KEITHLEY

Application Note Series

Number 2616

Converting a Series 2400 SourceMeter® SCPI Application to a Series 2600 System SourceMeter Script Application

Introduction

For many years, instrument manufacturers worked to define a standard set of commands to control programmable test and measurement devices in instrumentation systems. Today, that standard is called the "The Standard Commands for Programmable Instrumentation" or SCPI. SCPI provides a uniform and consistent language for the control of test and measurement instruments. The same commands and responses control corresponding instrument functions in SCPI equipment, regardless of the supplier or the type of instrument.

The number of SCPI commands for an instrument can be extensive. For example, Series 2400 SourceMeter instruments have more than 390 individual SCPI commands. In many applications, only a small percentage of the commands are actually used. Setting up a Series 2400 may require specific SCPI commands be used and sent to the instrument in a very specific sequence in order for the operation to function correctly. Only when you have a good understanding of the SCPI syntax can you then take advantage of the many potential benefits the instrument offers.

Keithley's new high speed Series 2600 SourceMeter instruments use a Test Script Processor (TSP) to process and run programs called scripts to meet the most demanding throughput requirements. A script is a collection of instrument control commands and/or program statements. Program statements control script execution and provide facilities such as variables, functions, branching, and loop control. Because scripts are programs, they are written using a programming language. This language is called the Test Script Processor language. This truly simple programming interface enables the user to create powerful, high speed, multi-channel tests with significantly reduced development times.

The functionality is very simple. You download your TSP script into either volatile or nonvolatile memory and the unit controls itself, independent of the system host controller. This capability can free up the system controller to interface with other instruments in the rack more frequently, thereby increasing the overall system throughput. Furthermore, the Series 2600 has very deep memory. Program memory can hold 50,000 lines of code and data memory can store at least 100,000 readings.

The TSP language gives the user far superior flexibility with the Series 2600 as compared to the Series 2400. The Model 260x is programmed using more than 100 simplified commands that perform many of the same functions as the 390+ SCPI commands do for the Series 2400. The smaller number of Model 260x commands means the learning curve is reduced, decreasing overall development time.

This application note compares SCPI commands with the new Series 2600 scripting approach and illustrates how to convert a Series 2400 SCPI-based application to a Series 2600 test script application.

The SCPI Instrument Model

Some measurements require direct control over an instrument's hardware. To provide this control, SCPI-based instruments contain command subsystems that control particular instrument functions and settings. These commands trade interchangeability for fine control. The ability to configure instruments and make measurements with different degrees of control is a major benefit of SCPI.

The SCPI Instrument Model controls the way instrument functionality is divided among the SCPI command sub-systems. For the case of the Series 2400, the command sub-systems are broken down into the following categories:

1. **Signal-Oriented Measurement:** Commands used to acquire readings.
2. **Calculate:** Used for math expressions, limit testing, and statistics.
3. **Display:** Controls the display of the SourceMeter instruments.
4. **Format:** Selects the data format for transferring readings over the bus.
5. **Output:** Controls the output of the selected source.
6. **Route:** Controls front/rear inputs or switching.
7. **Sense:** Configures and controls the measurement functions.
8. **Source:** Configures and controls the I-Source and V-Source.
9. **Status:** Controls the status registers.
10. **System:** Contains miscellaneous commands for instrument setup.
11. **Trace:** Configures and controls data storage into the buffer.
12. **Trigger:** Configures the Trigger Model.

TSP and Scripting: A more efficient programming method

The TSP language of the Model 260x offers an equivalent model of instrument commands. The following command sets apply to the Model 260x:

1. **Beeper:** Commands used to control the built-in beeper.
2. **Bit:** Used to perform logic operations on one or two numbers.
3. **Delay:** Used to control read/write and trigger operations for the digital I/O port.
4. **Digital I/O:** Selects the data format for transferring readings over the bus.
5. **Display:** Used to control display messaging on the front panel of the Model 260x.
6. **Error Queue:** Used to read the entries in the error/event queue.
7. **Exit:** Used to terminate a script that is presently running.
8. **Format:** Used for data printed with the printnumber and printbuffer commands.
9. **GPIB:** Used to set the GPIB address.
10. **LocalNode:** Used to set the power line frequency, control (on/off) prompting, and control (hide/show) error messages on the display.
11. **Make:** Used to set and retrieve a value for an attribute.
12. **Operation Complete:** Sets the OPC bit in the status register when all overlapped commands are completed.
13. **PrintBuffer:** Used to print data and numbers.
14. **Reset:** Used to return a Model 260x to its default settings.
15. **Serial:** Used to configure the RS-232 Interface.
16. **Setup:** Used to save/recall setups and set the power-on setup.
17. **SMU:** Used to control basic source-measure operations of the SMUs.
18. **Timer:** The timer can be used to measure the time it takes to perform various operations.
19. **Trigger:** Used to control triggering.
20. **TSPLink:** Used to assign node numbers to a mainframe and initialize the TSP-Link system.
21. **UserString:** Used to store/retrieve user-defined strings in non-volatile memory.
22. **Wait Complete:** Waits for all overlapped commands to complete.

