Chapter 13 Control of Stand-Alone Instruments

HARDWARE REQUIREMENTS FOR THIS CHAPTER

To perform the exercises in this chapter, you must have a National Instruments General Purpose Interface Bus (GPIB) device connected to your computer. This device might be a PCI-GPIB board plugged into a PCI expansion slot or a GPIB-USB device attached to a USB connector. A stand-alone instrument equipped with an IEEE 488.2 compliant interface (ideally the Agilent 34401A Digital Multimeter) is also needed. This instrument is connected to a PCI-GPIB board using a GPIB cable; a GPIB-USB device connects directly to the instrument.

If, while performing the chapter exercises, the communication between the GPIB device and instrument breaks down (e.g., caused by an accidental error in your programming), communication can often be restored either by turning the instrument off and then on again, or by restarting your computer.

In previous chapters, you have used LabVIEW software to transform a personal computer (connected to an appropriate National Instruments DAQ device) into several handy laboratory instruments. In particular, you programmed this system to become a DC voltmeter, digital oscilloscope, spectrum analyzer, waveform generator, and digital thermometer. Pause to consider the following tantalizing prospect: Perhaps the only instrument required in a modern-day laboratory is a DAQ device-equipped computer controlled by LabVIEW software. That is, by simply writing a collection of appropriate VIs, it might be possible for you—the contemporary scientific researcher—to satisfy all of your laboratory instrumentation needs with this single LabVIEW-based data acquisition and generation system. This system's tremendous flexibility would then obviate the need to purchase an expensive collection of stand-alone electronic equipment such as power supplies, function generators, picoammeters, spectrum analyzers, and oscilloscopes.

The functioning VIs that you have written in previous chapters demonstrate that the above "tantalizing prospect" can, at least in certain situations, be realized. But don't discard your stand-alone instruments just yet. The timing, speed, sensitivity, and simultaneous data-taking requirements of many contemporary research experiments are beyond the capabilities of your DAQ device. For instance, while the LabVIEW-based digital oscilloscope we constructed worked well for observing audio-range frequencies (less than 20 kHz), it would prove miserably inadequate at displaying the several nanosecond-wide voltage pulses emanating from a photomultiplier tube. In this latter situation, a stand-alone digital scope with a very fast analog-to-digital converter (on the time scale of several gigasamples per second) would do the job nicely. Thus stand-alone instruments play a central role in state-of-the-art research, and so it might not surprise you to find that they, too, fall under the scope of LabVIEW.

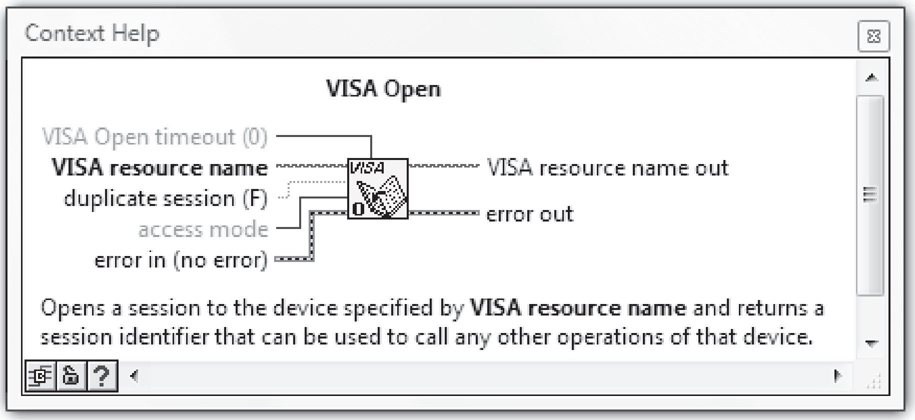
Over the past few decades, a message-based communications standard has evolved by which stand-alone instruments can be software-controlled using a personal computer. In this communications scheme, a particular instrument obeys an array of manufacturer-defined ASCII character commands that represent all the possible ways of manually pressing buttons, turning dials, and viewing output data on its front panel. While the hardware conduit (called an *interface bus*) through which these ASCII messages are passed between the PC and laboratory instrument can take on various guises (including RS-232, GPIB, Ethernet, and USB), there is a single set of LabVIEW icons available to control this communication process. This icon set is named *VISA* (short for Virtual Instrument Software Architecture) and is found in **Functions>>Instrument I/O>>VISA**.

In this chapter, you will learn how to use VISA icons to control the message-based communication between a stand-alone instrument and your computer. You will explore generic features of this communication process such as the Standard Commands for Programmable Instruments (SCPI) language and various synchronization methods while writing code that controls a particular stand-alone instrument—the *Agilent 34401A Digital Multimeter*—using a particular interface bus—the *General Purpose Interface Bus*.

13.1 THE VISA SESSION

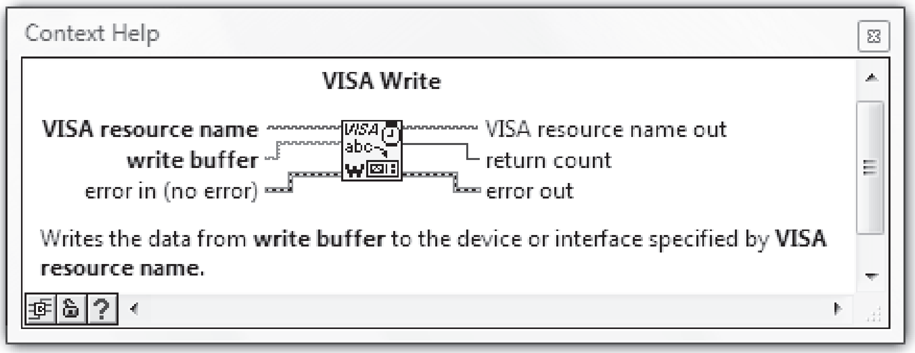
When using VISA icons to facilitate message-based communication between a computer and a particular stand-alone instrument, the instrument is termed a *VISA resource* and the communication activity is called a *VISA session*. To *"query"* a VISA resource (i.e., send it a command, then receive back its response), the required VISA session consists of the following four steps: *open the session*, *write the command message to the resource*, *read the response from the resource*, *close the session*. In **Functions>> Instrument I/O>>VISA** (and its subpalette **VISA Advanced**), the following four icons are available to perform the four given steps—**VISA Open**, **VISA Write**, **VISA Read**, and **VISA Close**. To understand how to wire these four icons together to query an instrument, we will first briefly describe the function of each individual icon.

The Help Window for **VISA Open** is shown in the following illustration. The job of this icon is to begin a VISA session between your computer and the resource defined at its **VISA resource name** input. The **VISA resource name** consists of text that specifies the interface type being used (e.g., GPIB or USB), the address of the resource (a number we'll discuss in a few minutes), and the resource type. For our work, the resource type will be a stand-alone instrument denoted by *INSTR*. To pass the **VISA resource name** to other VISA icons, this quantity is available at the **VISA resource name out** output terminal.



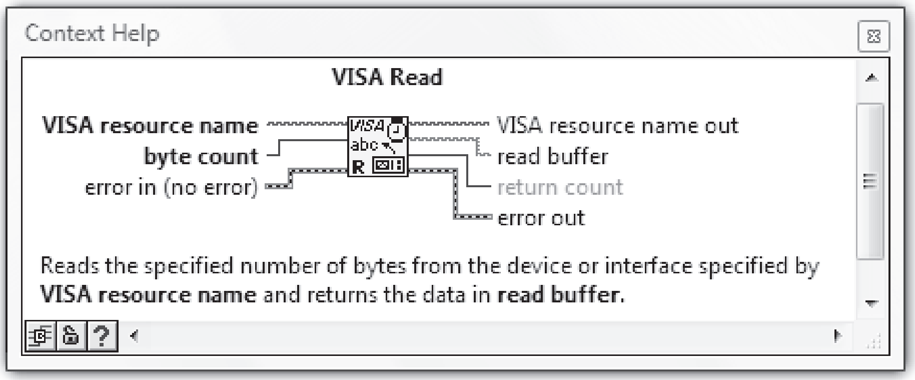
Opens a session to the device specified by **VISA resource name** and returns a session identifier that can be used to call any other operations of that device.

VISA Write, whose Help Window follows below, performs the actual ASCII message transfer from your computer to the stand-alone instrument. Once presented with the open session's **VISA resource name**, this icon writes the ASCII string at its **write buffer** input to the instrument. This string is one of the commands recognized by the instrument and, when received by the instrument, configures it properly for a desired data-taking measurement. Additionally, the VISA resource name is available at the **VISA resource name out** output terminal.



