Tektronix®

"One Click" Python Wrapper Creation with TSP Toolkit

APPLICATION NOTE



Introduction

When discussing high-density parallel testing solutions, the biggest question is how to ensure fast throughput and tight synchronization at scale. The larger the system, the more channels to synchronize, and the more important optimization becomes. Any delays or inefficiencies in the automation will compound as the system expands, leading to larger timing losses as channel count increases. The way we guarantee tight synchronization and fast throughput when automating Tektronix Precision DC instrumentation is by using Test Script Processing or TSP.

TSP is an instrument automation command set and programming language that enables scripts to execute directly on TSP-enabled instruments. With TSP scripting, instruments can be turned into purpose built tools that carry out test sequences and data analysis algorithms on the instrument. TSP also gives a virtual backplane, TSPLINK, for both communication and triggering with other TSP enabled products. These scripts can automate test routines, run sweeps, collect and process data, store functions and variables, respond to and generate digital triggers, and make decisions without relying on continuous communication with an external computer.

From applying calculations to measurement data to coordinating trigger sweeps across multiple instruments, TSP scripting brings automation directly to the instrument, reducing communication overhead and simplifying test workflows. TSP Toolkit provides an environment for creating and working with these scripts.

Unlocking the full potential of TSP-enabled instrumentation has never been easier. Tektronix's very own TSP scripting tool, TSP Toolkit, is equipped with real-time error checking, intelligent autocompletion, inline help, an on-instrument debugger, and other features which make writing TSP scripts to automate the MP5000 Series Modular Precision Test System and other TSP enabled instruments a powerful and intuitive experience.

This application note will discuss the latest TSP Toolkit feature: Python Interoperability, and how it simplifies automation by introducing the ability to generate a python wrapper file for a TSP script with a single mouse click – giving user's access to all the synchronization and timing benefits of on-instrument TSP scripting without the need to overhaul or replace an existing python automation ecosystem.

How do TSP & Python Work Together?

The concept of automating a test system using two different programming languages may sound strange or unintuitive at first but is a simple concept in practice.

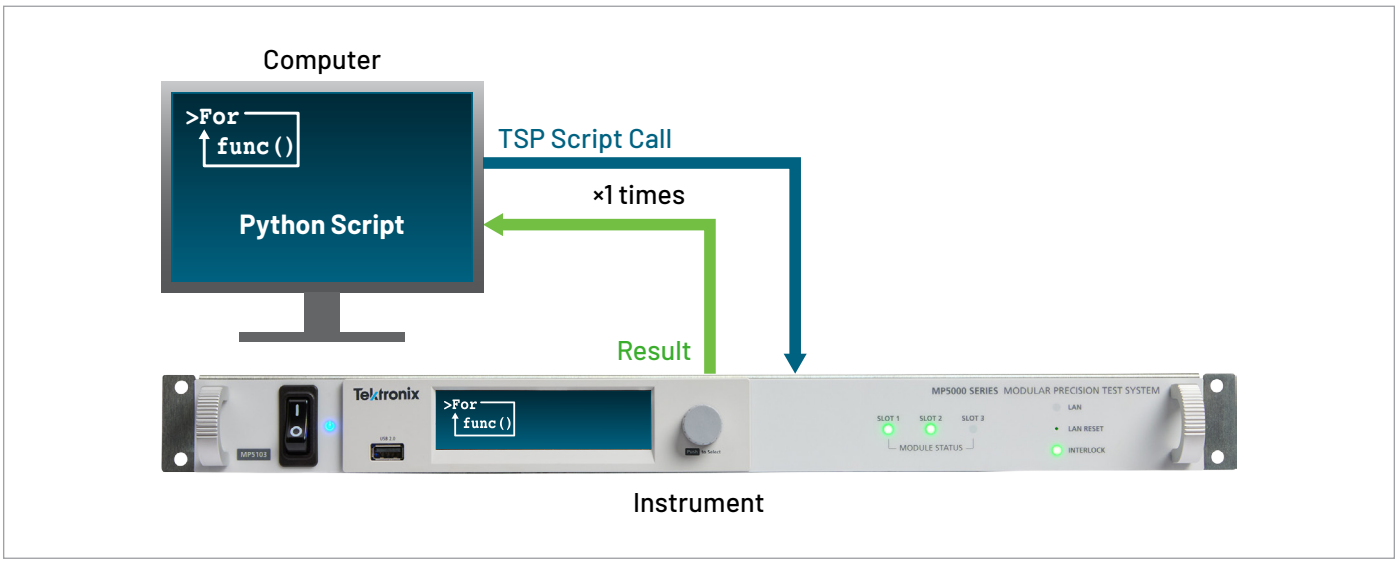


Figure 1: Diagram illustrating a Python script executing on a PC while a TSP script executes locally on the instrument.

