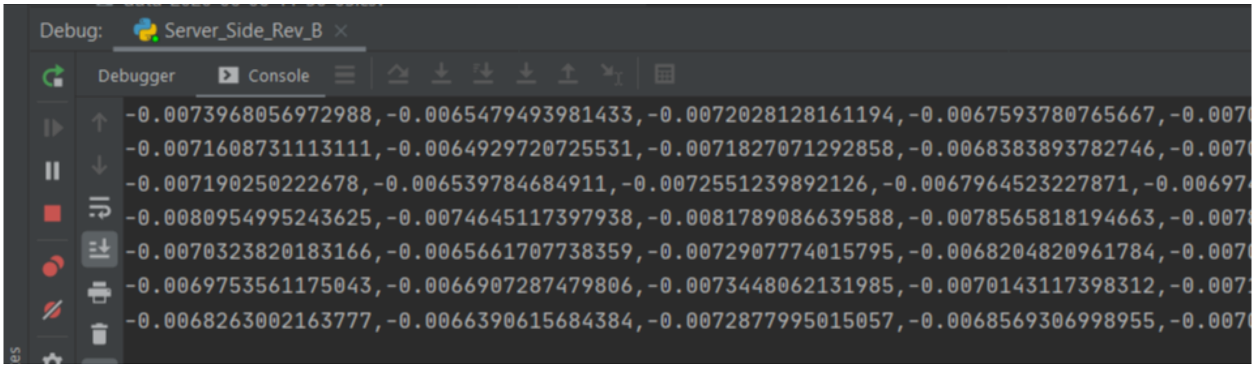
Insert a USB flash drive into a compatible port on your PC then copy the **KE3706A_Write_Scan_Data_to_Network_Location.tsp** script on to the root of a USB flash drive. Remove the USB flash drive from the PC and plug it into the front port on the 3706A.

The next step is to load the script into the instrument memory. Using the buttons on the 3706A, navigate to **MENU->SCRIPT->LOAD->USB**. Use the **rotary knob** to locate and identify the script by name; it will be the active selection when the script name is visible and blinking on the display. To activate loading, depress the **rotary knob**. Note that the script will be loaded as **"<anonymous>"**.

Fully exit out of the script menu and back to the main front panel by pressing the **EXIT** button.

Recall that the script will run and provide a display update to notify your that the scan operation is in progress, blinking the "Send Data Direct to Server" message on the second line of the instrument display.

During the run, you can see the incoming data arrive at the server, being printed in the terminal window where the Python server script is running.



Additionally, you will be able to locate the data file either local to where the Python server script is running, or via the system path specified in the data file name.

Upon scan completion (in this case the scan runs for an hour), the display will indicate “Scanning Complete!!!”.

Conclusion

Removing the PC from a data logging application is useful in a number of situations. We have provided some high-level walk-throughs on constructing your instrument script code to run independent of the PC as well as offering a very basic Python server script that can be deployed on a remote network host. We complete the setup with instruction on how to move the 3706A TSP script on to the instrument and run it.

With all this in place, we take a moment of reflection to ponder how you might enhance this solution further.

- The client-server connection is highly optimistic in that we are assuming success and there are no provisions in checking for the integrity of the first and continuous connection. Code can be established on both the client and server sides to handle erroneous scenarios.
- Part of the series of tspnet wrapper functions includes `tspnet_query()` and `tspnet_read()` which might be leveraged for better handshaking between the instrument and server, further adding to the robustness of the code.

- Likewise, the server-side script should be enhanced such that it implements the code/commands necessary to send feedback to the client.
- The server-side script must be terminated by a user who as access to the server and its network location. However, it might make more sense to elaborate on the server script to have a means of closing out the application remotely.

Resources

Below is a listing of software and articles that were used in creating this document:

- Keithley Test Script Builder (TSB): https://www.tek.com/keithley-test-script-builder
- Keithley Series 3706A Switch/Multimeter System: https://www.tek.com/keithley-switching-and-data-acquisition-systems/keithley-3700a-systems-switch-multimeter
- Lua: https://www.lua.org/
- Python: https://www.python.org/
- JetBrains PyCharm: https://www.jetbrains.com/pycharm/
- “Network Programming – Server & Client B: File Transfer”: https://www.bogotobogo.com/python/python_network_programming_server_client_file_transfer.php
- “Getting Started with Instrument Control Using Python 3”: https://github.com/tektronix/keithley/tree/master/Instrument_Examples/Instructables/Get_Started_with_Instr_Control_Python

