

Increasing Production Throughput with Data Acquisition Systems

APPLICATION NOTE



Introduction

While product design is investigated and worked through to its final state, it is important to be mindful of how units will be tested both prior to and after the final incorporation into their enclosure. The design team often provides test pads at numerous strategic locations on bare printed circuit boards (PCB), allowing access to points of interest that can be measured at the test time to ensure the operation and quality of the product. With its ability to select from up to 80 channels using any of its fifteen measurement functions, the DAQ6510 Data Acquisition and Multimeter System is a prime candidate for integration into a test system to help meet multi-point test needs.

However, system integrators, and the engineering team that designs and supports them, need to also focus their attention on throughput and need to find ways to optimize their systems to allow them to effectively and repeatedly check for performance and quality. The most obvious method for increasing throughput is to increase bandwidth by creating multiple test systems. However, this is typically not a cost-effective solution, especially when the product is new and has not generated a return on investment. Therefore, engineers must turn their attention to the selected set of test instrumentation for methods that allow them to increase the speed of their testing.

An engineer may wish to compare the speed of similar instruments, such as a data acquisition system, to determine if a newer product will promote greater efficiency. Most modern data acquisition systems employ switching across numerous channels to capture measurements. Therefore, switching hardware options should be evaluated to see what speed advantages one model might hold over another.

Engineers will also need to consider how their software will control the test equipment. While accepting default measurement function settings removes some degree of detail, the programmer requires a setup that can quickly begin collecting measurements. These default settings often include setup latency that can unnecessarily extend test time as well. Additionally, because the product tends to go through rigorous evaluation and auditing through its design lifecycle, measurement tolerances can be loosened during final test scenarios. Gaining an intimate knowledge of overall instrument operation can reveal ways to speed up testing and help eliminate unwanted delay.

As data acquisition systems and their communication interfaces have evolved. While standard interfaces, such as GPIB and RS-232, promote confidence and reliability to the engineer, they may be sacrificing some speed advantages provided by newer interfaces, such as Ethernet and USB. The user should evaluate the available options to ensure



that the speed at which a controlling computer manipulates the test equipment and extracts readings does not act as a bottleneck to their testing environment.

Finally, an engineer should look for additional methods to increase test speed. One such consideration would be to have instrumentation that allows for on-the-box processing to help cut down on the amount of overall system communication necessary to execute testing. The DAQ6510 uses an embedded Test Script Processing (TSP) engine that promotes the execution of loaded program statements helping to offset the processing at the computer.

The following examples progressively provide scenarios that highlight the suggestions above, starting with a simple data acquisition scan executed on the DAQ6510 and, compared to its predecessor, the Model 2701. Relay cards and the types of relays (electromechanical, reed, and solid state) are briefly discussed and tests executed to provide results show the performance difference between each. Instrument configuration is then investigated, showing how deviating from using the easier default settings can remove delays. Test cases are also provided to show throughput changes with respect to the different communication options available with

the DAQ6510. Finally, a test scenario using TSP processing is shown as an example of how a user might utilize it in their test environment.

Modernized Instrumentation – a Basic DC Voltage Scan Comparison

To observe the possible throughput improvements, we suggest using more modern equipment, which are equipped with faster processing devices such as microprocessors, FPGAs, and memory chips. We will compare the test speed of two instruments by implementing a simple DC voltage scan on both the newer-generation, DAQ6510, and on the older-generation, Model 2701. Full details of the test scenario are as follows:

- DC voltage measurements
- 10-channel scan (channels 101-110) on a Model 7700 Multiplexer card
- Scan count of 100 for a total of 1000 readings
- Extracted readings from the instrument to the controlling computer, 50 at a time
- All other default settings for each instrument (auto delay, auto zero, aperture time, etc.) are the same and need not be manipulated



The following table provides the SCPI commands used by each instrument to execute the test in question along with pseudocode in certain spots to help highlight the control of data extraction and timing.

DAQ6510 SCPI Commands	Model 2701 SCPI Commands	Comments
string dudCntr = "" string rcvBuffer int startIndex = 1 int endIndex = 50 int totalRdgs = 0	string dudCntr = "" string rcvBuffer int accumVals = 0;	Create a variable to hold the counter for the loop Create a variable to hold the receive buffer Create a variable to hold the accumulated values count Create a variable to hold the total readings count Create variables to hold the indexes for the loop
*RST	*RST	Put the instrument in a known state
FORM:DATA:ASCII	FORM:DATA ASCII	Format data as an ASCII string
	FORM:ELEM READ	Return the reading with units only
ROUT:SCAN:COUN:SCAN 100	SAMP:COUN 1000	Set Scan(6510)/Sample(2701) Count
FUNC 'VOLT:DC', (@101:110)	FUNC 'VOLT:DC', (@101:110)	Set function to DCV
TRAC:CLE	TRAC:CLE	Clear the reading buffer
TRAC:POIN 1000, "debuffer1"		Set the buffer size
ROUT:SCAN:CRE (@101:110)	ROUT:SCAN (@101:110)	Set the scan list
	ROUT:SCAN:LSEL INT	Enable the scan
*OPC?	*OPC?	Check that previous commands are complete before initiating the scan
INIT	INIT	Initiate the scan
	Delay 50	Delay 50 ms to allow readings to accumulate
while totalRdgs != 1000	while accumVals < 1000	Begin while loop
dudCntr = TRAC:ACT:END?	dudCntr = TRAC:NEXT?	Check for the last buffer reading
If dudCntr – totalRdgs >= 50	If dudCntr – accumVals > 50 or dudCntr = 0 DudCntr = ""	Check that there are enough readings to print to the console
rcvBuffer = TRAC:DATA? startIndex, endIndex, "debuffer1", READ	rcvBuffer = TRAC:DATA:SEL? accumVals, 50	Return readings in chunks of 50
Print rcvBuffer totalRdgs = totalRdgs + 50 startIndex = startIndex + 50 endIndex = endIndex + 50 End if End while	Print rcvBuffer accumVals = accumVals + 50 End if End while	Print the contents of the receive buffer to the console window Increment indexes End loop
*OPC?	*OPC?	Hold for operation to complete
	ROUT:SCAN:LSEL NONE	Disable the scan

The following table highlights the overall command execution time along with the throughput results in units of readings per second to provide an understanding of how many measurements the user can expect returned to the test program using each instrument.