What may appear to be a larger list of command definitions compared to the Series 2400 SourceMeter is instead a reduced set of individual commands.

There are two simple methods to program and communicate with the Model 260x SourceMeter, either by executing individual TSP commands (similar to sending individual SCPI commands) or by writing test scripts. Scripts are a collection (list) of instrument control commands and/or program statements. All commands and statements in the script are executed by the Model 260x SourceMeter. Running a script at the SourceMeter is faster than running the test program from the PC. The GPIB transmission times from PC to SourceMeter can be eliminated by the use of scripts. There are two types of scripts: factory scripts and scripts created by the operator. Factory scripts are created by Keithley and are saved in nonvolatile memory of the SourceMeter Instrument.

SCPI vs. TSP Language Comparison

Let's examine the difference between using SCPI commands and those used with TSP. We will compare the two programming methods for a simple Source-Measure cycle. This example will source 10V, with a current compliance of 10mA, and measure the current. The current reading is sent to the host PC and displayed.

The Series 2400 SCPI commands to send to the instrument and an equivalent Series 2600 TSP script are shown in ***Table 1***.

A complete comparison of the Series 2600 TSP language commands versus the comparable SCPI commands for the Series 2400 is listed at the end of this Application Note. The SCPI program shown in ***Table 1*** converts easily to an equivalent TSP script. Note the similarity of the TSP commands in structure to the SCPI commands.

One of the main differences is when you take readings. In the SCPI case, you must send the READ? command to initiate the measurement. The reading is stored in a reading queue in the Series 2400. The control program must then get the reading from the instrument in order to complete the process. This is not the case with the TSP script. Note that in the script, a measurement is executed by the command **READING = smua.measure.i()**. But here, the measurement is stored in the variable **READING**. It is not necessary to return the measurement to the host controller unless it is required. **READING** could be used within the TSP

Table 1.

SCPI Commands	Comments	TSP Script Commands	Comments
`*RST`	Restore GPIB defaults.	`reset()`	Resets the Model 260x.
`:SOUR:FUNC VOLT`	Select voltage source.	`smua.source.func = smua.OUTPUT_DCVOLTS`	Select voltage source.
`:SOUR:VOLT:LEV 10`	Source output = 10V.	`smua.source.levelv = 10`	Source output = 10V.
`:SENS:CURR:PROT 10E-3`	10mA compliance.	`smua.source.limiti = 0.01`	10mA compliance.
`:SENS:FUNC "CURR"`	Current measure function.		
`:SENS:CURR:RANG 10E-3`	10mA measure range.	`smua.measure.rangei = 0.01`	10mA measure range.
`:OUTP ON`	Output on before measuring.	`smua.source.output = smua.OUTPUT_ON`	Output on before measuring.
`:READ?`	Trigger, acquire reading.	`READING = smua.measure.i()`	Acquire reading. Store in variable.
`:OUTP OFF`	Output Off	`smua.source.output = smua.OUTPUT_OFF`	Output Off.

script for other operations, such as limit testing, a math operation, or as part of an overall testing strategy. This is where the power of the TSP functions begins.

The TSP language goes well beyond just sending instrument commands. The TSP language also includes a set of capabilities that include variable and variable typing, math operators and operations, tables and arrays, creation of user functions callable from scripts, precedence, logical operators, string concatenation, conditional branching, loop control, and built-in standard string and math callable libraries. These tools, which are built into the TSP language, open up the programming potential for the Model 260x, but more importantly, simplify the application development compared to SCPI programming.

To illustrate this potential, let's convert a few SCPI Series 2400 applications into Series 2600 scripts.