Appendix A: Full TSP Script Code for the Series 3706A Scan Logging

```
--[[ ==============================================================================
	Start TSP-Net Function Wrappers
]] -- =============================================================================
-- ================================================================================
-- Initialize connection between the instrument and remote host
function tspnet_connect(remote_ip, remote_port, host_id, initialization_string)
	host_id = tspnet.connect(remote_ip, remote_port, initialization_string)
	if host_id == nil then return nil end
	tspnet_ipaddress = remote_ip
	tspnet.termination(host_id, tspnet.TERM_LF)
	return host_id
end

-- Send command to controlled remote instrument
function tspnet_write(host_id, command)
	tspnet.execute(host_id, command .. "\n")
	if echo_cmd == 1 then
		print(command)
	end
end

-- Query data from the controlled instrument and return as a string
function tspnet_query(host_id, command, timeout)
	timeout = timeout or 5.0 --Use default timeout of 5 secs if not specified
	tspnet_write(host_id, command)
	while tspnet.readavailable(host_id) == 0 and timer.gettime() < timeout do
		delay(0.1)
	end
```

```
        return tspnet.read(host_id)
end

-- Terminate the connection between the master and subordinate instrument
function tspnet_disconnect(host_id)
        if host_id ~= nil then
                tspnet.disconnect(host_id)
                host_id = nil
        end
end
--[[
        End TSP-Net Function Wrappers
]]--
--[[
        Start Scan Configuration Function Wrappers
]]--
function configure_dcv_scan(nplc, dcv_range, use_input_divider, scan_interval,
         scan_channels, scan_count)
        reset()
    dmm.func = dmm.DC_VOLTS                    -- Set measurement function
    dmm.nplc = nplc                            -- Set NPLC
    if dcv_range < 0.001 then                  -- Set Range
        dmm.autorange = dmm.ON
    else
         dmm.autorange = dmm.OFF
         dmm.range = dcv_range
    end

    dmm.autodelay = dmm.ON                     -- Ensure Auto Delay is enabled
    dmm.autozero = dmm.ON                      -- Enable Auto Zero
    if use_input_divider == 1 then             -- Apply the 10M input divider as needed
        dmm.inputdivider = dmm.ON
    else
        dmm.inputdivider = dmm.OFF
        end

    dmm.configure.set("mydcvolts")             -- Save Configuration
    dmm.setconfig(scan_channels, "mydcvolts")          -- Assign configuration to channels

    channel.connectrule = channel.BREAK_BEFORE_MAKE

    if scan_interval > 0.1 then
         -- Establish the settings that will apply the interval between the start
         -- of scans
        trigger.timer[1].reset()               -- Ensure the timer gets to a
                                               -- known relative time start
                                               -- point
         trigger.timer[1].count = 0                    -- No reapeating timer events
         trigger.timer[1].delay = scan_interval        -- Apply the anticipated scan
                                                       -- interval
         trigger.timer[1].stimulus = scan.trigger.EVENT_MEASURE_COMP
         trigger.timer[1].passthrough = false          -- Trigger only initiates the
                                                       -- delay
         trigger.blender[1].reset()                    -- Configure the blender
                                                       -- stimulus...
         trigger.blender[1].orenable = true            -- ... for OR'ing operation
         trigger.blender[1].stimulus[1] = trigger.timer[1].EVENT_ID -- ... to
                                                       -- respond/notify upon a timer
                                                       -- event
         trigger.blender[1].stimulus[2] = scan.trigger.EVENT_SCAN_READY -- ... or
                                                               -- when then scan is ready
                                                               -- (configured)
         scan.trigger.arm.stimulus = trigger.blender[1].EVENT_ID     -- Key triggering
                                                               -- off of the blender event
    end

    scan.create(scan_channels)                                 -- Create the scan
    scan.scancount = scan_count                                -- Set the Scan Count
    reading_buffer = dmm.makebuffer(scan.scancount * scan.stepcount)       -- Configure
                                                                           -- the buffer
    scan.background(reading_buffer)                    -- Execute Scan and save to buffer
```

```
    -- Update the instrument display
    display.clear()
    display.setcursor(1, 1)
    display.settext("KE3706A DCV Scan")
    display.setcursor(2, 1)
    display.settext("$BSend Data Direct to Server")
    return reading_buffer
end
--[[
       End Scan Configuration Function Wrappers
]]--
-- ================================================================================
--[[ ******************************************************************************

               MAIN PROGRAM STARTS HERE

*********************************************************************************
]]
scanchannels = "1001:1020"           -- Define the channels to scan here. Note the
                                     -- following format possibilities...
                                     --   1001:10060 - All channels starting with
                                     --     1001 and ending with 1060
                                     --   1001,1002,1004 - Just channels 1001,
                                     --     1002, and 1004
                                     --   1007:1010,1021,1031:1040 - Channels
                                     --     1007 through 1010, channel 1021, and
                                     --     channels 1031 through 1040
rangedcv = 10                        -- Define the DCV range. If auto-ranging is
                                     --   desired, pass 0
useinputdivider = 1                  -- 1 = True; 0 = False
scancount = 360                      -- Number of times to run the scan
scaninterval = 10                    -- Delay between the start of each scan (if
                                     -- needed)
nplc = 1

reading_buffer = configure_dcv_scan(nplc, rangedcv, useinputdivider, scaninterval,
                                                                    scanchannels, scancount,
reading_buffer)
channelcount = scan.stepcount
startindex = 1
endindex = channelcount
total_readings_count = 0
target = channelcount * scancount
delay(0.5)
data_string = ""

-- Intitialize overall tsp-net configuration...
tspnet.reset()
tspnet.timeout = 5.0

-- Establish variables for remote connection
host_ipaddress = "192.168.1.111"
host_port = 60000
host_connection_id = nil
filename = os.date("data_%Y-%m-%d_%H-%M-%S.csv")

-- Connect to the server/host
host_connection_id = tspnet_connect(host_ipaddress, host_port, host_connection_id,
                                                                filename)

-- Extract readings while the scan is running....
while(total_readings_count < target) do
    vals = reading_buffer.n

    -- wait unit the buffer is ready with as scan's worth of readings
    while(vals < endindex) do
        delay(0.1)
        vals = reading_buffer.n
    end
```

```
    -- build the string of channels we want to write to the server
    for i = startindex, endindex, 1 do
        if i == startindex then
                data_string = reading_buffer.readings[i]
        else
                data_string = data_string .. "," .. reading_buffer.readings[i]
        end
    end

    -- Send the scan data to the server
    tspnet_write(host_connection_id, data_string)

    -- Update the variables that handle the buffer indexing
    startindex = startindex + channelcount
    endindex = endindex + channelcount
    total_readings_count = total_readings_count + channelcount
end


-- After the scanning is complete, notify the server that we are done and disconnect
tspnet_write(host_connection_id, "done")
tspnet_disconnect(host_connection_id)

display.clear()
display.setcursor(1, 1)
display.settext("KE3706A DCV Scan")
display.setcursor(2, 1)
display.settext("$RScanning $BComplete!!!")
```