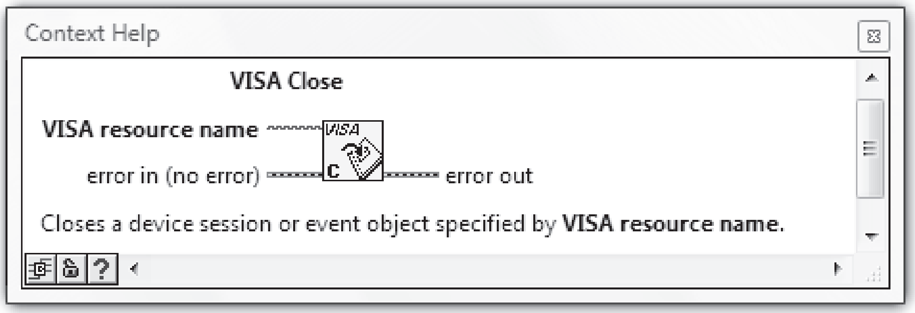
Writes the data from **write buffer** to the device or interface specified by **VISA resource name.**

Next, the Help Window for **VISA Read** is shown. This icon transfers the response message from the stand-alone instrument into your computer's memory. When given the open session's **VISA resource name**, this icon receives the ASCII response string consisting of (a maximum of) **byte count** number of bytes and outputs this string at its **read buffer** terminal. This string typically contains the results of a data-taking measurement performed by the instrument. Additionally, the VISA resource name is available at the **VISA resource name out** output terminal.



Reads the specified number of bytes from the device or interface specified by **VISA resource name** and returns the data in **read buffer.**

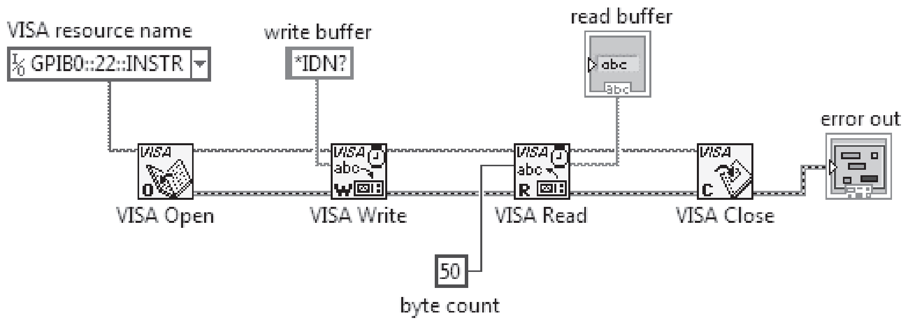
Finally, **VISA Close**'s Help Window is given below. This icon closes the VISA session specified at its **VISA resource name** input.



Closes a device session or event object specified by **VISA resource name.**

Note that all four of these VISA icons include error reporting via an error cluster, which is appears at the **error in** and **error out** terminals.

The four-step VISA session to query an instrument is accomplished by wiring these four VISA icons together as follows.



In this example, the message **IDN?* is sent over a GPIB interface to an instrument at address 22. **IDN?*, a command recognized by most instruments, instructs the instrument to identify itself. After receiving this command, the instrument's response (which is an ASCII string consisting of identification information) is received by the computer over the GPIB and displayed in the **read buffer** front-panel string indicator.

Similar to the File I/O and DAQmx icons that you have studied previously, the wiring scheme of VISA icons takes advantage of the principle of LabVIEW programming called *data dependency*. Simply stated, data dependency means that an icon cannot execute until data is available at *all* of its inputs. In the previous diagram, all of **VISA Open**'s required inputs (given in bold on the Help Window) are wired. So, when this diagram is run, **VISA Open** executes immediately. Upon completion, **VISA Open** outputs a VISA resource name at its **VISA resource name out** terminal, which is passed through the wiring to VISA Write's **VISA resource name** input. Because of data dependency, **VISA Write** cannot execute until it receives the VISA resource name from **VISA Open.** In a similar way, **VISA Read** cannot execute until **VISA Write** completes, and so on. Thus, through this programming scheme, we are assured that the icons will execute in the desired sequence: **VISA Open** followed by **VISA Write** followed by **VISA Read** followed by **VISA Close**.

Also the correct manner of chaining together VISA icons for error reporting is shown above. If an error does occur at one point in the chain, subsequent icons will not execute and the error message will be passed to the **error out** indicator cluster.

Because VISA-based programming is so robust, you can write highly dependable data-taking programs with just the information already presented. However, with a bit more grounding in the message-based communication scheme, you'll be able to create programs in which you can have near-total confidence. The following paragraphs will take you to the next level of sophistication in stand-alone instrument control.

13.2 THE IEEE 488.2 STANDARD

When remote control of laboratory instruments first became possible, there was a chaotic period during which, more or less, each instrument manufacturer defined its own communications protocol through a unique blend of parallel and serial modes, positive and negative polarities, and assorted handshaking signals. In 1965, Hewlett-Packard (now named Agilent) ended this cacophony by designing a universal instrument interface called the Hewlett-Packard Interface Bus (HP-IB) and offered it as the only option on all of its new computer-programmable instruments. Because of its high transfer rates, HP-IB quickly gained popularity with other instrument manufacturers and, in 1975, was accepted as an industry-wide standard known as IEEE 488 or, more commonly, the General Purpose Interface Bus (GPIB). In 1987, an improved version of this standard called IEEE 488.2 was adopted, which enhanced and strengthened message-based communication by specifically defining an instrument's minimally required communication capabilities, a protocol for message exchange, a generic set of commonly needed commands, and a status reporting system. Today, most computer-controlled laboratory instruments are IEEE 488.2 compliant, even those that communicate over interface buses other than the GPIB (such as Ethernet and USB).

13.3 COMMON COMMANDS

One important innovation of the IEEE 488.2 standard was the introduction of a standardized set of "*common commands*" for the many generic operations that all instruments must perform. The mnemonics for these common commands begin with asterisks to delineate them from the other device-specific commands recognized by a particular instrument. All IEEE 488.2 compliant instruments, at the very least, are required to recognize the subset of 13 common commands given in the following table. Many of these commands are related to the reporting of events using two status registers called the SBR and SESR, which will described in detail starting in the next paragraph.

Table 13.1 Common Commands for IEEE 488.2 Compliant Instruments

MANDATORY COMMON COMMANDS	FUNCTION
*IDN?	Reports instrument identification string.
*RST	Resets instrument to known state.
*TST?	Performs self-test and reports results.
*OPC	Sets operation complete (OPC) bit in SESR upon completion of command.
*OPC?	Returns "1" to the output buffer upon completion of command.
*WAI	Waits until all pending operations complete execution.
*CLS	Clears status registers.
*ESE	Enables event-recording bits in SESR.
*ESE?	Reports enabled event-recording bits in SERS.
*ESR?	Reports value of SESR.
*SRE	Enables a SBR bit to assert the SRQ line.
*SRE?	Reports SBR bits that are enabled to assert the SRQ line.
*STB?	Reports the contents of the SBR.

13.4 STATUS REPORTING

Another IEEE 488.2 innovation is a standardized scheme for *status reporting*. This status reporting system is available to inform you of significant events that occur within each instrument connected to an interface bus. In this scheme, each instrument is equipped with two status registers, called the *Standard Event Status Register (SESR)* and the *Status Byte Register (SBR)*. Each bit in these registers records a particular type of event that may occur while the instrument is in use, such as an execution error or the completion of an operation. When the event of a given type occurs, the instrument sets the associated status register bit to a value of one, if that bit has previously been enabled (see the following). Thus by reading the status registers, you can tell what events have transpired.

The Standard Event Status Register, which is schematically shown here, records eight types of events that can occur within a data-taking instrument.

Standard Event Status Register (SESR)

7	6	5	4	3	2	1	0
PON	URQ	CME	EXE	DDE	QYE	RQC	OPC

The eight events associated with the eight bits of the SESR are described in the following table. In our work, the OPC bit will be most useful.

Table 13.2 Eight SESR Events

BIT	ASSOCIATED EVENTS OF SESR
7 (MSB)	**PON** (Power On): Instrument was powered off and on since the last time the event register was read or cleared.
6	**URQ** (User Request): Front-panel button was pressed.
5	**CME** (Command Error): Instrument received a command with improper syntax.
4	**EXE** (Execution Error): Error occurred while instrument was executing a command.
3	**DDE** (Device Error): Instrument is malfunctioning.
2	**QYE** (Query Error): Attempt was made to read the instrument's output buffer when no data was present, or a new command was received before previously requested data had been read from the output buffer.
1	**RQC** (Request Control): Instrument requests to be controller.
0 (LSB)	**OPC** (Operation Complete): All commands prior to and including an *OPC command have been executed.

The SESR exists as an event-signaling tool for you to use in your programs. However, this status register completely lacks initiative and will not perform any work unless you request it to do so. Thus, when initiating communications with an instrument, one of the messages that you may wish to send is an instruction that activates the subset of event-reporting SESR bits that are of interest to you. For instruments that conform to the IEEE 488.2 standard, this activation process is accomplished via the **ESE* (Event Status Enable) command. For example, supposed you wished the QYE bit to be activated and thus record any execution errors in the SESR's bit 2. Since 000001002 = 410, the QYE bit can be activated by performing a VISA Write of the ASCII command **ESE 4* to the instrument. In our work to come, we will activate the OPC bit with the command **ESE 1*.

The Status Byte Register, which is schematically shown next, records whether data is available in the instrument's output buffer, whether the instrument requests service, and whether the SESR has recorded any events.