Instrument	Run Time @ Data Acquisition Rate (readings/second)
DAQ6510	34.12 @29.30
Model 2701	39.75 @25.15

With the DAQ6510 showing greater than six seconds of speed improvement, we can envision how the new model can be used on a broader scale and help to reduce overall test time across multiple test instances.

Speeding Scan Time Through Relay Type Selection

Another factor you will want to consider with your data acquisition system is switching hardware options available to route your signals. The relay type used on your multiplexer/switch will have typical specifications for closed and open times (the time it takes to connect and break signal paths). This directly impacts the speed of your test. The quicker the relays can be closed or opened, the faster the test process can operate.

There are three main types of relays you will want to be aware of, and the following examples highlight the actuation times as well as which multiplexer module they are used with.

- Electromechanical Relay (Model 7700 Multiplexer Module): <3 ms actuation time
- Dry Reed Relay (Model 7703 Multiplexer Module): <1 ms actuation time
- Solid State Relay (Model 7710 Multiplexer Module): <500 μ s actuation time

Each of the multiplexing modules indicated above are compatible with the DAQ6510 and the Model 2701. Therefore, we can run the previous code examples and only vary the card model to highlight how the relay type affects the overall test times and throughput. The table below shows the results of the testing.

Scan Card Model		Run Time @ Data Acquisition Rate (readings/second)		
		7700 (EM)	7703 (Reed)	7710 (SS)
Instrument	DAQ6510	34.12 @29.30	24.48 @40.84	21.37 @46.78
	Model 2701	39.75 @25.15	30.57 @32.71	29.82 @33.53

Not only does the DAQ6510 show better throughput performance across each multiplexing module, in comparison to the Model 2701, but we can reduce the test time by approximately nine seconds when the dry reed relay unit is used and by approximately twelve seconds when the solid state relay unit is used.

While these results help to reveal the benefits that you can reap by selecting a different model of multiplexing card, you need to be aware that the cards differ in other ways (such as termination type, number of channels, etc.). We encourage you to review the datasheets for each of the switching modules indicated to determine which is best for your setup and application.

For more information about relay types, timing, and other switching-related details, download your copy of the “Data Acquisition Primer: An Introduction to Multi-Channel Measurement Systems” from www.tek.com.

Instrument Configuration Considerations for Increased Speed

Many test instruments tend to ship with their default settings configured to obtain the best measurements at power-up. This is quite convenient for instruments used on a bench-top and for manual probing and making measurements. Since your hands are occupied with holding test leads, you can rely on the equipment to automate most of the function setup and focus on the outcome.

While default configurations are excellent for hands-on usage, certain settings can hinder throughput. The following list helps to highlight specific settings and provide insight into how they can slow you down.

- Autorange – When applied, this feature automatically determined the best range for the stimulus being measured.
- Auto Delay – With auto delay selected, the instrument automatically selects a delay period that will provide sufficient settling for function changes, autorange changes, and multi-phase measurements.

- Auto Zero – To help maintain stability and accuracy over time and changes in temperature, the instrument periodically measures internal voltages corresponding to offsets (zero) and amplifier gains.
- Line Cycle Synchronization – Synchronizing A/D conversions with the power line frequency increases common mode and normal mode noise rejection. When line cycle synchronization is enabled, the measurement is initiated at the first positive-going zero crossing of the power line cycle after the trigger (which may take several milliseconds).
- Filtering and Math – A digital filter can be used to stabilize noisy measurements. The displayed, stored, or transmitted reading is an average of many reading conversions. Math functions can be applied to readings to provide a scaled, percentage-based, or reciprocal result.
- Digits Displayed – The more digits of resolution that are made visible on the display, more processing power is shared by the measurement engine.
- Trigger Delay – This is the applied delay between trigger signal detection and when the instrument will proceed to the next step in the measurement process.
- Scan Interval – This is the period between the start of scans and may be set to a duration longer than it takes to perform a single scan.
- Integration Rate – Defines the amount of time the A/D circuit measures a signal. The integration rate is expressed in terms of the number of power line cycles (NPLC). Integral PLCs eliminate line cycle noise by measuring over a complete power line period and effectively averaging out the power line noise. Sub-line cycle integration rates increase reading speed at the expense of higher noise.

When you are implementing your test scenario in production, many of these features can be disabled or have their functionality scaled back. A production-functional test determines a pass or fail condition based on expected levels with certain limits. Therefore, you can apply a fixed range and reduce the measurement delays. Additionally, you may not want to consider the auto zero feature, due to its additional measurements, or line cycle synchronization. Filtering may be unnecessary if your measurements can sustain an acceptable noise level, and special math functions, in most cases, are unnecessary. Because production-functional testing implies that the instrumentation is integrated into a test system, the display is typically not needed during normal operation, allowing for the number of digits displayed to be reduced or to have the display turned off.

Since it works best to show these recommendations in action, the original test is modified such that:

- The voltage range is set at 10 V.
- Auto delay and/or channel delay are disabled and/or set to 0 s.
- Auto zero is disabled.
- Disable the front panel LCD screen prior to starting the scan, then re-enable the front panel screen after the scan is complete but prior to stopping the timer since the SCPI command is still part of the controlling code.
- Line sync is disabled.
- Limits testing is turned off.
- Channel stats/math is turned off.
- The trigger delay and/or scan interval is set to 0 s.
- Start with the integration rate (aperture time) set to 1 PLC.
 - Reduce to 0.02 PLC.
 - Reduce further to 0.002 PLC, the lowest possible setting for the legacy instrument.

The following table provides the SCPI commands used by each instrument to execute the test in question along with pseudocode in certain spots to help highlight the control of data extraction and timing.