The PC or controller runs the Python script that controls the entire system. Within this Python script are TSP commands that allow a TSP script to be loaded onto the instrument for execution. Once a TSP script has been loaded onto the instrument, any functions defined in that TSP script are accessible and can be called from the Python script as though it were a single TSP or SCPI command call. This way, a Python program can control all the overarching automation for the system such as controlling non-TSP instrumentation and data handling while the channel synchronization and timing is offloaded to the TSP instrument.

Python Interoperability

To make integrating TSP scripts into a python workflow simpler, TSP Toolkit has a feature called "Python Interoperability." This feature can automatically wrap an existing *.tsp script into a Python wrapper class with a single click. The wrapper loads the same TSP script onto the instrument and exposes its public functions and supported global variables to a Python application, allowing users to access all the timing and synchronization benefits of TSP on-instrument scripting without needing to overhaul an existing Python code base.

How it Works

A python wrapper script is generated for any *.tsp file open in TSP Toolkit by clicking the Python icon in the upper right-hand corner of the editor window or by selecting "Generate Python Wrapper from a TSP Script" from the explorer context menu. This action will open a "Select Python Output File" dialog with a default name of whatever the original TSP file was named with a *.py extension. Clicking the "Generate python wrapper file" button in the dialog will open the new *.py file beside the TSP source in VS Code.

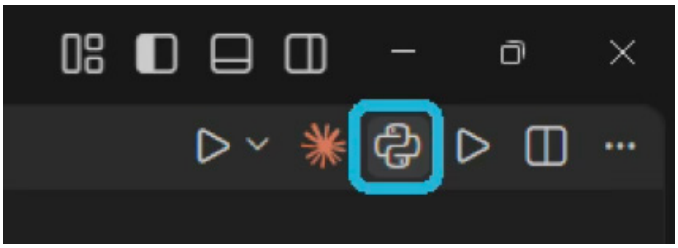


Figure 2: Generate Python Wrapper from a TSP Script button shown circled.

The generated python wrapper file includes the commands that load the TSP script onto the instrument. Top-level TSP functions are packaged into python methods with matching parameters and top-level global variables receive getter and setter methods that allow the values to be adjusted from the python layer. NOTE: The TSP script that is loaded on the instrument remains the source of the instrument's behavior; the wrapper does not translate or run the function body in Python.



Figure 3: Diagram explaining the workflow from TSP script, to wrapper, to Python script.