Status Byte Register (SBR)

7	6	5	4	3	2	1	0
—	RQS	ESB	MAV	—	—	—	—

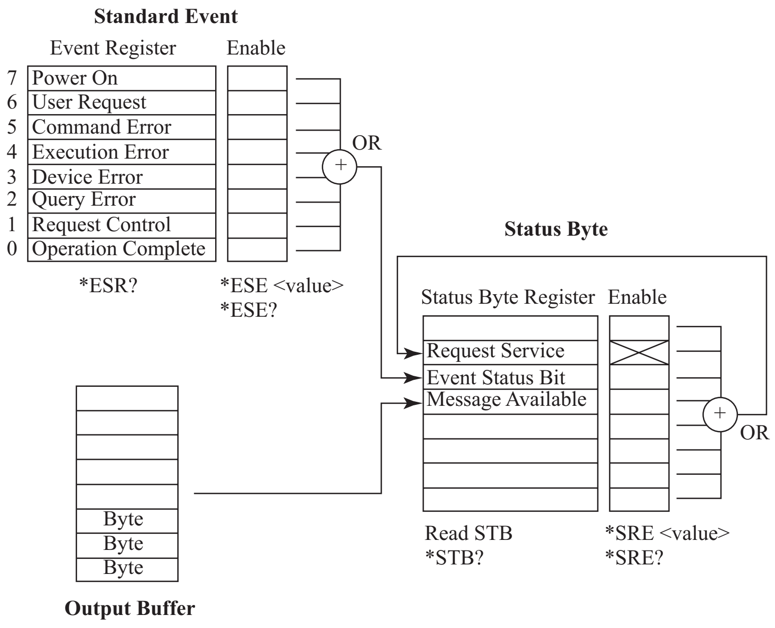
The functions of the eight bits of the SBR are described in the following table. The SBR bits are studious, performing their status reporting duties without need of a request from you.

Table 13.3 Functions of the Eight SBR Bits

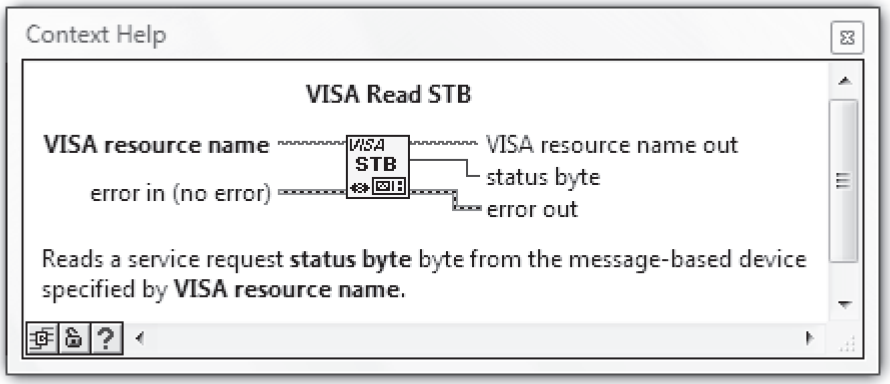
Bit	Function of SBR Bit
7 (MSB)	May be defined for use by instrument manufacturer.
6	**RQS** (Request Service): The instrument has asserted the SRQ line because it requires service from the GPIB controller.
5	**ESB** (Event Status Bit): An event associated with an enabled SESR bit has occurred.
4	**MAV** (Message Available): Data is available in the instrument's output buffer.
3—0	May be defined for use by instrument manufacturer.

An instrument can be configured to assert a *Service Request (SRQ)*, which is a digital signal carried on a dedicated wire within the 24-wire GPIB cable, in response to either of two events—an event detected by the Standard Event Status Register or the presence of previously requested data in the instrument's output buffer (that is, the assertion of the ESB or MAV bit, respectively). This configuration process is accomplished on IEEE 488.2 instruments by using the **SRE* (Service Request Enable) command. For example, if you wish an event detection by the SESR to trigger a request for service by the instrument, initialize the instrument by writing the ASCII command **SRE 32* to the instrument. Since $32_{10} = 00100000_2$, the setting of the SBR's fifth (ESB) bit will then be the criterion for the instrument asserting a SRQ. If, instead, you wish the presence of data in the output buffer (signaled by the MAV bit being set) to trigger a SRQ, then write **SRE 16*. Finally, **SRE 0* will disable the instrument ability to assert a SRQ.

The relationship between the Standard Event Status Register, the instrument's output buffer, and the Status Byte Register (along with the common commands that configure and query each) is illustrated in the following diagram.



As an alternative to the use of the SRQ, *serial polling* is a common method for determining the status of an instrument. In a serial poll process, the interface bus queries an instrument and the instrument responds by returning the value of the bits in its Status Byte Register. A serial poll is easily accomplished in LabVIEW using **VISA Read STB**, found in **Functions>>Instrument I/O>>VISA**. This icon's Help Window is shown here.



Reads a service request **status byte** byte from the message-based device specified by **VISA resource name.**

13.5 DEVICE-SPECIFIC COMMANDS

Finally, each stand-alone instrument is designed for a specialized purpose and has its own idiosyncratic methods for accomplishing its objectives. Thus, every programmable instrument comes with a set of *device-specific commands* that allow the user to control its functions remotely and to transfer the information it produces into a computer's memory. The array of device-specific commands for an instrument is listed in its user manual. This set of commands is defined by the instrument's maker… and therein lies a problem. When surveying the user manuals for programmable instruments of varying models and manufacturers, you will find a great diversity in the style of the various command sets. Some (especially those associated with older model instruments) are an alphabetized collection of cryptic one- or two-character strings (the designers' thinking was obviously "shorter commands yield quicker and, therefore, better computer-instrument communication"). At the other extreme are the user-friendly sets, with similar commands logically grouped, each represented by an easy-to-read-and-remember mnemonic.

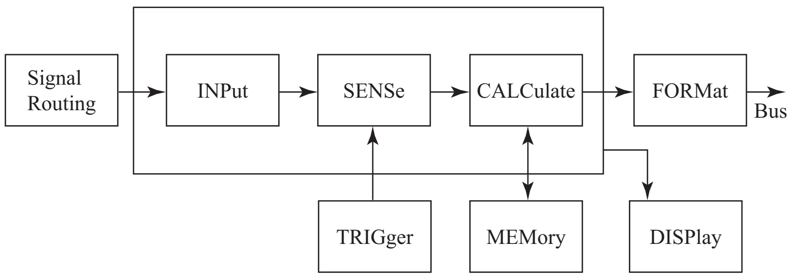
As programmable instruments have come into wider use, it has become apparent that development costs and unscheduled delays can be diminished markedly by simplifying the instrument programmers' task whenever possible. Thus, user-friendly device-specific command sets are the rule, rather than the exception, for instruments being currently manufactured. Commonly, these command sets are organized in a hierarchical *tree structure*, similar to the file system used in computers. Each of an instrument's major functions, such as **TRIG**ger, **SENS**e (alternately, **MEAS**ure), **CALC**ulate, and **DISPlay**, define a *root* and all commands associated with that root form its *subsystem*. So, for example, to configure the Agilent 34401A Digital Multimeter to measure a DC voltage whose value is expected to fall within the range of ±10 Volts (an action within its **SENS**e subsystem), the appropriate command is as follows:

```
SENSe:VOLTage:DC:RANGe 10
```

Here, *SENSe* is the root keyword and colons (:) represent the descent to the lower-level *VOLTage*, then *DC*, then lowest-level *RANGe* keywords. Finally, *10* is a parameter associated with *RANGe*. While the full command mnemonic given here can be sent to the instrument, it is only absolutely necessary to send the capitalized characters.

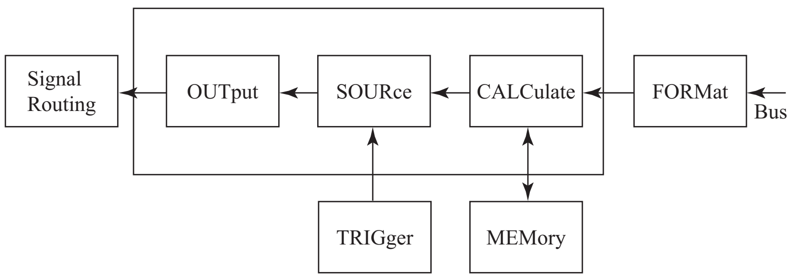
In 1990, a consortium of equipment manufacturers defined the *Standard Commands for Programmable Instruments (SCPI)* in an effort to standardize the device-specific command sets of computer-controlled instrumentation. While this standard has not been universally adopted, it is not uncommon to discover that your post-1990 instrument is SCPI-compliant. As a means of categorizing generally applicable command groups, the SCPI standard posits the following model for a generic programmable instrument. An instrument that performs measurements on an input signal is assumed to have the root functions shown in the next diagram. Here, for example, **SENS**e includes any action involved in the actual conversion of an incoming signal to internal data, such as setting the range, resolution, and integration time, while **INP**ut consists of actions that condition the signal prior to its conversion, such as filtering, biasing, and attenuation.

SIGNAL MEASUREMENT INSTRUMENT



Alternately, an instrument that generates signals is modeled by the following diagram.

SIGNAL GENERATION INSTRUMENT



The SCPI command set is organized in a hierarchical tree structure using the syntax illustrated above by the Agilent 34401A Multimeter command. You'll learn more about the SCPI command syntax as you work your way through this chapter. But maybe now is a good time to dive in and actually control a stand-alone instrument.