DAQ6510 SCPI Commands	Model 2701 SCPI Commands	Comments
string dudCntr = "" string rcvBuffer int startIndex = 1 int endIndex = 50 int totalRdgs = 0	string dudCntr = "" string rcvBuffer int accumVals = 0;	Create a variable to hold the counter for the loop Create a variable to hold the receive buffer Create a variable to hold the accumulated values count Create a variable to hold the total readings count Create variables to hold the indexes for the loop
*RST	*RST	Put the instrument in a known state
FORM:DATA:ASCII	FORM:DATA ASCII	Format data as an ASCII string
	FORM:ELEM READ	Return the reading with units only
ROUT:SCAN:COUN:SCAN 100	SAMP:COUN 1000	Set Scan(6510)/Sample(2701) Count
FUNC 'VOLT:DC', (@101:110)	FUNC 'VOLT:DC', (@101:110)	Set function to DCV
VOLT:RANG 10, (@101:110)	VOLT:RANG 10, (@101:110)	Set fixed range at 10 V
VOLT:NPLC 1, (@101:110)	VOLT:NPLC 1, (@101:110)	Set integration rate to 1 PLC
DISP:LIGH:STAT OFF	DISP:ENAB OFF	Turn off the display
VOLT:DEL:AUTO OFF, (@101:110)	TRIG:DEL:AUTO OFF	Turn off auto delay
ROUT:CHAN:DEL 0, (@101:110)	TRIG:DEL 0	Set channel/trigger delay to 0 s
ROUT:SCAN:INT 0		Set scan interval to 0 s
	VOLT:AVER:STAT OFF, (@101:110)	Disable background statistics
CALC:VOLT:MATH:STAT OFF		Turn off math operations
CALC2:VOLT:LIM1:STAT OFF, (@101:110) CALC2:VOLT:LIM2:STAT OFF, (@101:110)	CALC3:OUTP OFF	Turn off limit tests
VOLT:AZER:STAT OFF,	SYST:AZER:STAT OFF	Turn off auto zero
VOLT:LINE:SYNC OFF,	SYST:LSYN OFF	Turn off line sync
	CALC3:LIM1:STAT OFF, (@101:110)	Turn off limits
TRAC:CLE	TRAC:CLE	Clear the reading buffer
TRAC:POIN 1000, "debuffer1"		Set the buffer size
ROUT:SCAN:CRE (@101:110)	ROUT:SCAN (@101:110)	Set the scan list
	ROUT:SCAN:LSEL INT	Enable the scan
*OPC?	*OPC?	Check that previous commands are complete before initiating the scan
INIT	INIT	Initiate the scan
	Delay 50	Delay 50 ms to allow readings to accumulate
while totalRdgs != 1000	while accumVals < 1000	Begin while loop
dudCntr = TRAC:ACT:END?	dudCntr = TRAC:NEXT?	Check for the last buffer reading
If dudCntr – totalRdgs >= 50	If dudCntr – accumVals > 50 or dudCntr = 0 DudCntr = ""	Check that there are enough readings to print to the console
rcvBuffer = TRAC:DATA? startIndex, endIndex, "debuffer1", READ	rcvBuffer = TRAC:DATA:SEL? accumVals, 50	Return readings in chunks of 50
Print rcvBuffer totalRdgs = totalRdgs + 50 startIndex = startIndex + 50 endIndex = endIndex + 50 End if End while	Print rcvBuffer accumVals = accumVals + 50 End if End while	Print the contents of the receive buffer to the console window Increment indexes End loop
*OPC?	*OPC?	Hold for operation to complete
DISP:LIGH:STAT ON	DISP:ENAB ON	Turn on the display
	ROUT:SCAN:LSEL NONE	Disable the scan

The table below highlights the overall command execution time along with the throughput results in units of readings per second to help determine how many measurements the user can expect returned to the test program using each instrument. In comparison to the previous test cases, you can immediately see how applying direct control of measurement functions and features yields a 300% speed improvement for just the combination of the DAQ6510 + Model 7700.

		Run Time @ Data Acquisition Rate (readings/second)								
Scan Card Model		7700 (EM)	7703 (Reed)	7710 (SS)	7700 (EM)	7703 (Reed)	7710 (SS)	7700 (EM)	7703 (Reed)	7710 (SS)
Integration Rate		1 PLC			0.02 PLC			0.002 PLC		
Instrument	DAQ6510	27.67 @36.14	18.83 @53.08	17.80 @56.16	11.33 @88.25	2.51 @398.41	1.46 @680.74	11.28 @88.61	2.42 @412.88	1.37 @727.27
	Model 2701	29.04 @34.42	20.14 @49.64	19.49 @51.30	12.64 @79.10	3.72 @268.46	3.10 @322.37	12.36 @80.85	3.45 @289.52	3.09 @323.62

In the previous example, we limited the integration time to 0.002 PLC because it is the lowest that the Model 2701 can achieve. The DAQ6510 can use a 0.0005 PLC integration time. However, this does not mean that the runtime will be cut by more than half or the number of readings per channel are more than doubled. As is the case for all equipment, at some point, the hardware itself will become a limiting factor and will offset any additional throughput gains that might be achieved. In this case, unless the run time provides millisecond or microsecond resolution, you may not immediately witness benefits until the amount of measurements acquired increases.

The following table is provided to show that, while we can run with the lowest possible integration time on the DAQ6510, the system runs to its limits and with no significant speed performance gains unless the instrument is used as a single-channel DMM.

		Run Time @ Data Acquisition Rate (readings/second)		
Scan Card Model		7700 (EM)	7703 (Reed)	7710 (SS)
Integration Rate		0.0005 PLC		
Instrument	DAQ6510	11.28 @88.61	2.42 @412.03	1.36 @733.14

For a comprehensive table showing the speed advantages of the above examples as well as a comparison against a primary competitor's solution, see Appendix A.

Communications Interface Selection Considerations

Because a test system is typically composed of more instruments than just a data acquisition instrument and a controlling computer, you will need to consider what communications tools will best suit your setup. Because VISA simplifies the software, providing application programming interface (API) plugins that make read, write, and query operations effortless and appear nearly identical, you can put more thought into the hardware interface.