Converting a SCPI Series 2400 Application to a Series 2600 Script

Let's take a look at a few simple Series 2400 SCPI programs that we want to convert to a Model 2602 script. One of the most common applications for the Series 2400 and 2600 is performing an I-V sweep. Sweeps allow you to program the instrument to step through specific voltage and current values and perform measurements at each source value.

The first program is designed to perform the following task:

1. I-V sweep: Source Voltage, Measure Current.
2. The sourcing mode will be a linear voltage sweep, with a starting voltage of 1 volt and a stopping voltage of 5 volts. Each step will be a 1 volt step.
3. A trigger count of 5 will be required to complete the five points of the sweep.
4. Turn the output on.
5. Perform a READ? to start the test and get the data.

The following ten lines of SCPI commands are required to execute the test:

```
 SOUR:FUNC VOLT
:SENSE:FUNC 'CURR:DC'
:SOUR:VOLT:START 1
:SOUR:VOLT:STOP 5
:SOUR:VOLT:STEP 1
:SOUR:VOLT:MODE SWE
:SOUR:SWEEP:SPACE LIN
:TRIG:COUNT 5
:OUPUT ON
:READ?
```

These ten lines of SCPI commands can be converted to a simple Series 2600 TSP test sequence. The following lines of TSP code can be used to perform the same function:

```
smua.source.func = smua.OUTPUT_DCVOLTS      -- Set the source function to DC Volts
smua.source.output = smua.OUTPUT_ON         -- Turns on the output
for j = 1, 5 do                             -- Create a For..Do loop to sweep from 1 to 5 volts
      smua.source.levelv = j                -- Set the output level to the integer value of 'j'
      current = smua.measure.i( )           -- Measure the current.
      print( current )                      -- Return the measurement to the PC
end                                         -- End of the For..Do loop
smua.source.output = smua.OUTPUT_OFF        -- Turns off the output
```

Using the simple **For..Do** loop, we can loop five times and use the integer loop values as the source voltage levels. Each time through the loop, the variable 'j' contains the output voltage value. The measured current is placed into a variable named "current." Lastly, the value of current is printed or sent over the IEEE bus back to a host computer.

If performing a linear sweep is a function that you need to perform often, this script can be rewritten as a TSP function that is reusable and callable at any time once loaded into the Model 260x. Here is how this function might look:

```
function linearSweep(channel,func,start,stop,step)
-- Setup source parameters here depending on the selection of Current or Voltage and the Channel.
-- To set the appropriate range, determine the maximum value between either the 'start' or 'stop
-- value. Perform an absolute value function to use the correct value in the command.

reset()

-- Verify which channel to use. Alias the command 'smuX' to channel
      if (channel=="smua") then
            channel = smua
      else
```

```
            channel = smub
      end

-- Disable all autoranging to measure both I and V
            channel.source.autorangei = channel.AUTORANGE_OFF
            channel.measure.autorangev = channel.AUTORANGE_OFF
            channel.source.autorangev = channel.AUTORANGE_OFF
            channel.measure.autorangei = channel.AUTORANGE_OFF

-- Set the appropriate source functions based on sourcing ‘amps’ or ‘volts’.
      if (func==”amps”) then
            channel.source.func = channel.OUTPUT_DCAMPS
            channel.source.rangei = math.max(math.abs(start),math.abs(stop))
      else
            channel.source.func = channel.OUTPUT_DCVOLTS
            channel.source.rangev = math.max(math.abs(start),math.abs(stop))
      end

-- Define and set specific variables

      ireadings = {}
      vreadings = {}
      sweep_index = 1
      newlevel = start
      points = math.ceil(((stop-start)/step)+1)

-- Set the output to the initial value

      if (func==”amps”) then
            channel.source.leveli = start
      else
            channel.source.levelv = start
      end

-- Turn output ON depending on channel selected

      channel.source.output = channel.OUTPUT_ON

-- Execute sweep. Measure both I and V. Index the new level by the step value in order to
-- provide the ‘channel.measureivandstep’ command with the next value.

      while (sweep_index < points+1) do
            newlevel = newlevel + step
            ireadings[sweep_index], vreadings[sweep_index] = channel.measureivandstep(newlevel)
            print(ireadings[sweep_index], vreadings[sweep_index])
            sweep_index = sweep_index + 1
      end

-- Turn output OFF depending on channel selected

      channel.source.output = channel.OUTPUT_OFF

end
```