13.6 SPECIFIC HARDWARE USED IN THIS CHAPTER

In designing this chapter, I faced the following problem. There are thousands of computer-controllable stand-alone instruments available for purchase from the myriad of worldwide scientific instrument makers. Each of these instruments communicates using one (or many times, a few) of the handful of available interface buses. I have a small subset of these instruments in my laboratory and I can use them there to practice the art of message-based communication. You also, hopefully, have a small subset of such instruments available to practice with in your own laboratory. What's the problem? Well, because of the high cost and specialized nature of such equipment, the probability that my subset and your subset have some common instrument is most likely very small. The unfortunate thing about this situation is that each stand-alone instrument is designed to take specialized measurements and understands its own unique set of ASCII commands (which are defined by its maker and are listed in its user manual). Thus, before attempting to control a particular instrument using an interface bus, the programmer must have a detailed understanding of the measurement that that instrument is designed to take, the procedure that it implements in doing its work, and the command list that it recognizes. All of these considerations greatly constrain the writing of a set of generic laboratory exercises that everyone can perform.

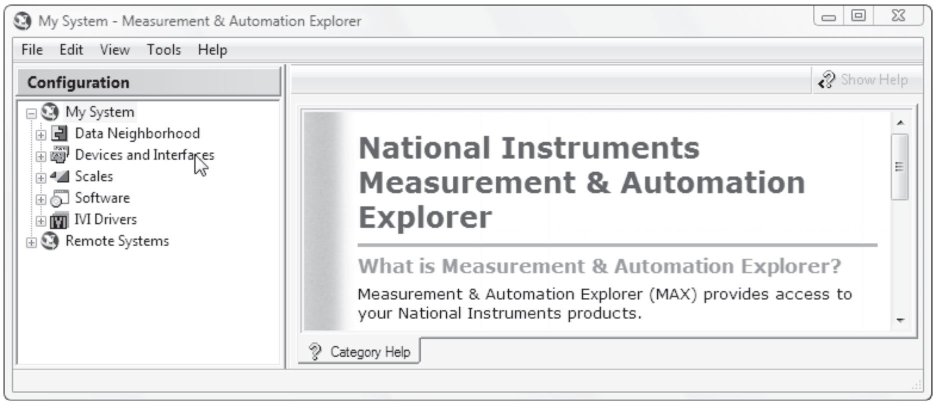
That said, I still was faced with the fact that I had to choose a particular instrument and interface bus to work with in this chapter's exercises. For the interface bus, I chose the GPIB because it is the interface you will almost certainly encounter in your computer-based laboratory work. Although USB and Ethernet are gaining in popularity with scientific instrument makers (due to the fact that most PCs come equipped with these interfaces), GPIB is currently (by far) the most widely used interface bus for laboratory equipment. With an estimated 10 million GPIB-equipped instruments in use in research and industry worldwide, the GPIB most likely retain its popularity for many years to come. For the instrument used in this chapter's exercises, I chose the Agilent 34401A Digital Multimeter for the following reasons. First of all, this instrument measures voltage, current, and resistance—vanilla-flavored quantities that require no specialized knowledge to understand (unlike, for instance, the control of grating angle and slit size in a spectrometer). Second, for such a high-quality and useful GPIB-equipped instrument, its price tag of approximately $1000 makes it extremely affordable. Every lab should have one and many do! Third, this instrument is both IEEE 488.2 and SCPI compliant. Thus, the following exercises, rather than being narrowly tied to one specific device, can be much more universally applicable by demonstrating the generic features of these widely used standards, such as command syntax and status reporting.

In the best circumstance, an Agilent 34401A (earlier purchased units of this same instrument are named the Hewlett-Packard 34401A) is already available for your use or, with a modest investment, you can purchase this worthwhile instrument. Then, without need for modification, you can straightforwardly work your way through the given exercises to learn the basics of message-based communication. If, instead, you have some other interface-equipped instrument available, try reading the following pages to understand the generic issues being investigated. Then, by consulting the user manual, it may be fairly easy, for instance, to use the interface-appropriate VISA resource name and substitute an ASCII command string here and there in order to adapt the exercises to your particular instrument and interface bus. If neither of the above describes your situation, simply read through the following pages. I believe you will learn some valuable features of instrument control that will serve you well in future work.

13.7 MEASUREMENT & AUTOMATION EXPLORER (MAX)

To carry out the exercises in this chapter, you must have a National Instruments GPIB device connected properly to your computer, which in turn is connected (e.g., by a GPIB cable) to a GPIB-equipped stand-alone instrument. Also the GPIB device's driver software (called *NI-488.2*) must be correctly installed. To verify that these conditions are met, we will use the handy utility *Measurement & Automation Explorer*, which is nicknamed *MAX*.

To open MAX, either select **Tools>>Measurement & Automation Explorer…** (if you have an open VI or **Getting Started** window), or else double-click on MAX's desktop icon (if available). After MAX opens, double-click on **Devices and Interfaces** in the **Configuration** box. This action will command MAX to determine all of the National Instrument devices present within your computing system.

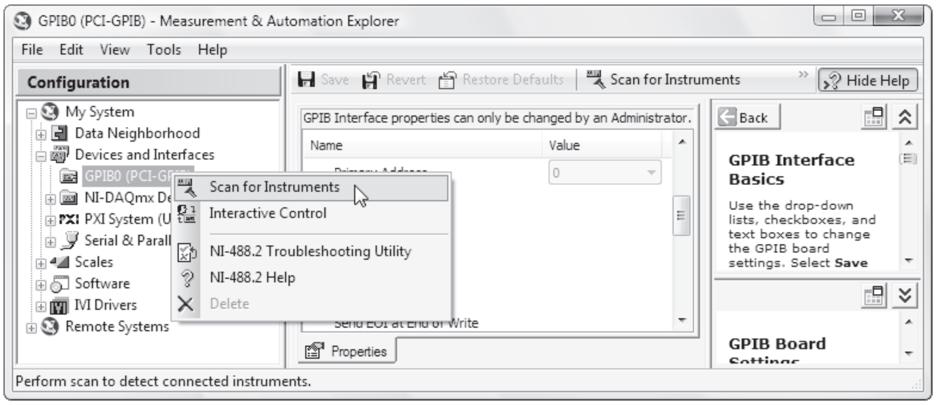


National Instruments Measurement & Automation Explorer

What is Measurement & Automation Explorer?

Measurement & Automation Explorer (MAX) provides access to your National Instruments products.

MAX will list the findings of its device survey in hierarchical tree fashion as shown next. If a GPIB device is connected correctly to your computer, a folder labeled **GPIB0** will appear in the resulting list (your system may have a different number than *0* in the folder's label). To find all of the stand-alone instruments properly connected to this device, right-click on the **GPIB0** folder and select the **Scan for Instruments** option. Alternately, you can click on the **Scan for Instruments** button near the top of the window.

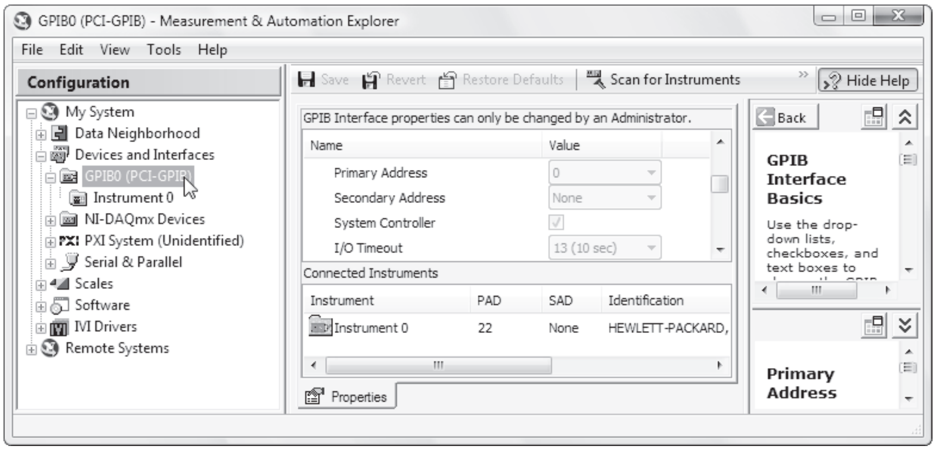


GPIB Interface properties can only be changed by an Administrator.

Use the drop-down lists, checkboxes, and text boxes to change the GPIB board settings. Select **Save**

Perform scan to detect connected instruments.

In a few moments, MAX will complete the scan. To view its results, double-click on the **GPIB0** folder. For the case shown below, one stand-alone instrument was found and information about it is stored in the folder labeled **Instrument 0**.



GPIB Interface properties can only be changed by an Administrator.