While GPIB has been an industry standard for test instrumentation control for many years, a GPIB controller is required by your computer, adding some cost. Serial, also known as RS-232, is reliable and has been around for even longer and is used for communications beyond a test environment. Some control elements for the serial option have additional settings, but, once these are addressed, the communication is not much different from using GPIB. The cost of a serial interface is far less than GPIB, which is appealing. However, the speed advantage of GPIB over serial (480 Mb/s for high-speed controllers compared to 115,200 baud, respectively) may justify the additional cost of a GPIB interface when throughput is a concern.

Ethernet, or LAN, connectivity is appealing since LAN is the standard interface used on most computers and allows instrumentation to be accessed from virtually any location. This promotes the integration of additional software tools

directly on the instrument, such as embedded web interfaces and virtual front panels that can be used for general control or debugging. Ethernet has great speed potential and can get the user more data in shorter amounts of time. Whether working at 10 Mb/s, 100 Mb/s, or 1 Gb/s, keep in mind that computer networking rules and limitations still apply. The more devices connected to and utilizing a network the greater the chance for a reduction in bandwidth and speed. While hooking up to your corporate network may not yield the best results in data transfer speed, creating a private local network, by using an extra Ethernet port or adapter, can help to remove bottleneck conditions.

With USB 2.0 and 3.0 offering faster data rates, up to 480 Mb/s and 5 Gb/s, it is appealing to engineers who need to quickly transfer data large amounts of data with the lowest latency possible. USB is also a standard interface on computers and is becoming a more common interface on test instrumentation. One criticism, however, is that USB can be susceptible to noise in a production test environment. You can help to limit the effect of noise with proper shielding.

The DAQ6510 supports all four communications interfaces listed above, with USB 2.0 and Ethernet delivered as a standard part of the product. Executing the optimized scanning example above with each interface yields the results shown in the table below.

Some things to note:

- USB is 2.0 as indicated above
- GPIB-HS indicates that the “high speed” GPIB-to-USB device was used to show fastest possible speed with a GPIB connection, reaching up to 480Mb/s (same as USB 2.0)
- RS-232 baud rate set to 115,200

		Run Time @ Data Acquisition Rate (readings/second)		
Scan Card Model		7700 (EM)	7703 (Reed)	7710 (SS)
Integration Rate		0.002 PLC		
Interface	USB	10.96 @91.19	2.22 @450.25	1.05 @950.57
	Ethernet	11.28 @88.61	2.42@412.88	1.37 @727.27
	GPIB-HS	10.99 @90.93	2.25 @443.66	1.09 @917.43
	RS-232	11.07 @90.32	2.33 @429.18	1.94 @514.40

USB shows the strongest performance here providing about 4% greater throughput than the second-best performer.

All previous examples have offered insight on potential throughput enhancements with respect to scanning across multiple points. We should also acknowledge the common case of data logging on a single point and measurement function, where you may want to acquire as much data as quickly as possible and stream it to your computer for further analysis. The DAQ6510 includes a 1 MS/s digitizer that can sample through its front terminals and any of the configured, switched channels.

Regarding streaming readings from your test instrument, the DAQ6510, as well as the DMM6500 and DMM7510, can achieve streaming rates up to 100 k readings/second using the USB interface. For more on this topic, including examples, visit the Keithley product pages on the tek.com website.

Reduce Communication Overhead and Computer Processing

The previous section noted different instrument control protocol possibilities, and the speed throughput expectations when sending and receiving with the same series of SCPI commands. In the examples above, we presented the SCPI commands in their abbreviated form – the more verbose form aids in user understanding as the commands hold more characters to help better define the action the specific command performs. The advantage of the abbreviated form is that the user can achieve the same outcome sending fewer characters. Every ASCII character sent and received takes time to propagate from sender to receiver. Therefore, fewer characters translates to less communication time.

Additionally, a host computer manages all the computer-to-instrument communication and test program execution. While newer computer architectures better support multi-threaded applications using multi-core processors, it would be advantageous to discover methods to reduce the processing burden on the computer to allow it to use its resources to perform other activities.

Let us consider how Keithley Instruments addresses this issue with Test Script Processor (TSP®) enabled instruments. TSP instruments operate like conventional instruments by responding to a sequence of commands sent by the controller. You can send individual commands to the TSP-enabled instrument the same way you would when using any other instrument. Unlike conventional instruments, TSP-enabled instruments can execute automated test sequences independently, without an external controller. You can load a series of TSP commands into the instrument using a remote computer or the front-panel port with a USB flash drive. You can store these commands as a script that can be run later by sending a single command message to the instrument, or load and run from the front panel. The primary tradeoffs are programming complexity versus throughput. Relatively simple programming techniques will allow noticeable throughput

improvements. Slightly more complex programming (using TSP-based instruments and programming “best practices”) can produce significantly higher throughput gains.

As an example of how you might use TSP to your advantage, let us use the code samples in the appendix to provide some insight, with details as follows:

- Appendix B: TSP function file to be loaded on to the DAQ6510
- Appendix C: Python Example Using TSP Command Calls to Execute Scans
- Appendix D: Python Example Using SCPI Command Calls to Execute Scans

Note that the entire contents of the Lua script file (named “scanFunctions.lua”) found in Appendix B is loaded directly on to the target instrument by the Python code of Appendix C:

```
def load_functions(s):
    # This function opens the scanFunctions.lua file in the same directory
    # as the Python script and transfers its contents to the DAQ6510
    # internal memory. All the functions defined in the file are callable
    # by the controlling program.
    func_file = open(functions_path, "r")
    contents = func_file.read()
    func_file.close()
    s.send("if loadfuncs ~= nil then "
          "script.delete('loadfuncs') "
          "end\n".encode())
    s.send("loadscript loadfuncs\n{0}\nendscript\n"
          .format(contents)
          .encode())
    s.send("loadfuncs()\n".encode())
    print(s.recv(100).decode())
```