This function allows you to pass parameters, specifically the SMU channel you wish to use, the sourcing functions (“volts” or “amps”), the start, stop, and step values. The script determines which range to use based upon the start or stop value. Note the use of the math library **math.max()** built into the TSP language. The script then checks to see which sourcing function you selected and sets the output function accordingly. The script continues by defining variable tables to hold the measurements, as well as calculating how many points will make up the sweep. The initial value of the sweep is set and the output is turned on. Next, a **While** loop is used to perform the sweep from the start value to the stop value, incrementing by the value of ‘step’. The key to high throughput in the sweep is the use of the ‘**reading = smuX.measureYandstep(sourcevalue)**’ function call. This function is provided for very fast execution of source-measure loops. The measurement will be made prior to stepping the source. Prior to using this command, and before any loop this command may be used in, the source value should be set to its initial level.

Once the sweep is completed, the output is turned off. At this point, both the current and voltage measurements are stored in their respective tables, which can be returned to the host controller, or used in another part of the test script.

Source Memory Test Sequencing

It is well recognized that communications between test equipment and the system controller can be a significant bottleneck that limits test system throughput. It is common practice to perform command and data transfers while a prober or handler is performing mechanical operations so that valuable test time is not consumed. However, as test systems get increasingly complex and more controller/instrument interaction is generally required, this gets more difficult to accomplish. Keithley's Series 2400 SourceMeter instruments addressed the issue with the source memory test sequencer, but it has its limitations. You can use all memory locations for a single test sequence or you can divide the memory locations among multiple test sequences; in either case, there are only 100 total memory locations or sequence steps available. Typically, each step in the sequence generates a reading, which can be stored in data buffers for retrieval. Depending on how you set up and initiate the instrument, the buffers can hold a maximum of from 2500 to 5000 readings. If it is necessary to retrieve the actual test values for SPC or other reasons, then depending on how many steps there are in a single test sequence, this can severely limit the number of discrete components that can be tested before interaction with the controller is required.

The second conversion example will illustrate sending the SCPI commands to perform a source memory sweep on the Series 2400. Assume a three-point sweep with the following operating modes:

Source Memory Location #1: Source voltage, measure current, 10V source value.

Source Memory Location #2: Source current, measure voltage, 100mA source value.

Source Memory Location #3: Source current, measure current, 100mA source value.

Table 2 summarizes the basic remote command sequence for performing the basic source memory sweep described above.

Table 2.

Command	Description
`*RST`	Restore GPIB default conditions.
`:SENS:FUNC:CONC OFF`	Turn off concurrent functions.
`:SOUR:FUNC MEM`	Source memory sweep mode.
`:SOUR:MEM:POIN 3`	Number memory points = 3.
`:SOUR:MEM:STAR 1`	Start at memory location 1.
`:SOUR:FUNC VOLT`	Volts source function.
`:SENS:FUNC 'CURR:DC'`	Current sense function.
`:SOUR:VOLT 10`	10V source voltage.
`:SOUR:MEM:SAVE 1`	Save in source memory location 1.
`:SOUR:FUNC CURR`	Current source function.
`:SENS:FUNC 'VOLT:DC'`	Volts sense function.
`:SOUR:CURR 100E-3`	100mA source current.
`:SOUR:MEM:SAVE 2`	Save in source memory location 2.
`:SENS:FUNC 'CURR:DC'`	Current sense function.
`:SOUR:MEM:SAVE 3`	Save in source memory location 3.
`:TRIG:COUN 3`	Trigger count = # sweep points.
`:OUTP ON`	Turn on source output.
`:READ?`	Trigger sweep, request data.

A source memory sweep represents a challenge for an equivalent Series 2600 script. The challenges lie in how you handle the different source-measure parameters and their associated values. A very quick and simple way of performing the same sweep is to write a sequential source-measure function for each value. A typical script may look like that shown in ***Table 3***:

Table 3.

TSP Script Commands	Comments
`reset()`	Resets the 260x.
`smua.source.func = smua.OUTPUT_DCVOLTS`	Set the source function to volts.
`smua.source.levelv = 10`	Source output = 10V.
`smua.source.output = smua.OUTPUT_ON`	Turn the output on.
`current1 = smua.measure.i( )`	Acquire the current measurement.
`print( current1 )`	Return the reading to the host.
`smua.source.func = smua.OUTPUT_DCAMPS`	Set the source function to current.
`smua.source.leveli = 0.1`	100mA measure range.
`current3, voltage2 = smua.measure.iv( )`	Acquire both current and voltage readings.
`print( voltage2, current3 )`	Return readings to the host.
`smua.source.output = smua.OUTPUT_OFF`	Output off.