Name	Value
Primary Address	0
Secondary Address	None
System Controller	✓
I/O Timeout	13 (10 sec)

Connected Instruments

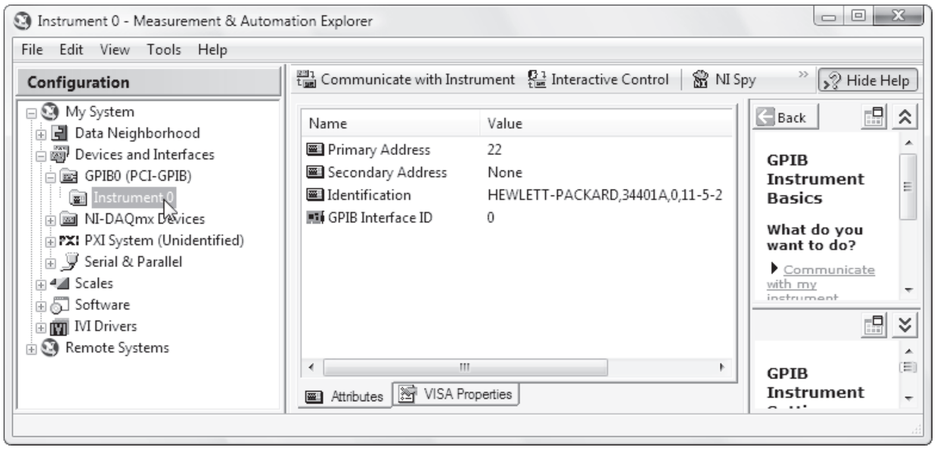
Instrument	PAD	SAD	Identification
Instrument 0	22	None	HEWLETT-PACKARD,

Use the drop-down lists, checkboxes, and text boxes to

Because up to 15 instruments can be connected to a single GPIB device, each instrument has an identification number called its *GPIB address*. A GPIB address can be any integer from 0 to 30 and is typically defined via a hardware DIP switch setting within the instrument or a sequence of button-pressing and/or knob turning on its front panel. The instrument's user manual will describe the method for setting its address.

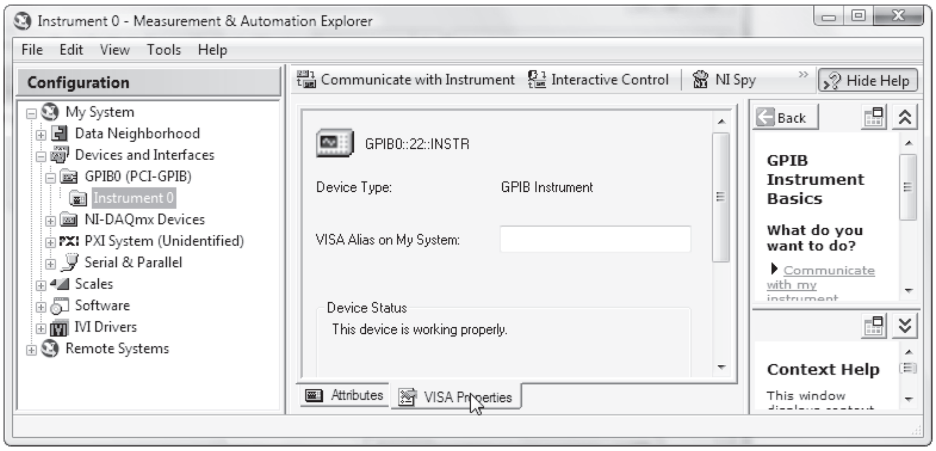
The GPIB address of the instrument found in the **Scan for Instruments** operation is found by double-clicking on the **Instrument 0** folder. After the double-click, in the box associated with the **Attributes** tab, we find that the instrument's GPIB address is

22 (the address of your instrument may be different). A text description identifying the instrument also appears.



Name	Value
Primary Address	22
Secondary Address	None
Identification	HEWLETT-PACKARD,34401A,0,11-5-2
GPIB Interface ID	0

Clicking on the **VISA Properties** tab, we find that the correct **VISA resource name** for this instrument is *GPIB0::22:INSTR* and also are told (under **Device Status**) that the instrument is working (i.e., communicating) properly.



GPIB0::22::INSTR

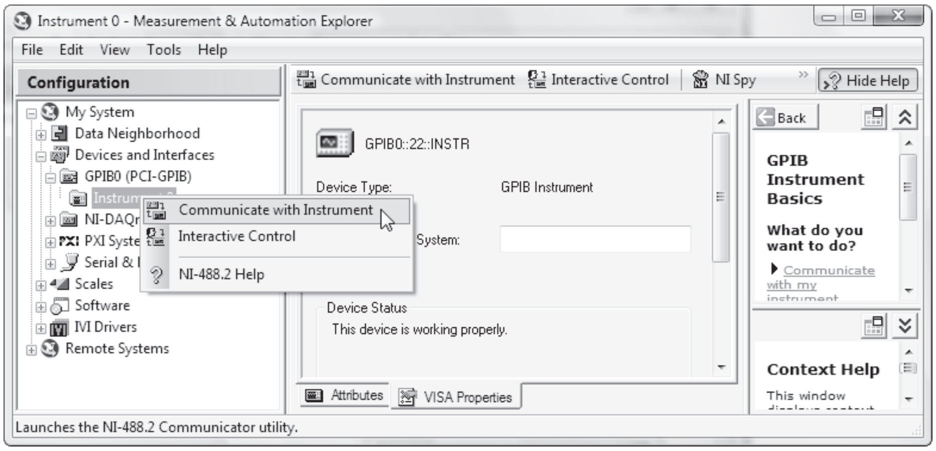
Device Type: GPIB Instrument

VISA Alias on My System:

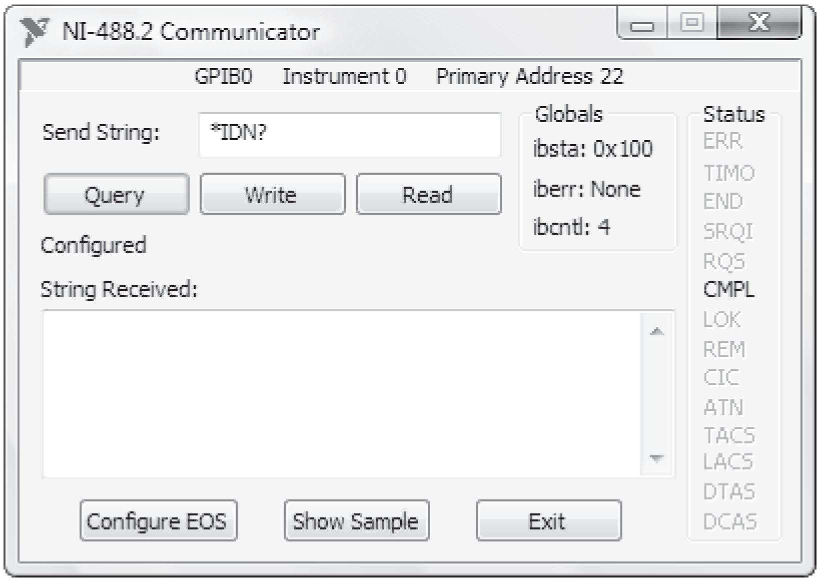
Device Status

This device is working properly.

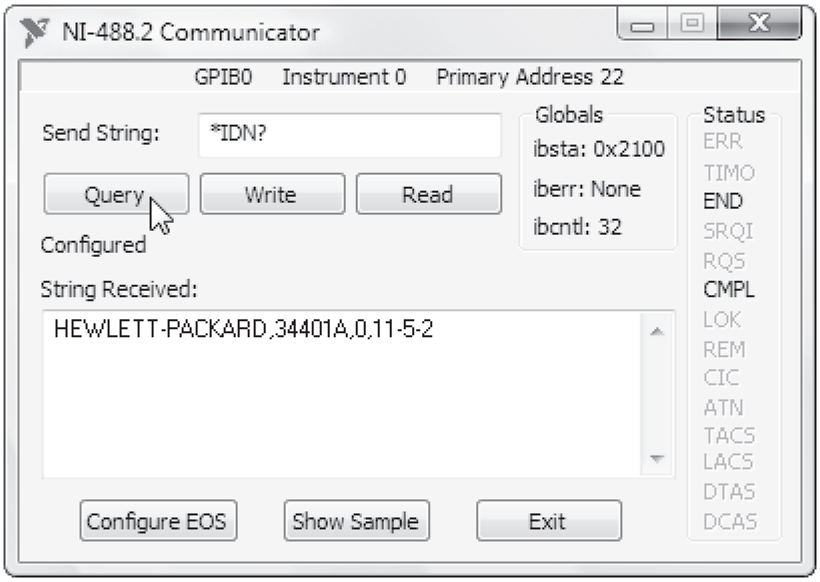
To verify that the instrument is indeed properly communicating over the GPIB, right-click on the **Instrument 0** folder and select **Communicate with Instrument**. Alternately, you can click on the **Communicate with Instrument** button near the top of the window.



An interactive dialog box will appear. Here, after typing an ASCII message in the **Send String:** box, a mouse-click on the **Query** button will carry out a write-then-read action. That is, the message in **Send String:** will be written over the GPIB to the selected instrument, and then the instrument's ASCII response will be read back over the GPIB to the computer and displayed in the **String Received:** box. When the dialog window opens, its **Send String:** box is preloaded with *_IDN?_, the IEEE 488.2 common command for an instrument to identify itself.



Click on the **Query** button. If your GPIB communication is configured properly, the identification string received from the instrument will appear in the **String Received:** box. For the Agilent 34401A Digital Multimeter used here, this identification string identifies the instrument's manufacturer and model number followed by some integers that denote the version numbers of installed firmware that controls the multimeter's three internal microprocessors.



Send String: *IDN?