We consider this part of the initialization phase, so when your test program launches for the first time the file is loaded only once. All of the functions contained within that file are local to the instrument and are now callable by their simpler function name. Therefore, to configure the test in question – one that scans temperature on four channels, DC voltage on four channels, two-wire resistance on three channels, four-wire resistance on three channels, and DC current on two channels – we send the following either in SCPI or in TSP:

SCPI Setup Calls	TSP Setup Call
cnfgSpeedScan()	cnfgSpeedScan()
cnfgFunc(s, fnc, chLst)	
cnfgSetRng(s, fnc, rng, chLst)	
cnfgSetNplc(s, fnc, nplc, chLst)	
enblAZero(s, fnc, 0, chLst)	
enblADelay(s, fnc, 0, chLst)	
enblLimit(s, fnc, 0, 0, chLst)	
enblLineSync(s, fnc, 0, chLst)	
setScanList(s, chLst)	
setScanCount(s, cnt)	
setScn2Scn(s, s2sInt)	
sendOpc()	

All of the indented functions in the column of *SCPI Setup Calls* will issue at least one command string of several ASCII characters each. We will say the average command string length is twenty characters, meaning that to perform one of the speed scan setups will require the computer to issue 220 characters. To perform all five will require **1100**, and this is just for the setups.

In comparison, one TSP command (with passed arguments) may require forty characters, but issued five times may only require **200** characters. Now consider sending either the SCPI or TSP setup commands to test 100 devices, and you can see how the character transmission count for the SCPI implementation far exceeds that of TSP.

Also, consider the scenario in the examples where the instrument is polled for scan completion: five times for either implementation. While the TSP implementation (`waitScanCmplt()`) uses 14 characters and the SCPI implementation (`TRIG:STAT?`) uses 10 characters, only the former can have its command call shortened to use fewer characters.

The following table shows the performance comparison for a DCV scan looped ten times using both the SCPI and TSP code instances found in the appendices. As in the above examples, we optimize each setup for the best speed achievable by fixing measure ranges, turning off auto delay and auto zero, reducing the integration time to 0.001 PLC, disabling line sync and limits testing, and use the Model 7710 solid-state multiplexer module.

Command Set Performance Compare		
Command Set	SCPI	TSP
Run Time (seconds)	2.232	1.871

Finally, regardless of whichever command set you choose, you should consider using binary data transfers in lieu of ASCII, which is the default reading format for measurement instruments. The binary format is condensed and separates consecutive readings by byte size (four for single precision data and eight for double precision) and removes the need for a comma separator that is commonly used with the ASCII format.

As an example, we capture one reading from the DAQ6510 using ASCII format, which may read back as many as 13 characters, or thirteen bytes. One reading using single precision binary format will read back four bytes. As the number of bytes your program reads back for any given test instance increases, you can start to gain an appreciation for the binary formats and the potential throughput advantages they can provide.

Conclusion

For increasing your production test throughput, you will want to consider more than just purchasing modern equipment. Your multiplexer module options can provide you additional advantages, with reed relays having faster actuation time than electromechanical relays, and solid-state relays being the fastest of the three. Having a deep understanding of your target values and permissible tolerances will help you make the best decisions of when you can apply fixed ranges and turn off certain time-consuming instrument features, allowing you to better balance the tradeoffs of speed versus accuracy. Your communications interface can affect test performance as well, so you should consider your needs as defined by your test system and what your company approves. Finally, you should investigate the byte reducing capabilities that can be achieved when TSP programming is implemented, while also choosing to transfer large amounts of data using binary format.

Appendix A – A Speed Comparison with the Keysight 34972A

The following chart provides speed comparison information for the Model 2701, DAQ6510 and the Keysight 34972A using comparable multiplexer hardware and communication tools. The electromechanical relay multiplexer module used for the Keithley instruments is the Model 7700; for Keysight it is the 34901A. The reed relay multiplexer for the Keithley instruments is the Model 7703; for Keysight it is the 34902A. The solid-state relay multiplexer for Keithley instruments is the Model 7701.

Configuration:

- Measure Function: DC Voltage
- Auto Delay: Off
- Auto Zero: Off
- Line Sync: Off
- Calculations/Math: Off
- Ranged: 10V (fixed)
- Channel Delay: 0 s
- Trigger Delay/Scan Interval: 0s
- Readings Format: Reading only in ASCII

NPLC	0.02			0.002			0.0005		
	Relay Type	EM	Reed	SS	EM	Reed	SS	EM	Reed
Model 2701 Ethernet	12.37 s @ 80.8 rdg/s	3.74 s @ 268.5 rdg/s	3.12 s @ 320.3 rdg/s	12.37 s @ 80.8 rdg/s	3.45 s @ 289.8 rdg/s	3.10 s @ 321.7 rdg/s	-	-	-
Keysight 34972A Ethernet	16.88 s @ 59.2 rdg/s	5.14 s @ 194.9 rdg/s	N/A	-	-	-	-	-	-
DAQ6510 (SCPI) Ethernet	11.24 s @ 88.9 rdg/s	2.54 s @ 399.0 rdg/s	1.34 s @ 745.2 rdg/s	11.24 s @ 88.9 rdg/s	2.40 s @ 416.1 rdg/s	1.22 s @ 815.0 rdg/s	11.24 s @ 88.9 rdg/s	2.40 s @ 416.5 rdg/s	1.21 s @ 824.4 rdg/s
DAQ6510 (SCPI) USB	11.0 s @ 88.4 rdg/s	2.03 s @ 434.4 rdg/s	1.01 s @ 878.0 rdg/s	11.0 s @ 89.2 rdg/s	2.03 s @ 432.3 rdg/s	1.03 s @ 973.7 rdg/s	11.0 s @ 88.9 rdg/s	2.03 s @ 432.9 rdg/s	1.00 s @ 988.1 rdg/s

