Benefits of LXI and Scripting

BY PAUL FRANKLIN AND TODD A. HAYES, KEITHLEY INSTRUMENTS

Scripting is a powerful and convenient way to provide programmability for instruments in test and measurement applications. Script-based instruments provide architectural flexibility, improved performance, and lower cost for many applications. Scripting enhances the benefits of LXI instruments, and LXI offers features that both enable and enhance scripting.

Users comfortable with conventional instruments will find it easy and straightforward to begin using script-based instruments. They can be programmed in much the same way as conventional instruments are. However, with minor adjustments to system design and programming, the flexibility, improved performance, and other benefits of scripting are easy to incorporate into system configurations.

Programmable instruments have been available for many years. Although specific capabilities vary, a programmable instrument allows the user to create and store a set of instructions in the instrument itself, where it can be executed on demand.

Early programmable instruments generally had quite limited capability and capacity, which restricted the usefulness of the programmability to relatively small and simple applications. Larger or more complex applications required the use of a separate computer that controlled the instrument via a communications interface, often GPIB.

Improvements in computing technology and programming languages and the steadily declining cost of embedded computing capacity have led to a new generation of programmable instruments. This new breed of instruments breaks through the old limits to provide greatly increased capability and flexibility. One key improvement realized in these instruments is the use of a scripting language to provide programmability.

Scripting vs. Macros or Programming Languages

Simply put, scripting is writing programs in a scripting language to coordinate a sequence of actions. Scripting goes well beyond the more conventional use of macros or recorded sequences. It uses the full power of a scripting language, which includes looping, branching, and data processing.

Although macros can be repeated in a way that provides rudimentary looping control, scripting offers a full run-time environment in which values can be stored in variables. These variables then can be used to control both looping and branching decisions.

Unlike other programming languages, script programs do not need to be precompiled before being run. Scripting environments will either interpret the program directly or compile it automatically when needed.

Beyond that, scripting languages offer the full power of a programming language. This includes creating stored procedures or functions for code reuse.

A script need not be compiled as a separate step, so scripting languages are well suited for embedded use in test and measurement equipment. Scripts can be downloaded to the instrument without the need for extra preparation steps for greater user convenience.

One key difference between a scripting language that runs on a PC and a scripting language embedded in an instrument is the environment. When running on a PC, the scripting language generally has access to a file system, virtually unlimited memory, and a graphical display as well as a keyboard and mouse. When running on an instrument, a scripting language does not necessarily have access to any of these amenities, but generally they are not needed.

Scripting in Instrumentation

Popular scripting languages include Perl®, Python®, VBScript®, and JavaScript®. The Lua scripting language is particularly well suited for embedded use because it executes faster than most other scripting languages and is implemented as a library that takes very little code space.

When adding scripting support to test and measurement instrumentation, one of the most difficult choices

```
-- cycles programs how many waveform cycles to output
cycles = 25
-- del programs the delay time between each step in seconds
del = 0
-- define the smu channels to be used to output each phase
p1smu = node[1].smua
p2smu = node[1].smub
p3smu = node[2].smua

-- Set up the sources using Sourcesetup function defined below

function Sourcesetup(smu)
        smu.reset()
        smu.source.func = smu.OUTPUT_DCVOLTS
        smu.measure.autorangev = smu.AUTORANGE_OFF
        smu.source.autorangev = smu.AUTORANGE_OFF
        smu.source.rangev = 40
        smu.source.levelv = 0
        smu.source.limiti = 1
end

Sourcesetup(p1smu)
Sourcesetup(p2smu)
Sourcesetup(p3smu)

twopi = 2 * math.pi

-- Turn on the ouptuts and output the waveform
p1smu.source.output = p1smu.OUTPUT_ON
p2smu.source.output = p2smu.OUTPUT_ON
p3smu.source.output = p3smu.OUTPUT_ON

for i = 1, cycles do
     for i =  0, twopi, twopi/36 do
            p1smu.source.levelv = math.sin(i)
            p2smu.source.levelv = math.sin(i + twopi/3)
            p3smu.source.levelv = math.sin(i + twopi/1.5)
            delay(del)
     end
end
p1smu.source.output = p1smu.OUTPUT_OFF
p2smu.source.output = p2smu.OUTPUT_OFF
p3smu.source.output = p3smu.OUTPUT_OFF
```

Figure 1. Script That Outputs a 3-Phase AC Sine-Wave Voltage Using Three SMU Channels

to make is how to present the scripting to the user. This includes answering tough questions such as "How do I integrate the instrument command set with the scripting environment?" and "How will the user load scripts into the instrument?". Keithley chose to integrate the scripting environment fully with the command set, which means that all instrument commands also are fully legal Lua statements. Essentially, each command message sent to the instrument is executed as a Lua program.