Configured

ibsta: 0x2100
iberr: None
ibcntl: 32

String Received:

```
HEWLETT-PACKARD,34401A,0,11-5-2
```

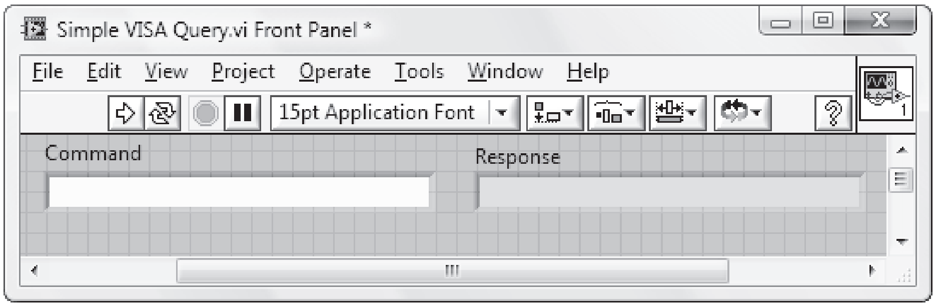
For future reference, this interactive dialog window is a handy tool for use in determining correct command syntax when developing message-based communication programs for a new instrument.

Exit this dialog window and then close MAX.

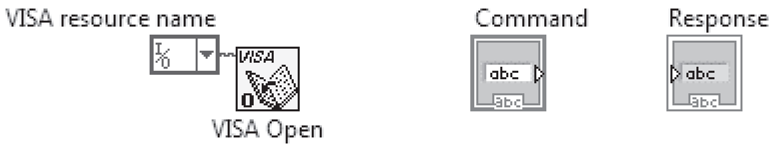
13.8 SIMPLE VISA-BASED QUERY OPERATION

Let's begin by writing a VISA-based program that carries out the query (i.e., write-then-read) action that you just completed using MAX.

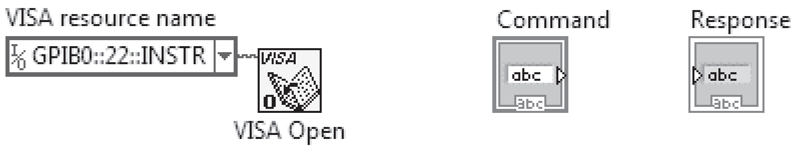
On the front panel of a blank VI, place a **String Control** and a **String Indicator**, then label them **Command** and **Response**, respectively. As shown in the next diagram, you'll want to resize these objects so that they can display strings much larger than their default sizes allow. Using **File>>Save**, first create a new folder named **Chapter 13** within the **YourName** folder, then save this VI under the name **Simple VISA Query** in **YourName\Chapter 13**.


Simple VISA Query.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font
Command
Response

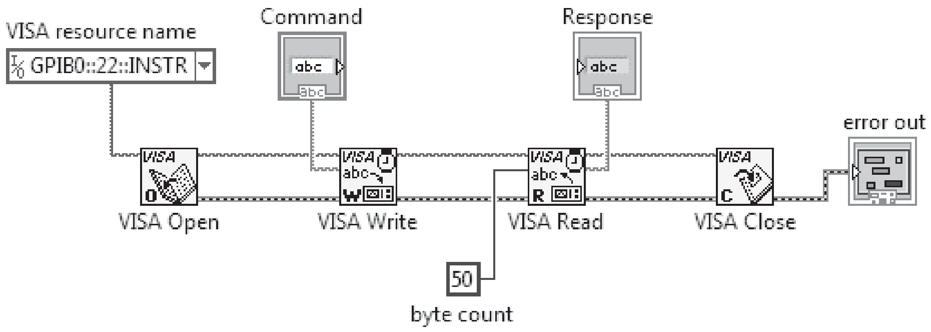
Switch to the block diagram and place a **VISA Open** icon (found in **Functions>> Instrument I/O>>VISA>>VISA Advanced**) there. Pop up on its **VISA resource name** input and create a **VISA Resource Name Constant** using **Create>>Constant**.



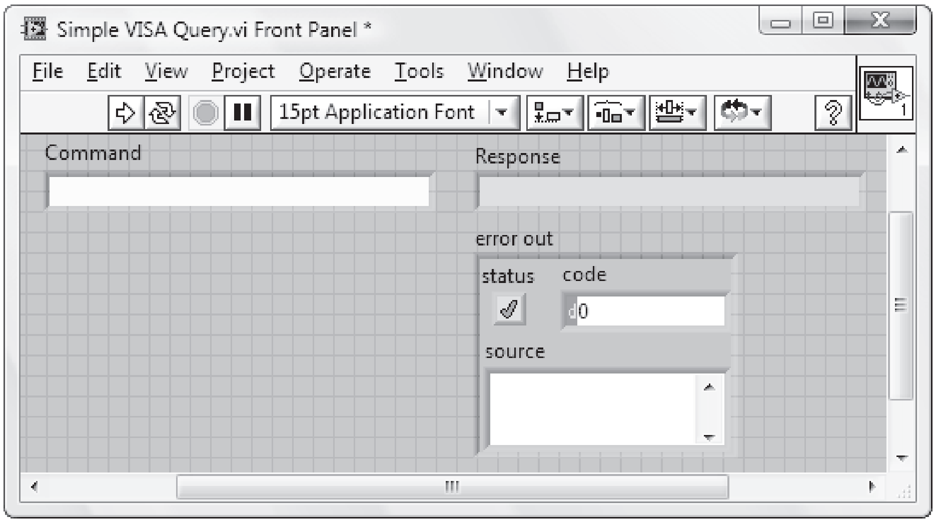
You must now load the **VISA Resource Name Constant** with the **VISA resource name** of the instrument with which you wish to communicate. By clicking on the Constant's *menu button* with the , you will be presented with the list of VISA resources that MAX found when it performed the **Scan for Instruments** operation. You can then simply choose the desired instrument from this list. Alternately, you can manually enter the appropriate **VISA resource name** for your instrument (as found using MAX) into the . The syntax for a **VISA resource name** is *Interface Bus Name::Resource Address::Resource Type*.



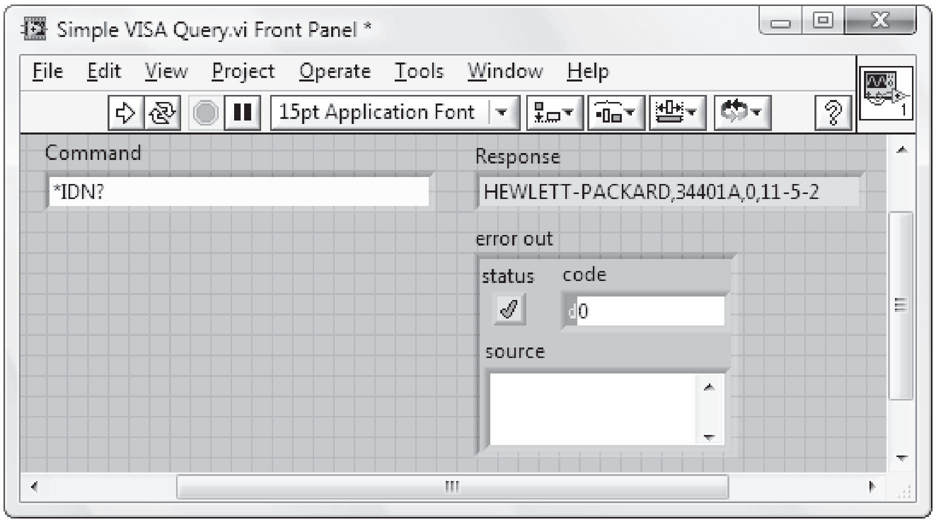
Complete the block diagram as shown next using the VISA icons found in **Functions>> Instrument I/O>>VISA** (and its subpalette **VISA Advanced**). Wire the **Command** control terminal to VISA Write's **write buffer** input and the **Response** indicator terminal to VISA Read's **read buffer** terminal. When executed, **VISA Read** will read up to *N* bytes from the selected resource, where *N* is equal to the integer wired to its **byte count** input. In a moment, we will read the Agilent 34401A identification string. According to this instrument's user manual, its identification string can be up to 35 characters long. Thus, wire the **byte count** input to an integer (**U32**) greater than or equal to 35 (I used *50*) as shown. Create the **error out** indicator cluster using the pop-up menu option **Create>>Indicator**.



Return to the front panel, arrange the objects as you wish, and then save your work.


Simple VISA Query.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font
Command
Response
error out
status
code
0
source

Enter **IDN?* into **Command**, then run your VI. If all goes well, **Response** will display the instrument's identification string upon completion of the VI execution, as shown.


Simple VISA Query.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font

Command: *IDN?