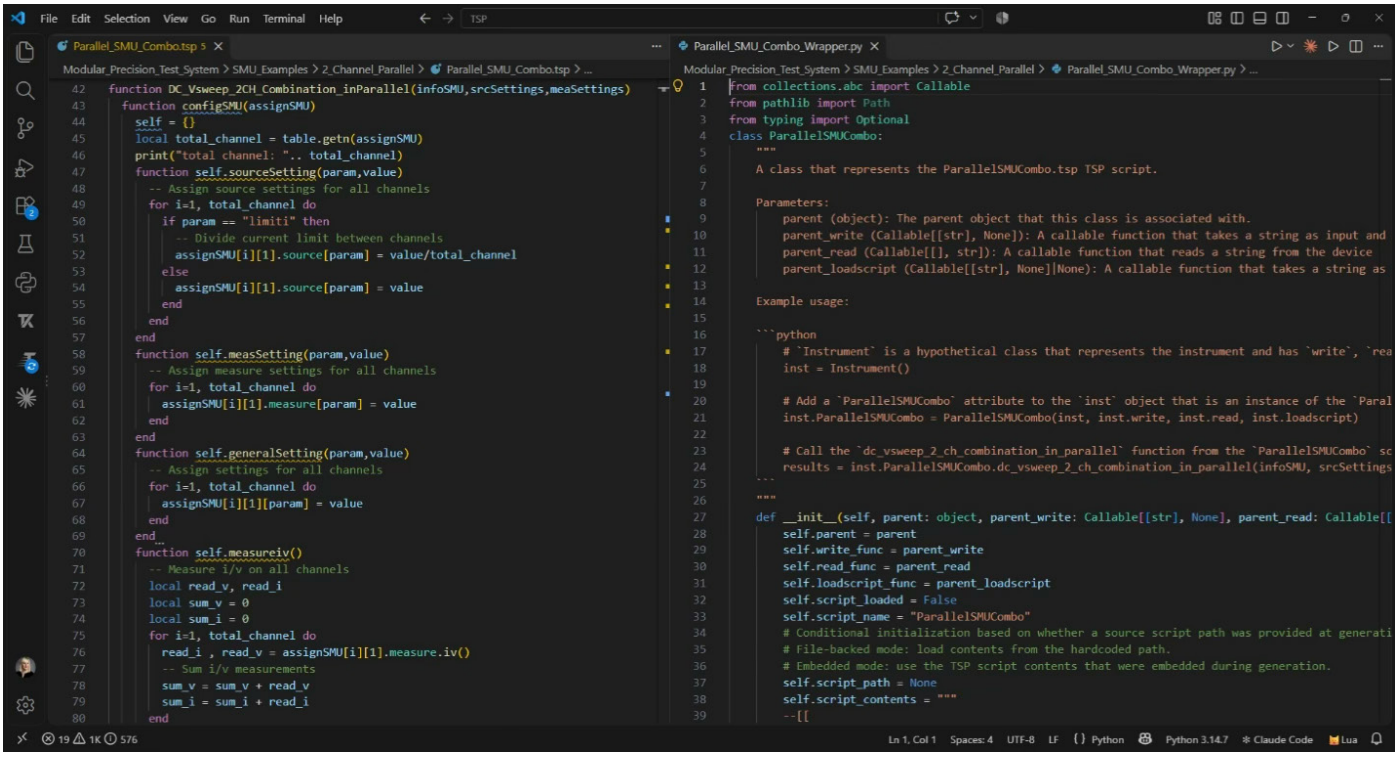


Figure 4: Original TSP script with generated python wrapper file opened side by side.

This provides a migration path for customers with established Python automation who still want the throughput, synchronization, and instrument-side execution advantages of TSP. The high-level automation can remain in Python while timing-critical execution is handled by TSP and offloaded onto the instrument.

Best Practices

As a best practice, TSP scripts that are intended to be used alongside python should be made up of defined TSP functions and variables rather than hard-coded values. A value exposed as a variable can be changed via the getter and setter methods generated by the Python wrapper, whereas a value embedded directly in the TSP script cannot be changed after the script has been loaded onto the instrument. Try to avoid leaving TSP commands "loose" and make sure the entirety of the script contents are consolidated into a function.

Example:

Below is an example of a well-constructed TSP script that is fully encapsulated in a function with defined variables and no hard coded values:

simple_sweep.tsp

```
start = 0
stop = 300e-3
points = 11
func = slot[1].smu[1].FUNC_DC_CURRENT
rangei = 1000e-3
leveli = 0
limitv = 6
rangev = 6
nplc = 1
sense = slot[1].smu[1].SENSE_2WIRE

function Simple_Current_Sweep(start, stop, points)
    reset()
    -- Source Settings
    slot[1].smu[1].source.func = func
    slot[1].smu[1].source.rangei = rangei
    slot[1].smu[1].source.leveli = leveli
    slot[1].smu[1].source.limitv = limitv

    -- Measure Settings
    slot[1].smu[1].measure.rangev = rangev
    slot[1].smu[1].measure.nplc = nplc

    --Sense Mode
    slot[1].smu[1].sense = sense

    --Calculate stop current / (number of sweep points -1)
    local delta = (stop - start) / (points - 1)
    --Turn Output On
    slot[1].smu[1].source.output = 1

    --for each sweep point, calculate the source level, take an iv measurement, and then print
the values to the terminal
    print("Current\t\t Voltage")
    for j = 1, points , 1 do
        slot[1].smu[1].source.leveli = 0 + ((j-1) * delta)
        delay(.5)
        slot[1].smu[1].measure.iv(slot[1].smu[1].defbuffer1, slot[1].smu[1].defbuffer2)
        waitcomplete()
        print(slot[1].smu[1].defbuffer1[j],slot[1].smu[1].defbuffer2[j])
```

Using TSP Toolkit to generate a python wrapper for the simple_sweep.tsp script results in this file:

pythonwrapper.py > TspError > from_dict > value

```
31  class SimpleSweep:
390     def get_start(self):
391         """Get start."""
392         return self._get_var("start")
393
394     def set_start(self, value):
395         """Set start."""
396         self._set_var("start", value)
397
398     def get_stop(self):
399         """Get stop."""
400         return self._get_var("stop")
401
402     def set_stop(self, value):
403         """Set stop."""
404         self._set_var("stop", value)
405
406     def get_points(self):
407         """Get points."""
408         return self._get_var("points")
409
410     def set_points(self, value):
411         """Set points."""
412         self._set_var("points", value)
413
414     def get_func(self):
415         """Get func."""
416         return self._get_var("func")
417
418     def set_func(self, value):
419         """Set func."""
420         self._set_var("func", value)
421
422     def get_rangei(self):
423         """Get rangei."""
424         return self._get_var("rangei")
```

Figure 5: Snippet of Python wrapper code generated by TSP Toolkit.