This choice makes it easy for the user to transition from controlling an instrument with single commands to using scripts because there's no need to learn a whole new command set. Commands that can be sent to the instrument over a GPIB or LXI interface are the same as the ones used within a script. This greatly simplifies the process of migrating from simple command-based control to script-based control. The user can simply send larger scripts to the instrument instead of individual commands.

There is a drawback to this choice: Instrument commands might seem a little strange to the first-time user. A few examples comparing Keithley's Model 2400 SourceMeter®, a SCPI-based unit, with the dual-channel Model 2602 System SourceMeter, a Test Script Processor (TSP)-based unit, will help demonstrate this.

The command used to make the instrument source output current on the Model 2400 is

```
:SOUR:FUNC CURR
```

The equivalent command for the Model 2602 is

```
smua.source.func = smua.DC_AMPS
```

The smua prefix designates Channel A of the Model 2602. The rest of the command is similar to the SCPI command with the exception of the equal sign. This is a Lua assignment operation that sets the value of the smua.source.func attribute to the value smua.DC_AMPS.

Queries are a little bit stranger. Because commands are valid Lua statements, the print function is used to generate output. The SCPI query to return the source function on the Model 2400 is

```
:SOUR:FUNC?
```

The equivalent command on the Model 2602 is

```
print(smua.source.func)
```

Just as a SCPI instrument supports compound commands by separating individual commands by a semicolon, the script-based instrument can accommodate compound commands by separating the commands with a statement separator. In Lua, the statement separator is a whitespace character.

Let's assume the instruments are already configured to source voltage. On the Model 2400, the following command message will set the output level and then turn the output on

```
:SOUR:VOLT 1.0; :OUTP 1
```

The equivalent command message on a Model 2602 is

```
smua.source.levelv = 1.0 smua.source.output = 1
```

Sending Script Messages

The examples illustrate that the scripting instrument can behave very much like the conventional instrument. Only the command syntax has changed slightly.

To take advantage of the full power of the scripting engine, the user simply starts sending messages that use the capabilities of the scripting language. For example, a user could ask the instrument to perform a binary search looking for the source voltage that will generate an output current of 1 mA by sending the following script:

```
step = 2.5
smua.source.levelv = step
while (step > 0.1) do
   if (smua.measure.i() > 0.001) then
      smua.source.levelv = smua.source.levelv – step
   else
      smua.source.levelv = smua.source.levelv + step
   end
```

```
    step = step / 2.0
  end
  print(smua.source.levelv)
```

A script such as this avoids the communications time required for reading each individual result and sending the commands to source new voltage levels. Although it is reasonable to question how long it takes to send the longer message, it generally will be much faster to send one longer message than to communicate several smaller messages back and forth.

However, one of the advantages of a scripting environment is that the preceding code can be packaged into a function definition and then reused, which would completely avoid sending the large message when used. For example:

```
function Search(start, target)
  step = start
  smua.source.levelv = step
  while (step >.1) do
     if (smua.measure.i() > target) then
        smua.source.levelv = smua.source.levelv – step
     else
        smua.source.levelv = smua.source.levelv + step
     end
     step = step / 2.0
  end
  print(smua.source.levelv)
end
```

This script does not make the instrument do anything right away, but it creates a stored procedure named Search that can later be invoked with this command

```
Search(2.5, 0.001)
```

Instruments can have several features that complement the scripting engine. If the scripting environment provides programmatic access to the instrument's front panel, the user can create interactive scripts that prompt the user for parameters or display results on the front panel.

The instrument also can provide on-board nonvolatile script storage so that these stored scripts can be automatically executed when the unit powers up. This allows executing a previously loaded application without any user action other than turning on the power for the instrument.

Embedded scripting provides significant benefits for test and measurement instrumentation users. Although it has some minor drawbacks associated with it, such as the unfamiliar nature of queries, most users can work around them or adapt to them readily.

Scripting languages generally manage memory automatically so the user does not need to allocate and de-allocate storage for strings or arrays. Although this is very convenient for the user, the scripting engine periodically needs to reclaim memory that no longer is being used, a process known as garbage collection.

Even though garbage collection is done automatically, it does take time, which can cause problems if it occurs during a time-critical portion of a test sequence. These problems can be prevented, but the user first must understand the impact of the garbage collector and how to avoid it in time-critical test sequences.

LXI and Scripting

The current LXI standards for instrumentation do not require that instruments be programmable or implement scripting. However, several features in the LXI specification anticipate programmable instruments and provide useful functionality that enhances scripting's capabilities on LXI-compliant instruments.

The LXI specification requires that Class A and B instruments support peer-to-peer messaging via LAN messages, and it permits Class C instruments to support it. LAN messages can be used to notify other LXI instruments of events or to trigger another instrument to perform some function.