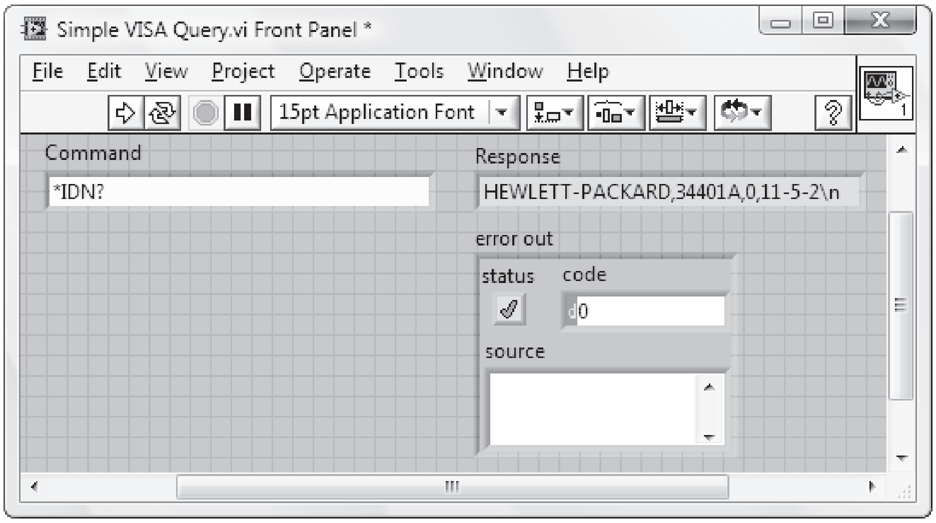
Response: HEWLETT-PACKARD,34401A,0,11-5-2

error out
status ✓ code 0
source

Simple VISA Query will leave the multimeter in *remote mode* with its triggering circuitry "idled." You can return to *local* mode, which continuously "triggers" measurements, by pressing the instrument's front-panel SHIFT/LOCAL key.

13.9 MESSAGE TERMINATION

At the conclusion of a message-transfer process, some method must be used to signal that the complete message has been passed. The IEEE 488.2 standard appoints the ASCII LF (line feed, also called new line) as its special *end of string* (EOS) character. That is, when receiving a message string, the LF character is always interpreted by the receiver as the last byte of a message. Thus, appending LF to a command string is one method of signaling message termination in IEEE 488.2 communication. Alternately, the IEEE 488.2 standard allows the assertion of an *end or identify* (EOI) while the last character in the string is being passed as another acceptable termination method. The EOI is a digital signal on a dedicated wire within the GPIB cable. When using VISA icons to control an IEEE 488.2 compliant instrument, message termination is taken care of automatically, allowing you to remain blissfully ignorant of this lower-level activity. If you'd like to view an example of this (usually invisible) message termination activity, pop up on the **Response** indicator on the front panel of **Simple VISA Query**, then select **'\' Code Display**. The *\n* character you see at the end of the identification string is the backslash code for LF. The multimeter appended this termination character to its identification string to the alert the receiver (in this case, the GPIB device) that the message has ended.


Simple VISA Query.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font

Command: *IDN?

Response: HEWLETT-PACKARD,34401A,0,11-5-2\n

error out
status
code: 0
source

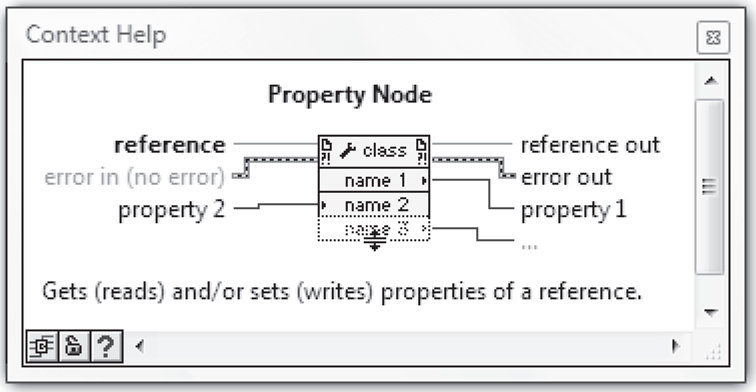
To deactivate backslash coding, pop up on **Response** and select **Normal Display.**

13.10 GETTING AND SETTING COMMUNICATION PROPERTIES USING A PROPERTY NODE

In addition to message termination, there are other low-level functions connected with message-based communication. Many of these low-level functions have an associated parameter setting, which is termed a *VISA property*. VISA assigns default values for these properties and, as long as the VISA-based programs that you write fall within the scope of these default settings, the VISA icons will automatically take care of these low-level functions without any programming effort needed by you (as demonstrated by the message termination example shown above). At times, however, you will most likely write programs that fall outside the scope of the default VISA property settings and so you will need to assign nondefault values to these quantities. Reading ("getting") and writing ("setting") VISA property values can be done within your programs using a *Property Node* (and also can often be done in MAX).

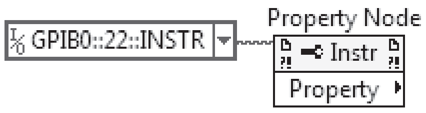
As a concrete example of a VISA property, consider **VISA Read**'s *timeout*, which is a fail-safe feature of **VISA Read** that prevents a program from running endlessly if an error occurs. If, for example, an instrument which is being queried doesn't seem to be responsive (perhaps a nameless experimenter forgot to flip on the instrument's power switch), **VISA Read** will only wait for the instrument's response for a certain number of milliseconds (given by the value of the VISA property named **Timeout**) before aborting the read operation and issuing an error message.

The default timeout value for VISA Read on your system can be determined using a **Property Node**. The Help Window for a **Property Node**, which is found in **Functions>>Instrument I/O>>VISA>>VISA Advanced**, is shown next.

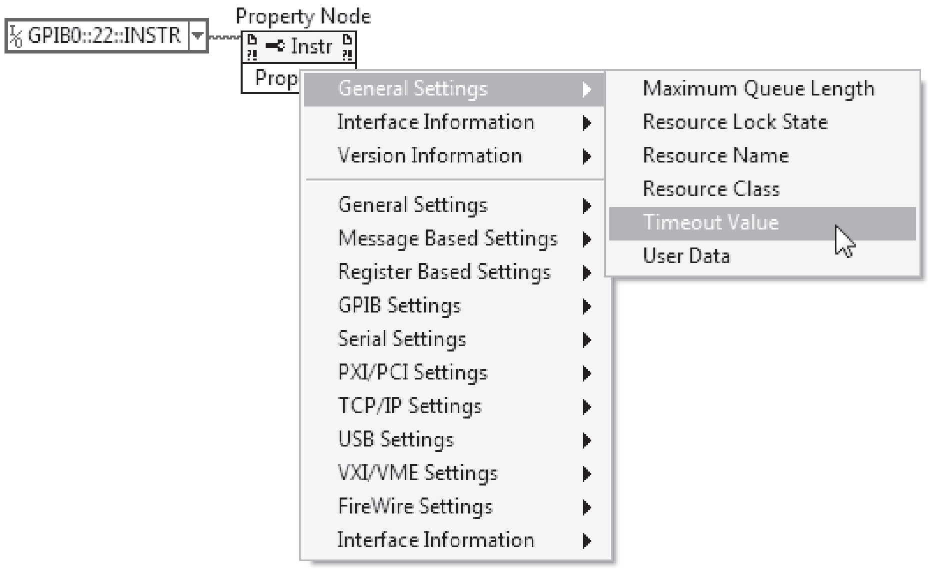


Gets (reads) and/or sets (writes) properties of a reference.

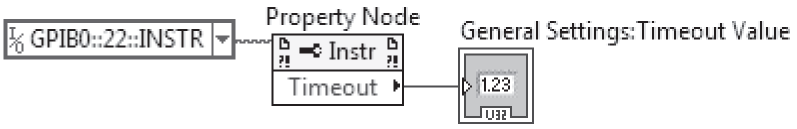
Write the following VI, which reads the current **Timeout** value on your system. Open a new VI, and save it under the name **Get Timeout Value** in **YourName\Chapter 13**. On a new block diagram, place a **Property Node** and then, using **Create>>Constant**, wire the **VISA resource name** for your instrument to the Property Node's **reference** input as shown.



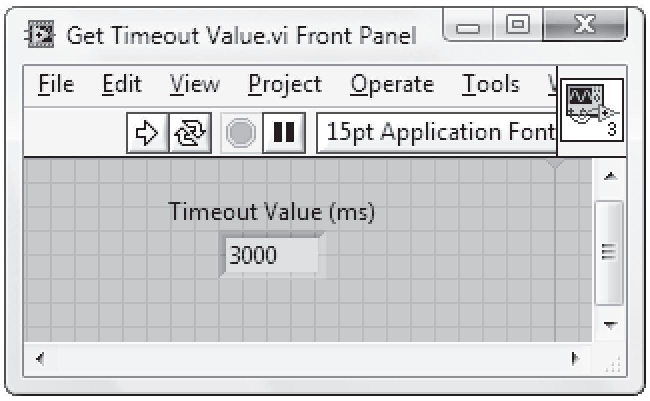
Next, using the , click on the **Property** terminal and select **General Settings>>Timeout Value**. You might explore what other Properties appear in this menu, many of which are specific to a particular interface bus.



Note that, within the **Timeout** terminal, a small arrow at the right points outward from the terminal's interior. This outward-directed arrow indicates that the **Timeout** terminal is configured as an indicator, that is, it reads ("gets") the current **Timeout** value. Using **Create>>Indicator**, create a front-panel indicator to display the value of **Timeout**.