This script performs the same function in the 18 lines of SCPI code previously shown. Because the second and third points of the Series 2400 source-memory list output the same current value of 100mA, Location #2 measures voltage and Location #3 measures the current. The Series 2600's **smuX.measure.Y** gives you the option to measure voltage, current, or voltage and current together. We can combine the equivalent of source-memory Locations 2 and 3 of the Series 2400 into a single Series 2600 command by sending the '**current3, voltage2 = smua.measure.iv( )**' command. Note that you also need to provide two variables to store the two readings.

Unfortunately, not every source-memory list sweep may result in an easy conversion as shown due to the larger number of points in the sweep to be executed. Let's modify the SCPI example by adding two more additional points to the test:

Source Memory Location #1: Source voltage, measure current, 10V source value.

Source Memory Location #2: Source current, measure voltage, 100mA source value.

Source Memory Location #3: Source current, measure current, 100mA source value.

Source Memory Location #4: Source voltage, measure current, 1V source value.

Source Memory Location #5: Source current, measure voltage, 1mA source value.

Table 4 summarizes the basic remote command sequence for performing the basic source memory sweep described in ***Table 3*** with a Series 2400.

Table 4.

Command	Description
`*RST`	Restore GPIB default conditions.
`:SENS:FUNC:CONC OFF`	Turn off concurrent functions.
`:SOUR:FUNC MEM`	Source memory sweep mode.
`:SOUR:MEM:POIN 5`	Number memory points = 5.
`:SOUR:MEM:STAR 1`	Start at memory location 1.
`:SOUR:FUNC VOLT`	Volts source function.
`:SENS:FUNC ‘CURR:DC’`	Current sense function.
`:SOUR:VOLT 10`	10V source voltage.
`:SOUR:MEM:SAVE 1`	Save in source memory location 1.
`:SOUR:FUNC CURR`	Current source function.
`:SENS:FUNC ‘VOLT:DC’`	Volts sense function.
`:SOUR:CURR 100E-3`	100mA source current.
`:SOUR:MEM:SAVE 2`	Save in source memory location 2.
`:SENS:FUNC ‘CURR:DC’`	Current sense function.
`:SOUR:MEM:SAVE 3`	Save in source memory location 3.
`:SOUR:FUNC VOLT`	Volts source function.
`:SENS:FUNC ‘CURR:DC’`	Current sense function.
`:SOUR:VOLT 1`	1V source voltage.
`:SOUR:MEM:SAVE 4`	Save in source memory location 4.
`:SOUR:FUNC CURR`	Current source function.
`:SENS:FUNC ‘VOLT:DC’`	Volts sense function.
`:SOUR:CURR 1E-3`	10mA source current.
`:SOUR:MEM:SAVE 5`	Save in source memory location 5.
`:TRIG:COUN 5`	Trigger count = # sweep points.
`:OUTP ON`	Turn on source output.
`:READ?`	Trigger sweep, request data.

The SCPI program has now increased by eight lines and the complexity of the source-memory sweep has increased. Again, we could write a Series 2600 script to perform the same sweep line by line. But there is another approach that will take advantage of the Model 260x's "Tables and Array" capabilities. The following Model 2602 in ***Table 5*** script uses tables and arrays to perform the equivalent source-memory sweep:

Table 5.