Users must be able to specify what action is performed upon receipt of a LAN message. The most flexible way to implement this, and the way recommended by the LXI specification, is to allow the user to download executable code such as a script or program into the instrument, which then is executed upon receipt of the appropriate LAN message. This provides a great deal of flexibility because the user is not constrained to a predefined set of actions.

Furthermore, the LAN message format defined by LXI includes a small space for including arbitrary data as part of the message. It is feasible to transfer executable code, such as a short script, as part of the LAN message. This would allow one instrument to control another via LAN messages without preprogramming the response.

For example, suppose an instrument performs a measurement on a DUT. Based on the result of that measurement, it must change a stimulus applied to the DUT by another instrument. The new stimulus value is calculated based on the first measurement, so it is not known in advance. In this case, the first instrument could send a LAN message containing a short script to the second instrument to adjust the stimulus value.

Benefits of Scripting

Script-based instruments provide several benefits. Many of these are enhanced when the instrument also conforms to the LXI specification.

For many test and measurement applications, using a PC as a controller for communicating to separate instruments or using slot-based systems with integral controllers is perfectly adequate. For other situations, those approaches are overkill—and consequently overly expensive—or not quite up to the task. These applications benefit from the additional capabilities and flexibility that script-based instruments offer.

Architectural Flexibility

Small test systems with a few instruments can be built without a separate computer; one of the instruments acts as the controller and coordinates the operation of the others. Large

```
-- 3001 through 3020 (card 3, channels 1-20). The measurement on each
-- channel is triggered by an alarm.
--
-- The alarm is a simple repeating alarm based on 1588 time. It is
-- set to 15 seconds after the program starts and repeats with an
-- interval of 100 ms 20 times
--
-- The script blocks (on "scan.execute(buffer)") until all 20 measurements
-- are complete. Then timestamps of the measurements are printed relative
-- to the desired trigger time.

reset()
scan.reset()
buffer=dmm.buffer.make(200)
dmm.connect=dmm.CONNECT_ALL
dmm.autodelay=dmm.OFF
dmm.range=10
dmm.autozero=dmm.OFF
dmm.nplc=.0005
dmm.measurecount=1
dmm.configure.set('mydcvolts')
dmm.setconfig('3001:3020', 'mydcvolts')

scan.add('3001:3020')
scan.measurecount=1

scan.trigger.measure.stimulus=schedule.alarm[1].EVENT_ID

sec,ns=ptp.time()
schedule.alarm[1] .ptpseconds=sec+15
schedule.alarm[1] .fractionalseconds=0
schedule.alarm[1] .repetition=20
schedule.alarm[1] .period=0.100
schedule.alarm[1] .enable=1

print("alarm set to trigger in ", schedule.alarm[1] .ptpseconds-
sec, " seconds")

scan.execute(buffer)

for j=1,20 do
print((buffer.ptpseconds[j] +buffer.fractionalseconds[j]) -sec-15-
 (j-1)*.1)  end
```

Figure 2. Script That Sets Up a Keithley 3706 Switch to Measure DC Volts on Channels

systems can be divided into subsystems of a few instruments each, with each subsystem coordinated by a script-based instrument. This simplifies system design and can help improve performance. With LXI script-based instruments, subsystems can be physically separated, such as in assembly lines, scientific applications, or RF testing applications.

Improved Performance

Dividing large systems into subsystems coordinated by script-based instruments spreads the control and data-processing functions across multiple processors, increasing the total processing power available in the system and often improving overall speed and throughput. Furthermore, such division of labor allows for parallel testing: Instruments or subsystems do not need to sit idle while a central controller is busy with another task.

Scripts running in an instrument can operate at maximum speed because there are fewer delays due to communications with the controller while each command and piece of data are transferred. This is especially significant when the instrument is performing a repetitive test sequence.

With a separate controller, the sequence of instructions is transferred to the instrument once for every pass, even if the same sequence is run hundreds or thousands of times. Contrast that approach with a script that needs to be transferred to the instrument only once and then executed as many times as desired using a short command.

Conditional processing, such as when the results of one measurement determine the next function to be performed, offers another avenue for performance improvement. Performing the condition check locally in the script can eliminate the delays when sending the first result to the controller, waiting for the controller to process it, and then sending the next commands to the instrument.

In systems with high data rates and large data sets, communications latency, bandwidth limitations, and controller throughput can be serious bottlenecks. Script-based instruments can compress data to reduce bandwidth requirements and buffer it for background transmission when bandwidth is available. They also can filter data, for example, by only transferring data that falls outside of normal limits.

Reduced Costs

Using script-based instruments, smaller or less complex test systems can be built without a separate controller, saving the cost of the controller and that of any separate test-executive software that otherwise would be used to control the instruments. When building subsystems from script-based instruments, the same cost savings can be realized when building large test systems.