Appendix B – TSP Function File Loaded on to the DAQ6510

The following is the TSP (Lua) script loaded on to the DAQ6510 prior to testing which aids in reducing the communication transactions necessary to execute test sequences. All primary functions called by the upper level controlling program are highlighted.

```
function rst()
  -- Call reset w/ reduced number of characters
  reset()
  print("OK")
end

function cnfgFunc(fnc, chLst)
  -- Set the measurement functions for the scan
  if fnc == 0 then
    -- Scan Temperature
    channel.setdmm(chLst, dmm.ATTR_MEAS_FUNCTION, dmm.FUNC_TEMPERATURE)
    channel.setdmm(chLst, dmm.ATTR_MEAS_REF_JUNCTION, dmm.REFJUNCT_INTERNAL)
    channel.setdmm(chLst, dmm.ATTR_MEAS_THERMOCOUPLE, dmm.THERMOCOUPLE_K)
  elseif fnc == 1 then
    -- Scan DC Volts
    channel.setdmm(chLst, dmm.ATTR_MEAS_FUNCTION, dmm.FUNC_DC_VOLTAGE)
  elseif fnc == 2 then
    -- Scan 2-Wire Resistance
    channel.setdmm(chLst, dmm.ATTR_MEAS_FUNCTION, dmm.FUNC_RESISTANCE)
  elseif fnc == 3 then
    -- Scan 4-Wire Resistance
    channel.setdmm(chLst, dmm.ATTR_MEAS_FUNCTION, dmm.FUNC_4W_RESISTANCE)
  elseif fnc == 4 then
    -- Scan DC Amps
    channel.setdmm(chLst, dmm.ATTR_MEAS_FUNCTION, dmm.FUNC_DC_CURRENT)
  end
end

function cnfgSetRng(chLst, rng)
  -- Set the measurement function range
  channel.setdmm(chLst, dmm.ATTR_MEAS_RANGE, rng)
end

function cnfgSetNplc(chLst, nplc)
  -- Set the measurement integration time via NPLCs
  channel.setdmm(chLst, dmm.ATTR_MEAS_NPLC, nplc)
end

function enblAZero(chLst, state)
  -- Set the state of Auto Zero
  if state == 0 then
    channel.setdmm(chLst, dmm.ATTR_MEAS_AUTO_ZERO, dmm.OFF)
  else
    channel.setdmm(chLst, dmm.ATTR_MEAS_AUTO_ZERO, dmm.ON)
  end
end

function enblADelay(chLst, state)
  if state == 0 then
    channel.setdmm(chLst, dmm.ATTR_MEAS_AUTO_DELAY, dmm.DELAY_OFF)
  else
    channel.setdmm(chLst, dmm.ATTR_MEAS_AUTO_DELAY, dmm.DELAY_ON)
  end
end

function enblLimit(chLst, use1, use2)
  if use1 == 0 then
```

```

        channel.setdmm(chLst, dmm.ATTR_MEAS_LIMIT_ENABLE_1, dmm.OFF)
    else
        channel.setdmm(chLst, dmm.ATTR_MEAS_LIMIT_ENABLE_1, dmm.ON)
    end

    if use2 == 0 then
        channel.setdmm(chLst, dmm.ATTR_MEAS_LIMIT_ENABLE_2, dmm.OFF)
    else
        channel.setdmm(chLst, dmm.ATTR_MEAS_LIMIT_ENABLE_2, dmm.ON)
    end
end

function enblLineSync(chLst, state)
    if state == 0 then
        channel.setdmm(chLst, dmm.ATTR_MEAS_LINE_SYNC, dmm.OFF)
    else
        channel.setdmm(chLst, dmm.ATTR_MEAS_LINE_SYNC, dmm.ON)
    end
end

function setScanList(chLst)
    scan.create(chLst)
end

function setScanCount(cnt)
    scan.scancount = cnt
end

function setScn2Scn(stosIntvl)
    scan.scaninterval = stosIntvl
end

function scanInit()
    trigger.model.initiate()
end

function cnfgSpeedScan(fnc, chLst, rng, nplc, cnt, s2sInt)
    -- Apply the measure function
    cnfgFunc(fnc, chLst)

    -- Set the measure range; not for Temp which is fixed
    if fnc ~= 0 then
        cnfgSetRng(chLst, rng)
    end

    -- Set the measure integration rate
    cnfgSetNplc(chLst, nplc)

    -- Disable auto zero
    enblAZero(chLst, 0)

    -- Disable auto delay
    enblADelay(chLst, 0)

    -- Disable limits checking
    enblLimit(chLst, 0, 0)

    -- Disable line synchronization
    enblLineSync(chLst, 0)

    -- Set the scan list
    setScanList(chLst)

    -- Set the scan count

```

```

    setScanCount(cnt)

    -- Set the scan to scan interval
    setScn2Scn(s2sInt)

    -- Ensure that all setup operations have completed
    opc()
    print("OK")
end

function init()
    trigger.model.initiate()
    opc()
    print("OK")
end

function chkTrgMdl()
    present _state, n = trigger.model.state() -- state, present block number
    if (present _state == trigger.STATE_RUNNING) or (present _state == trigger.STATE_WAITING) then
        print("1")
    else
        print("0")
    end
end

function getRdgs()
    printbuffer(1, defbuffer1.n, defbuffer1)
end

print("functions loaded")

```

Appendix C – Python Example Using TSP Command Calls to Execute Scans

The following is the Python script loads a TSP function then executes a DCV speed scan ten times and prints the results to the display along with the program run time.

```

## Communications Optimization Consideration Program Example...
import socket
import struct
import math
import time
import datetime

ip_address = "192.168.1.69"          # Place your instrument's IP address here.
scktNum = 5025                      # Define the instrument socket number
functions_path = "scanFunctions.lua" # This file holds the set of TSP (Lua-
                                     # based) functions that are called by
                                     # the Python script to help minimize the
                                     # amount of bytes needed to setup up and
                                     # extract readings from the instrument.
                                     # The file is opened and written to
                                     # the target instrument. The calling
                                     # program only needs to call the short
                                     # function names and not a whole series
                                     # of verbose commands.

def load_functions(s):
    # This function opens the scanFunctions.lua file in the same directory
    # as the Python script and transfers its contents to the DAQ6510
    # internal memory. All the functions defined in the file are callable
    # by the controlling program.
    func_file = open(functions_path, "r")