TSP Script Commands	Comments
`reset()`	Reset Model 260x.
`sourceMem = { { func = ‘v’, level = 10, meas = ‘i’`	Define points for the equivalent source-memory sweep in the Series 2400. Parameters define the source function, the output value, and the measurement function.
`},`	
`          { func = ‘i’, level = 0.1, meas = ‘v’ },`	
`          { func = ‘i’, level = 0.1, meas = ‘i’ },`	
`          { func = ‘v’, level = 1, meas = ‘i’ },`	
`          { func = ‘i’, level = 0.001, meas = ‘v’ }`	
`}`	
`readings = { }`	Define table readings to hold measurements.
`for j = 1, 5 do`	Set for..do loop from 1 to 5.
`      step = sourceMem[j]`	Alias ‘step’ to hold the parameters of sourceMem[j]
`      if (step.func == ‘v’) then`	If the parameter ‘func’ = "v", then
`          smua.source.func = smua.OUTPUT_DCVOLTS`	Set output function to DC Volts.
`          smua.source.levelv = step.level`	Set output level to the value of parameter ‘level’
`          smua.source.output = smua.OUTPUT_ON`	Turn source output on
`          if (step.meas == ‘i’) then`	If the parameter ‘meas’ = "i" then
`                readings[j] = smua.measure.i()`	measure current. Store in **readings** table.
`          else`	Else
`            readings[j] = smua.measure.v()`	measure voltage. Store in **readings** table.
`          end`	End if
`          smua.source.output = smua.OUTPUT_OFF`	turn source output off
`      else`	Else
`          smua.source.func = smua.OUTPUT_DCAMPS`	Set output function to DC Amps.
`          smua.source.leveli = step.level`	Set output level to the value of parameter ‘level’
`          smua.source.output = smua.OUTPUT_ON`	Turn source output on
`          if (step.meas == ‘v’) then`	If the parameter ‘meas’ = "v" then
`            readings[j] = smua.measure.v()`	measure voltage. Store in **readings** table.
`          else`	Else
`            readings[j] = smua.measure.i()`	measure current. Store in **readings** table.
`          end`	End if
`          smua.source.output = smua.OUTPUT_OFF`	turn source output off
`      end`	End If
`end`	End of For..Do loop

Although this script looks complicated, it is really easy to follow. The **For..Do** loops five times for the five sets of source-measure points that require execution. Also note that you are able to alias a variable to take on the parameters of **sourceMem**. This is done by setting **step = sourceMem[j]**. Conditional testing can take place to see if you need to source a voltage or a current and either measure a current or a voltage. In essence, we have created a source-memory sweep engine for the Model 2602 that would normally require many more lines of SCPI commands be sent to the Series 2400. Once the **For..Do** loop is created, modifying the **sourceMem** table for more or fewer points is all that is required, along with updating the **For..Do** loop for the number of source-measure points to be executed. The scripting does take on object oriented coding that is permitted with the Series 2600.

Conclusion

This application note touches on just a few of the potential applications that can be converted from a Series 2400 SCPI program to a Series 2600 TSP language script. ***Table 6*** maps Series 2600 TSP language commands to their equivalent Series 2400 SCPI commands. As you will see, the Series 2600 commands have been optimized for greater flexibility and operation as compared to the Series 2400 SCPI commands. In many cases, there are no equivalent SCPI commands. Using the table, you will be able to review your Series 2400 SCPI programs and determine if there is an equivalent Series 2600 command. Once you have determined if a command maps, you can then use the power of the Series 2600 TSP language to create simple to sophisticated test sequence scripts that are reusable and solve many challenging applications.

Additional Series 2600 TSP scripts can be found on Keithley's website at www.keithley.com.