Example Scripts

Figure 1 shows how two Keithley System SourceMeters can be controlled from a single script to generate a three-phase AC waveform. In this case, the TSP-Link technology connects the two instruments and makes it easy for a script to control both instruments.

Figure 2 demonstrates how timers based on LXI Class B technology can control script operation. In the script, a Keithley Model 3706 System Switch, a Class B instrument, uses timers based on IEEE 1588 to sequence a series of measurements. The timing features in Class B are particularly useful for avoiding or minimizing system delays caused by latency or communications delays.

Developing Effective Scripts

Scripts can be developed in several ways. Keithley Instruments provides an Integrated Development Environment (IDE) called Test Script Builder (TSB) for developing scripts for any TSP-enabled instruments. TSB can be used to edit, download, and execute scripts on the instrumentation. It includes a built-in simulator for debugging a script without the need to transfer it to the instrument, which allows developing scripts even when the hardware is unavailable.

Some LXI instruments have a Telnet port that can be used for remote control. For these instruments, using a text editor offers a quick and simple way to write and debug scripts. From the Telnet application, the user can paste script text or download script files directly to the instrument.

Some users prefer to embed scripts directly into the test-executive application. They develop and debug scripts and the test-executive application at the same time.

LXI's web connectivity has allowed Keithley to use a script-development tool called TSB Embedded in its Series 3700 Switch/DMM products. Users can access this tool via a Web page served by the instrument itself, using a Web browser to develop and manage their scripts without installing any software on the PC.

A function-based or object-oriented approach is advisable when developing scripts for a product with embedded script processing. Functions should be used wherever possible. This not only is good practice for maximizing code reuse, but it also reduces the amount of code stored in the run-time environment of the scripting engine and leaves more memory for additional scripts and data storage.

Embedded scripting can reduce the communications time between the host PC and the instrumentation. A function-based approach maximizes this advantage because the host PC need send only a short message to invoke a stored procedure. If more lengthy messages are often sent to the instrument, the communications reduction advantage is diminished.

Regardless of how a script is developed, scripting brings some new concerns to test management. Although it is useful in some situations to store scripts in nonvolatile memory on the instrumentation, it is not always best to do so. When a test executive expects that a particular version of a script will be on the instrumentation, it is better to load the scripts on the instrumentation when the test executive starts. That way there is complete control over which version of script code the test executive is using.

Script-Based Instruments

Script-based instruments may be used in conventional test systems with a separate controller. The details vary depending on exactly how the manufacturer chooses to implement scripting.

Those accustomed to using instrument drivers to interface the software and the instrument will find that they can continue to use an instrument driver and treat a script-based instrument much like a conventional instrument. However, doing so would eliminate many of the advantages scripting offers. Fortunately, there are methods that allow instrument-driver writers and users to benefit from the extra flexibility and capability script-based instruments offer.

When developing an instrument driver for a script-based instrument, you can choose from three general approaches:

Conventional

The driver is similar to one for a conventional instrument. No use is made of scripting capability. The only adjustment is to accommodate the differing syntax.

Extended

The conventional-style driver is enhanced with functions for transferring scripts to the instrument and perhaps managing return data. This provides a way for users to exploit scripting capability, but the driver itself does not do so.

Enhanced

An instrument driver for a script-based instrument can take advantage of scripting in many ways. For example, such a driver could download scripts that perform many of the functions normally handled by the driver to the instrument itself. Then, calls made to the driver are sent to the instrument as short simple commands rather than as longer series of typical instrument commands.

As always, there are trade-offs with such a design. But script-based instruments provide additional flexibility for optimizing system and software design to achieve the best performance possible for a given application.

The same three approaches apply to writing software that controls a script-based instrument directly without using an instrument driver.

About the Authors

Paul Franklin is the manager of Keithley Labs, the technology development group within Keithley Instruments. He chaired the LXI Consortium's Technical Committee from 2005-2007. Before joining Keithley Instruments in 2000, he gained more than 20 years of measurement and control industry experience as an engineer and a manager with electronic controls and industrial automation firms. Mr. Franklin earned B.S.E.E. and M.S.E. degrees at Case Western Reserve University and is a member of IEEE, the IEEE-Computer Society, the IEEE-Instrumentation and Measurement Society, and the Association for Computing Machinery. 440-542-8097, e-mail: pfranklin@keithley.com

Todd A. Hayes is a senior staff firmware engineer at Keithley Instruments. He has more than 15 years of experience in embedded firmware design and was the lead firmware architect on development of the company's TSP. Mr. Hayes received B.S.E.E. and M.S.C.S degrees from the University of Akron. 440-248-0400, e-mail: hayes_todd@keithley.com

Keithley Instruments, 28775 Aurora Rd., Cleveland, OH 44139