Appendix B: Full Python Server Script Code

```
# Server_Side.py
import socket                              # Import socket module
import sys
import os.path
import operator
import time


def write_data_to_file(output_data_path, dataStr):
    # This function writes the string data to the
    # target file.
    ofile = open(output_data_path, "a")   # append the target data
    dataStr = "{0}\n".format(dataStr)
    ofile.write(dataStr)

    ofile.close()   # Close the data file.
    return

# ==================================================================================
#    MAIN CODE STARTS HERE
# ==================================================================================
serverIpAddress = socket.gethostname()      # Get local machine name
serverPort = 60000                          # Define the port to which the client
                                            # will use

# Create socket object for the server
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind((serverIpAddress, serverPort))

serverSocket.listen(5)                      # Enable our server to accept up to 5
                                            # connections before refusing new
                                            # connections

print ('Server listening...')
print (socket.gethostbyname(socket.gethostname()))
```

```
while True:
    clientSocket, addr = serverSocket.accept()  # Establish connection with
                                                # client.
    print("Got connection from {0}".format(addr))

    # The first transimission as part of the TSP connect will be the instrument
    # file name that will be written to during data transmission.
    filename = clientSocket.recv(1024).decode()

    # Subsequent writes will contain a string of data which will be written to
    # the file until the sender issues a "done" string.
    data_to_write = clientSocket.recv(1024).decode()
    while (data_to_write):
        print(data_to_write.rstrip())
        if(data_to_write.__contains__("done")):
            break
        write_data_to_file(filename, data_to_write.rstrip())
        data_to_write = clientSocket.recv(1024).decode()

    # Close the active client connection to allow for other incoming connections
    # to be made.
    clientSocket.close()
    print("Closed connection...")

# We can close the server socket, but have the option to leave open and
# run indefinitely so it is always listening for incoming connections
# and data.
serverSocket.close()
```

Revisions

Revision	Date	Authored by	Notes
1.0	2020-06-08	Josh Brown	