Table 6. Series 2600 TSP / 2400 SCPI Command Comparison

TSP Command	Series 2400 SCPI Command(s)
BEEPER Commands	
beeper.beep(duration, frequency)	:SYSTem:BEEPer[:IMMediate <freq, time>
beeper.enable = <0/1>	:SYSTem:BEEPer:STATe <ON/OFF>
BIT Commands	
bit.bitand(value1, value2)	N/A
bit.bitor(value1, value2)	N/A
bit.bitxor(value1, value2)	N/A
bit.clear(value1, index)	N/A
bit.get(value1, index)	N/A
bit.getfield(value1, index, width)	
bit.set(value1, index)	N/A
bit.setfield(value1, index, width, fieldvalue)	N/A
bit.test(value1, index)	N/A
bit.toggle(value1, index)	N/A
DELAY Commands	
delay(seconds)	:TRIGger:DELay <n>
DIGITAL I/O Commands	
digio.readbit(bit)	N/A
digio.readport()	N/A
digio.trigger[line].assert()	N/A
digio.trigger[line].clear()	:TRIGger:CLEar
digio.trigger[line].mode = <mode>	N/A
digio.trigger[line].pulsewdith = <width>	N/A
digio.trigger[line].release()	N/A
digio.trigger[line].wait(timeout)	Assumes waiting for a trigger using TriggerLink on a line::TRIGger:SEQuence1: TCONfigure:ILINe <Nrf>
digio.writebit(bit,data)	SOURce2:TTL <bit pattern>
digio.writeport(data)	SOURce2:TTL <bit pattern>
digio.writeprotect = <mask>	N/A
DISPLAY Commands	
display.clear()	:DISPlay:WINDow1:DATA " " (20 white space characters) :DISPlay:WINDow2:DATA " " (32 white space characters)
display.getannunciators()	N/A
display.getcursor()	N/A
display.gettext([embellished, row, column start, column end])	:DISPlay:WINDow1:DATA? :DISPlay:WINDow2:DATA?
display.input()	N/A
display.inputvalue(format[, default, min, max])	N/A
display.loadmenu.add(displayname, script)	N/A
display.loadmenu.delete(displayname)	N/A
display.locallockout – display.[LOCK, UNLOCK]	Requires low level GPIB programming to send the LLO command
display.menu(name, items)	N/A
display.prompt(format, units, help[, default, min, max])	N/A
display.screen	Dependent on which row is to be displayed or not: Top Row=> :DISPlay:WINDow1:TEXT:STATe <b> Bottom Row=> :DISPlay:WINDow2:TEXT:STATe <b>
display.sendkey(keycode)	:SYSTem:KEY <n>
display.setcursor(row, column[, style])	N/A
display.settext(text)	Top Row=> :DISPlay:WINDow1:TEXT:DATA <text>;STATe ON Bottom Row=> :DISPlay:WINDow2:TEXT:DATA <text>;STATe ON
display.smuX.digits = display.[DIGITS_4_5, DIGITS_5_5, DIGITS_6_5]	:DISPlay:DIGits <n> where n =4, 5, 6, 7
display.smuX.measure.func = display.[MEASURE_DCVOLTS, MEASURE_DCAMPS, MEASURE_ OHMS, MEASURE_WATTS]	:SENSe:FUNCtion:ON <"VOLTage/CURRent/RESistance">
display.trigger.clear()	:TRIGger:CLEar
display.trigger.wait(timeout)	:ARM:SOURce MANual
ERROR QUEUE Commands	
errorqueue.clear()	:SYSTem:CLEar
errorqueue.count	:SYSTem:ERRor:COUNt?
errorqueue.next()	:SYSTem:ERRor:NEXT?
EXIT Commands	
exit()	N/A
FORMAT Commands	
format.asciiprecision = <precision value>	N/A
format.byteorder = format.[NORMAL, SWAPPED, BIGENDIAN, LITTLEENDIAN, NETWORK]	:FORMat:BORDer <NORMal/SWAPped>
format.data = format.[ASCII, SREAL, REAL, REAL32, REAL64]	:FORMat:DATA <ASCii/REAL,32/SREal>
GPIB Commands	
gpib.address = <address>	N/A
LOCALNODE Commands	
localnode.linefreq = <50/60>	:SYSTem:LFRequency <50/60>
localnode.prompts = <0/1>	N/A

Table 6. (con't.)