```

```

contents = func_file.read()
func_file.close()
s.send("if loadfuncs ~= nil then "
      "script.delete('loadfuncs') "
      "end\n".encode())
s.send("loadscript loadfuncs\n{0}\nendscript\n"
      .format(contents)
      .encode())
s.send("loadfuncs()\n".encode())
print(s.recv(100).decode())

def send_reset(s):
# This function issues the reset that clears all existing
# instrument settings.
s.send("rst()\n".encode())
s.recv(10)

def send_cnfgSpeedScan(s, fnc, scanList, rng, nplc, cnt, s2sInt):
# This function configures the speed scan setup as
# defined by the passed parameters.
sndBuffer = "cnfgSpeedScan({0},{1},{2},{3},{4},{5})\n".format(fnc, scanList, rng, nplc, cnt, s2sInt)
s.send(sndBuffer.encode())
s.recv(10)

def send_init(s):
# This function issues the reset that clears all existing
# instrument settings.
s.send("init()\n".encode())
s.recv(10)

def send_waitForScan(s):
# This function issues the reset that clears all existing
# instrument settings.
s.send("chkTrgMdl()\n".encode())
response = (int)(s.recv(10))
while (response == 1):
    s.send("chkTrgMdl()\n".encode())
    response = (int)(s.recv(10))

def get_ScanData(s):
# This function extracts the scanned readings
# from the DAQ6510
s.send("getRdgs()\n".encode())
response = s.recv(1024).decode()
return response

# Main body of our program...
#configure, trigger, transfer
s = socket.socket() # Establish a TCP/IP socket object
s.connect((ip_address, scktNum)) # Connect to the instrument

nplc = 0.001 # Set the integration rate
cnt = 1 # Set the scan count
s2sInt = 0 # Set the scan to scan interval

t1 = datetime.datetime.now() # Capture the start time of our operation

load_functions(s)

send_reset(s)

# Configure DC Voltage scan
dcvScanList = '\101:110\'
rng = 10

```



```

send_cnfgSpeedScan(s, 1, dcVScanList, rng, nplc, cnt, s2sInt)

for j in range(0, 10):
    send_init(s)
    send_waitForScan(s)
    print(get_ScanData(s))

s.close()          # Close the socket.

t2 = datetime.datetime.now()    # Capture the stop time of the operation
delta = t2-t1
print("Run time: {0:.0f} ms".format(delta.total_seconds() * 1000))

input("Press Enter to continue...")
exit()

```

Appendix D – Python Example Using SCPI Command Calls to Execute Scans

The following is the Python script leverages SCPI commands to execute a DCV speed scan ten times and prints the results to the display along with the program run time. The primary commands called during the main program are highlighted, but the underlying functions require the transmission of several characters.

```

## Communications Optimization Consideration Program Example...
import socket
import struct
import math
import time
import datetime

ip_address = "192.168.1.69"          # Place your instrument's IP address here.
scktNum = 5025                      # Define the instrument socket number

def send_reset(s):
    # This function issues the reset that clears all existing
    # instrument settings.
    s.send("*RST\n".encode())
    return

def cnfgFunc(s, fnc, chLst):
    # This function sets up the measurment function across
    # scan channels.
    if fnc == 0:
        s.send('FUNC "TEMP"', (@{0})\n'.format(chLst).encode())
        s.send('SENS:TEMP:TRAN TC, (@{0})\n'.format(chLst).encode())
        s.send('SENS:TEMP:TC:TYPE K, (@{0})\n'.format(chLst).encode())
        s.send('SENS:TEMP:TC:RJUN:RSEL INT, (@{0})\n'.format(chLst).encode())
    elif fnc == 1:
        s.send('FUNC "VOLT"', (@{0})\n'.format(chLst).encode())
    elif fnc == 2:
        s.send('SENS:FUNC "RES"', (@{0})\n'.format(chLst).encode())
    elif fnc == 3:
        s.send('SENS:FUNC "FRES"', (@{0})\n'.format(chLst).encode())
    elif fnc == 4:
        s.send('SENS:FUNC "CURR"', (@{0})\n'.format(chLst).encode())

    return

def cnfgSetRng(s, fnc, rng, chLst):
    # This function sets the fixed range across scan channels.
    if fnc == 0:
        rng = 0
    elif fnc == 1:

```

```

        s.send('SENS:VOLT:RANG {0}, (@{1})\n'.format(rng, chLst).encode())
    elif fnc == 2:
        s.send('SENS:RES:RANG {0}, (@{1})\n'.format(rng, chLst).encode())
    elif fnc == 3:
        s.send('SENS:FRES:RANG {0}, (@{1})\n'.format(rng, chLst).encode())
    elif fnc == 4:
        s.send('SENS:CURR:RANG {0}, (@{1})\n'.format(rng, chLst).encode())

    return

def cnfgSetNplc(s, fnc, nplc, chLst):
    # This function sets the integration rate across scan channels.
    if fnc == 0:
        s.send('SENS:TEMP:NPLC {0}, (@{1})\n'.format(nplc, chLst).encode())
    elif fnc == 1:
        s.send('SENS:VOLT:NPLC {0}, (@{1})\n'.format(nplc, chLst).encode())
    elif fnc == 2:
        s.send('SENS:RES:NPLC {0}, (@{1})\n'.format(nplc, chLst).encode())
    elif fnc == 3:
        s.send('SENS:FRES:NPLC {0}, (@{1})\n'.format(nplc, chLst).encode())
    elif fnc == 4:
        s.send('SENS:CURR:NPLC {0}, (@{1})\n'.format(nplc, chLst).encode())

    return

def enblAZero(s, fnc, state, chLst):
    # This function sets the state of Auto Zero across scan channels
    if fnc == 0:
        s.send('SENS:TEMP:AZER {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 1:
        s.send('SENS:VOLT:AZER {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 2:
        s.send('SENS:RES:AZER {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 3:
        s.send('SENS:FRES:AZER {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 4:
        s.send('SENS:CURR:AZER {0}, (@{1})\n'.format(state, chLst).encode())
    #SENS:FRES:AZER ON
    return

def enblADelay(s, fnc, state, chLst):
    # This function sets the state of Auto Delay across scan channels.
    if fnc == 0:
        s.send('SENS:TEMP:DEL:AUTO {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 1:
        s.send('SENS:VOLT:DEL:AUTO {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 2:
        s.send('SENS:RES:DEL:AUTO {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 3:
        s.send('SENS:FRES:DEL:AUTO {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 4:
        s.send('SENS:CURR:DEL:AUTO {0}, (@{1})\n'.format(state, chLst).encode())

    return

def enblLimit(s, fnc, use1, use2, chLst):
    # This function sets the state of limits checking across scan channels.
    if fnc == 0:
        s.send('CALC2:TEMP:LIM1:STAT {0}, (@{1})\n'.format(use1, chLst).encode())
        s.send('CALC2:TEMP:LIM2:STAT {0}, (@{1})\n'.format(use2, chLst).encode())
    elif fnc == 1:
        s.send('CALC2:VOLT:LIM1:STAT {0}, (@{1})\n'.format(use1, chLst).encode())
        s.send('CALC2:VOLT:LIM2:STAT {0}, (@{1})\n'.format(use2, chLst).encode())
    elif fnc == 2:

```

```

        s.send('CALC2:RES:LIM1:STAT {0}, (@{1})\n'.format(use1, chLst).encode())
        s.send('CALC2:RES:LIM2:STAT {0}, (@{1})\n'.format(use2, chLst).encode())
    elif fnc == 3:
        s.send('CALC2:FRES:LIM1:STAT {0}, (@{1})\n'.format(use1, chLst).encode())
        s.send('CALC2:FRES:LIM2:STAT {0}, (@{1})\n'.format(use2, chLst).encode())
    elif fnc == 4:
        s.send('CALC2:CURR:LIM1:STAT {0}, (@{1})\n'.format(use1, chLst).encode())
        s.send('CALC2:CURR:LIM2:STAT {0}, (@{1})\n'.format(use2, chLst).encode())

    return

def enblLineSync(s, fnc, state, chLst):
    # This function sets the state of Line Sync across scan channels.
    if fnc == 0:
        s.send('TEMP:LINE:SYNC {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 1:
        s.send('VOLT:LINE:SYNC {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 2:
        s.send('RES:LINE:SYNC {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 3:
        s.send('FRES:LINE:SYNC {0}, (@{1})\n'.format(state, chLst).encode())
    elif fnc == 4:
        s.send('CURR:LINE:SYNC {0}, (@{1})\n'.format(state, chLst).encode())

    return

def setScanList(s, chLst):
    # This function sets the active list of scan channels.
    s.send('ROUT:SCAN:CRE (@{0})\n'.format(chLst).encode())

    return

def setScanCount(s, cnt):
    # This function sets the number of times the scan will
    # iterate.
    s.send('ROUT:SCAN:COUN:SCAN {0}\n'.format(cnt).encode())
    return

def setScn2Scn(s, stosIntvl):
    # This function sets the delay time between the start
    # of one scan to the start of the next.
    s.send('ROUT:SCAN:INT {0}\n'.format(stosIntvl).encode())
    return

def sendOpc():
    # This function is issued to allow all blocking commands
    # to complete their operation before moving on to other
    # control commands.
    s.send('*OPC\n'.format(cnt).encode())
    return

def cnfgSpeedScan(s, fnc, chLst, rng, nplc, cnt, s2sInt):
    # This function configures the speed scan setup as
    # defined by the passed parameters.
    cnfgFunc(s, fnc, chLst)
    cnfgSetRng(s, fnc, rng, chLst)
    cnfgSetNplc(s, fnc, nplc, chLst)
    enblAZero(s, fnc, 0, chLst)
    enblADelay(s, fnc, 0, chLst)
    enblLimit(s, fnc, 0, 0, chLst)
    enblLineSync(s, fnc, 0, chLst)

    setScanList(s, chLst)
    setScanCount(s, cnt)

```

```

    setScn2Scn(s, s2sInt)

    sendOpc()
    return

def send_init(s):
    # This function starts a scan.
    s.send("INIT\n".encode())
    return

def send_waitForScan(s):
    # This function is used to poll the DAQ6510 to determine the
    # active trigger state. If RUNNING or WAITING, this means the
    # trigger model is still running and the scan is not done.
    s.send("TRIG:STAT?\n".encode())
    response = s.recv(1024).decode()
    #print(response)
    while (response.find('RUNNING') != -1 or response.find('WAITING') != -1) :
        s.send("TRIG:STAT?\n".encode())
        response = s.recv(1024).decode()
    return

def get_ScanData(s, val):
    # This function extracts the scanned readings
    # from the DAQ6510
    s.send("TRAC:DATA? 1, {0}, \defbuffer1\", READ\n".format(val).encode())
    response = s.recv(1024).decode()
    return response

#=====
# Main body of our program...
#configure, trigger, transfer
s = socket.socket()          # Establish a TCP/IP socket object
s.connect((ip_address, scktNum)) # Connect to the instrument

nplc = 0.001    # Set the integration rate
cnt = 1        # Set the scan count
s2sInt = 0     # Set the scan to scan interval

t1 = datetime.datetime.now()    # Capture the start time of our operation

send_reset(s)

# Configure DC Voltage scan
dcvScanList = '101:110'
rng = 10
cnfgSpeedScan(s, 1, dcvScanList, rng, nplc, cnt, s2sInt)

for j in range(0, 10):
    send_init(s)
    sendOpc()
    send_waitForScan(s)
    print(get_ScanData(s, 10))

s.close()    # Close the socket.

t2 = datetime.datetime.now()    # Capture the stop time of the operation
delta = t2-t1
print("Run time: {0:.0f} ms".format(delta.total_seconds() * 1000))

input("Press Enter to continue...")
exit()

```


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) 3759 7627
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) 56 04 50 90
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 6917 5000
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. 090617



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.

041618 AH 1KW-61369-0