A python script can use the methods defined by the wrapper by importing the SimpleSweep class from the generated wrapper file.

NewPythonScript.py

```
from pythonwrapper import SimpleSweep
import pyvisa

# Configure Visa Connection
rm = pyvisa.ResourceManager()
inst = rm.open_resource('TCPIP0::192.0.2.3::5025::SOCKET')
# for using the sockets based implementation
inst.write_termination = '\n'
inst.read_termination = '\n'
inst.send_end = True
inst.timeout = 10000

inst.SimpleSweep = SimpleSweep(inst, inst.write, inst.read)

start = inst.SimpleSweep.get_start()
stop = inst.SimpleSweep.get_stop()
points = inst.SimpleSweep.get_points()

results = inst.SimpleSweep.simple_current_sweep(start, stop, points)

# Retrieve data from buffers
print(str(results))

inst.clear()
inst.close()
```

Which will print the data collected by the instrument to the Python terminal:

```
PS C:\Users\stenagli\OneDrive - Ralliant\Documents\Scripts\TSP> (Set-Exe
ant\Documents\Scripts\TSP\.venv\Scripts\Activate.ps1")
(.venv) PS C:\Users\stenagli\OneDrive - Ralliant\Documents\Scripts\TSP>
/stenagli/OneDrive - Ralliant/Documents/Scripts/TSP/NewPythonScript.py"'
Current          Voltage
-1.30122e-05    7.29681e-03
2.77853e-07     -2.81156e-04
4.45920e-07     -1.63406e-05
4.07135e-07     -4.84394e-05
5.88131e-07     1.76253e-04
7.69126e-07     4.14318e-04
8.33767e-07     2.85923e-04
9.63050e-07     5.05264e-04
1.14404e-06     6.47033e-04
9.88906e-07     4.30367e-04
1.38968e-06     7.00531e-04
```

Conclusion

With Python Interoperability in TSP Toolkit, you don't have to choose between the Python automation you rely on and the speed of on-instrument TSP scripting. One click generates a Python wrapper that loads your TSP script onto the instrument and exposes its functions and variables to your Python code. Python keeps control of the overall system, and the instrument handles the timing-critical work of synchronization and execution.

Bring the power of TSP into your Python workflow with TSP Toolkit.

Contact Information:

Australia 1 800 709 465
Austria* 00800 2255 4835
Balkans, Israel, South Africa and other ISE Countries +41 52 675 3777
Belgium* 00800 2255 4835
Brazil +55 (11) 3530-8901
Canada 1 800 833 9200
Central East Europe / Baltics +41 52 675 3777
Central Europe / Greece +41 52 675 3777
Denmark +45 80 88 1401
Finland +41 52 675 3777
France* 00800 2255 4835
Germany* 00800 2255 4835
Hong Kong 400 820 5835
India 000 800 650 1835
Indonesia 007 803 601 5249
Italy 00800 2255 4835
Japan 81 (3) 6714 3086
Luxembourg +41 52 675 3777
Malaysia 1 800 22 55835
Mexico, Central/South America and Caribbean 52 (55) 88 69 35 25
Middle East, Asia, and North Africa +41 52 675 3777
The Netherlands* 00800 2255 4835
New Zealand 0800 800 238
Norway 800 16098
People's Republic of China 400 820 5835
Philippines 1 800 1601 0077
Poland +41 52 675 3777
Portugal 80 08 12370
Republic of Korea +82 2 565 1455
Russia / CIS +7 (495) 6647564
Singapore 800 6011 473
South Africa +41 52 675 3777
Spain* 00800 2255 4835
Sweden* 00800 2255 4835
Switzerland* 00800 2255 4835
Taiwan 886 (2) 2656 6688
Thailand 1 800 011 931
United Kingdom / Ireland* 00800 2255 4835
USA 1 800 833 9200
Vietnam 12060128
* European toll-free number. If not accessible, call: +41 52 675 3777

Rev. 02.2022

Find more valuable resources at TEK.COM

Copyright © Tektronix. All rights reserved. Tektronix products are covered by U.S. and foreign patents, issued and pending. Information in this publication supersedes that in all previously published material. Specification and price change privileges reserved. TEKTRONIX and TEK are registered trademarks of Tektronix, Inc. All other trade names referenced are the service marks, trademarks or registered trademarks of their respective companies.
110926 SBG 1KW-74300-0