TSP Command	Series 2400 SCPI Command(s)
localnode.showerrors = <0/1>	N/A
MAKE Commands	
makegetter(table, attributename)	N/A
makesetter(table, attributename)	N/A
OPERATION COMPLETE Commands	
opc()	*OPC
PRINTBUFFER Commands	
printbuffer(start_index, end_index, st_1 [, st_n])	:TRACe:DATA?
printnumber(v1, [, vn])	N/A
RESET Commands	
reset()	*RST or :SYSTem:PREset
SERIAL (RS-232) Commands	
serial.baud = <baud>	N/A
serial.databits = <bits>	N/A
serial.flowcontrol = <flow>	N/A
serial.parity = <parity>	N/A
serial.read(maxchars)	N/A
serial.write(data)	N/A
SETUP Commands	
setup.poweron = <n>	:SYSTem:POSetup <RST, PRESet, SAV 0-4>
setup.recall(n)	*RCL <0,1,2,3,4>
setup.save(n)	*SAV <0,1,2,3,4>
SMU Commands	
smuX.measure.autorangeY = smuX.[AUTORANGE_ON, AUTORANGE_OFF]	DC Volts=> :SENSe:VOLTage:DC:AUTO <ON,OFF> DC Amps=> :SENSe:CURRent:DC:AUTO <ON,OFF>
smuX.measure.autozero = smuX.[AUOTZERO_OFF, AUTOZERO_ONCE, AUTOZERO_ALWAYS, AUTOZERO_AUTO]	:SYSTem:AZERO:STATe <ON,OFF>
smuX.measure.count = <count>	Dependent on Trigger Model and Trace Buffer setup: Trace Buffer=> :TRACe:POINts <Nrf> Trigger Model=> :TRIGger:COUNt <n>
smuX.measure.filter.count = <count>	:SENSe:AVERage:COUNt <n>
smuX.measure.filter.enable = smuX.[FILTER_ON, FILTER_OFF]	:SENSe:AVERage:STATe <ON/OFF>
smuX.masure.filter.type = smuX.[FILTER_MOVING_AVG, FILTER_REPEAT_AVG]	:SENSe:AVERage:TCONtrol <MOVing/REPeat>
smuX.measure.interval = <interval>	Equivalent to using a Timer trigger in the ARM Layer: :ARM:SEQuence:SOURce TIMer;TIMer <n>
smuX.measure.lowrangeY = <lowrange>	DC Volts=> :SENSe1:VOLTage:DC:RANGe:AUTO:LLIMit <n> DC Amps=> SENSe1:CURRent:DC:RANGe:AUTO:LLIMit <n>
smuX.measure.nplc = <nplc>	DC Volts=> :SENSe:VOLTage:DC:NPLCycles <n> DC Amps=> :SENSe:CURRent:DC:NPLCycles <n>
smuX.measure.overlappedX(buffer)	READ? (Buffer requires using => :TRACe:DATA?)
smuX.measure.rangeY = <rangeval>	DC Volts=> :SENSe:VOLTage:DC:RANGe <n> DC Amps=> :SENSe:CURRent:DC:RANGe <n>
smuX.measure.rel.enableY = smuX.[REL_OFF, REL_ON]	:CALCulate2:NULL:STATe <On, Off>
smuX.measure.rel.levelY = <relval>	:CALCulate2:NULL:OFFSet <n>
<reading> = smuX.measure.Y([rbuffer])	:FETCh? :TRACe:DATA?
smuX.measureYandstep(sourcevalue)	N/A
smuX.nvbuffer1	N/A
smuX.nvbuffer2	N/A
smuX.reset()	*RST
smuX.sense = <sense>	:SYSTem:RSENse <On,Off>
smuX.source.autorangeY = smuX.[AUTORANGE_ON, AUTORANGE_OFF]	DC Volts=> :SOURce1:VOLTage:RANGe:AUTO <b> DC Amps=> :SOURce1:CURRent:RANGe:AUTO <b>
smuX.source.func = smuX.[OUTPUT_DCVOLTS, OUTPUT_DCAMPS]	:SOURce1:FUNCtion:MODE <name>
smuX.source.levelY = <sourceval>	DC Volts=> :SOURce1:VOLTage:LEVel <n> DC Amps=> :SOURce1:CURRent:LEVel <n>
smuX.source.limitY = <level>	DC Volts Limit for I-Source=> :SENSe:VOLTage:DC:PROTection:LEVel <n> DC Amps Limit for V-Source=> :SENSe:CURRent:DC:PROTection:LEVel <n>
smuX.source.lowrangeY = <rangeval>	N/A
smuX.source.offmode = smuX.[OUTPUT_NORMAL, OUTPUT_HIGH_Z, OUTPUT_ZERO]	:OUTPut:SMODe <name>
smuX.source.output = smuX.[OUTPUT_OFF, OUTPUT_ON]	:OUTPut:STATe <On,Off>
smuX.source.outputenableaction = smuX.[OE_NONE, OE_OUTPUT_OFF]	N/A
smuX.source.rangeY = <rangeval>	DC Volts=> :SOURce1:VOLTage:RANGe <n> DC Amps=> :SOURce1:CURRent:RANGe <n>
TIMER Commands	
<time> = timer.measure.t()	:SYSTem:TIME?
Timer.reset()	:SYSTem:TIME:RESet
TRIGGER Commands	
trigger.clear()	:TRIGger:CLEar
trigger.wait(timeout)	:ARM:SOURce BUS
TSPLINK Commands	
tsplink.node = mynode	N/A
tsplink.reset()	N/A
state = tsplink.state	N/A
USERSTRING Commands	
userstring.add(name, value)	N/A
userstring.catalog()	N/A
userstring.delete(name)	N/A
<value> = userstring.get(name)	N/A
WAIT COMPLETE Commands	
waitcomplete()	*WAI

N/A = Not Applicable. No corresponding SCPI command.

KEITHLEY

Keithley Instruments, Inc. 28775 Aurora Road • Cleveland, Ohio 44139 • 440-248-0400 • Fax: 440-248-6168
1-888-KEITHLEY (534-8453) • www.keithley.com

No. 2616
0405