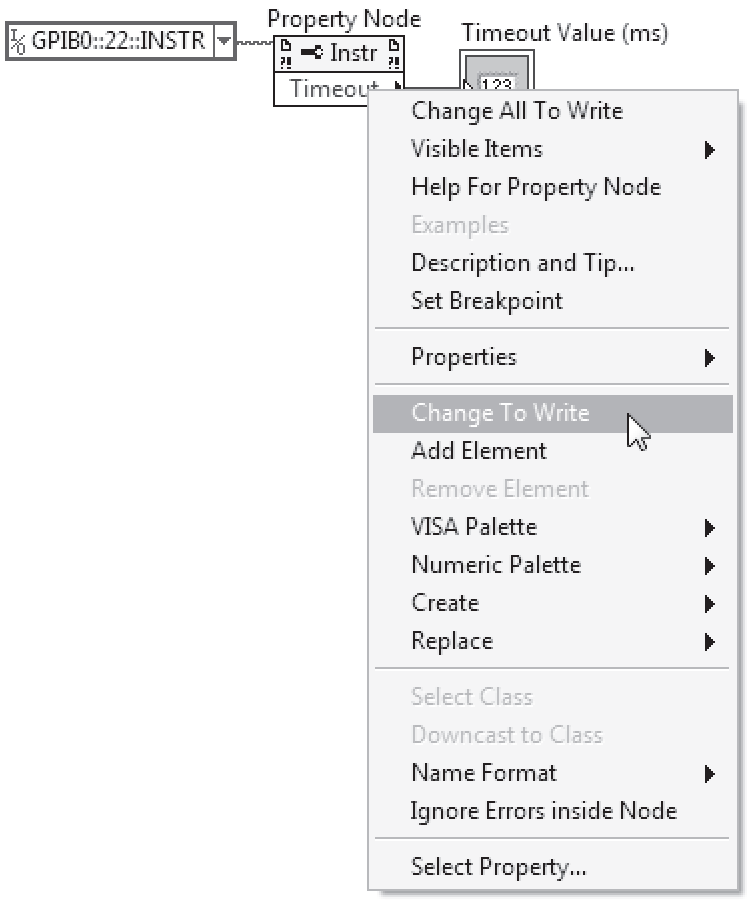
Switch to the front panel, change the indicator label to **Timeout Value (ms)**, then save your work. Run the VI. As shown below, the (default) **Timeout** value for my system is 3000 ms = 3 seconds.


Get Timeout Value.vi Front Panel
File Edit View Project Operate Tools
15pt Application Font
3

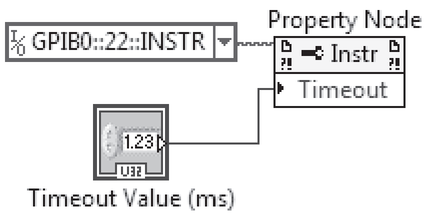
Timeout Value (ms)
3000

You can also use a Property Node to set the value of a VISA property. To demonstrate this procedure, with **Get Timeout Value** open, use **Save As...** to create a new program called **Set Timeout Value**, and store it in **YourName\Chapter 13**. When run, **Set Timeout Value** will change VISA Read's **Timeout** to a value input from its front panel.

On the block diagram of **Set Timeout Value**, pop up on the **Timeout** terminal and select **Change To Write**.



Note that, within the **Timeout** terminal, the small arrow is now at the left pointing inward toward the terminal's interior. This inward-directed arrow indicates that the **Timeout** terminal is configured as a control, that is, it writes ("sets") the **Timeout** value. Delete the **Timeout Value (ms)** indicator terminal, then using **Create>>Control**, create a front-panel control labeled **Timeout Value (ms)**.



Return to the front panel and save your work. Set **Timeout Value (ms)** to be *10000*, and then run **Set Timeout Value**. Next, run **Get Timeout Value**. Is **Timeout** now equal to 10000 ms = 10 seconds? Try setting **Timeout** equal to *8000*. You will find that only certain values for **Timeout** are allowed. LabVIEW takes the value you input to **Set Timeout Value** as a suggestion (rather than an order) and sets **Timeout** to the nearest allowed value.

13.11 PERFORMING A MEASUREMENT OVER THE GPIB

Now that **Simple VISA Query** has given us a template for the VISA query process, let's try controlling a real measurement. Hook up some known DC voltage difference, say 5 or 6 Volts, between the HI and LO Voltage Inputs of the Agilent 34401A Multimeter. This instrument's user manual instructs us that delivering the following sequence of ASCII commands will result in one DC voltage sample being acquired, and then loaded into the instrument's output buffer (which is part of its interface circuitry):

```
CONF:VOLT:DC<Space>10,0.000001
INIT
FETC?
```

Here is the meaning of this secret code. First, the Agilent 34401A can be programmed to perform 11 different types of measurement functions, including DC voltage, AC voltage, DC current, AC current, resistance, and frequency. Given these options, the first command instructs the instrument that we desire to take a DC voltage measurement. The full command is *CONFigure:VOLTage:DC <Space><Range>,<Resolution>* (this command actually executes a collection of commands drawn from the Agilent 34401A's **INP**ut, **SENS**e, **TRIG**ger, and **CALC**ulate root subsystems). The command mnemonic *CONFigure:VOLTage:DC* is constructed in the hierarchical tree structure, typical of SCPI-compliant instruments. *CONFigure* is the root-level keyword and colons (:) represent the descent to the lower-level *VOLTage*, then lowest-level *DC* keywords. While the full command mnemonic can be sent to the instrument, it is only absolutely necessary to send the capitalized characters. Separated from the command mnemonic *CONF:VOLT:DC* by a <*Space*>, the numerical values for two measurement parameters—<*Range*> and <*Resolution*>—are specified. <*Range*> selects among the instrument's five available voltage measurement scales. Each scale offers a different sensitivity with <*Range*> giving the maximum measurable value on a particular scale. The five available ranges are 100 mV, 1V, 10 V, 100 V, and 1000 V. In our situation of measuring a signal of approximately 5 V, the 10 V scale is appropriate. <*Resolution*> specifies the precision of the measurement, with the options of three levels of accuracy—4½, 5½, and 6½ digits (the ½ digit means that the most significant decimal place can only take on a value of "1" or "0"). Thus on the 10 V scale, voltages can either be resolved at the level of 0.001, 0.0001, or 0.00001 Volts. The trade-off in requesting higher accuracy is that the measurement takes a longer time. In the command sequence above, the highest resolution of 6½ digits is selected by setting <*Resolution*> equal to *0.00001* when <*Range*> equals *10*. Note the syntax of the *CONF* command, which obeys the conventions of the SCPI language: A *comma* (,) separates the parameters from each other and a <*Space*> separates the mnemonic from the parameters.

Once the multimeter has been configured for the desired measurement function as described in the previous paragraph, the data-taking process is begun by sending the *INITiate* command (from the **TRIG**ger root subsystem). Upon receipt of *INIT*, the multimeter will acquire the requested voltage sample, and then store this value in its internal memory. Finally, the *FETCh?* command (from the **MEM**ory root subsystem) instructs the instrument to transfer the reading in its internal memory to its interface-related output buffer.

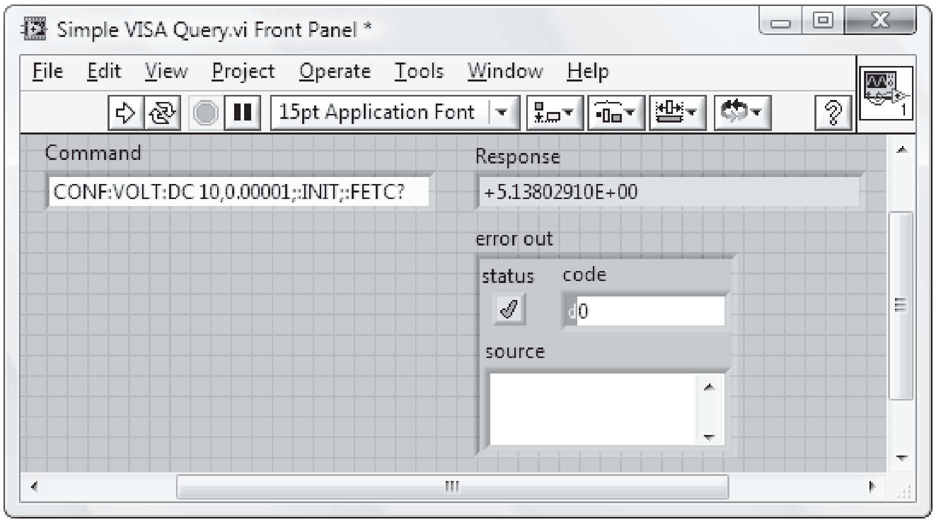
We'd like now to place this command sequence into **Simple VISA Query**. Since there are three commands to be sent, it appears that we must modify the VI to include a sequence of three successive implementations of **VISA Write**. While you are free to do so, a much easier solution is available. The SCPI language allows the programmer to concatenate several commands into one long multicommand string that can be sent in a single **VISA Write** statement. The syntax for this concatenation process is as follows:

- Use a semicolon (;) to separate two commands within the string.
- Begin a command with a colon (:), if it has a different root-level than the command preceding it. The first command in the concatenated string and IEEE 488.2 common commands (which begin with an asterisk) do not require a leading colon.

Since each of our three commands has a different root-level, applying the above rules results in the following concatenated string:

CONF:VOLT:DC<Space>10,0.00001;:INIT;:FETC?

Type this command into the **Command** control on the front panel of **Simple VISA Query** as shown next. Run the VI. Your computer will instruct the multimeter to acquire a 6½ digit voltage reading, retrieve this value, and then display it on the front panel in the **Response** indicator. Cool, eh?


Simple VISA Query.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font

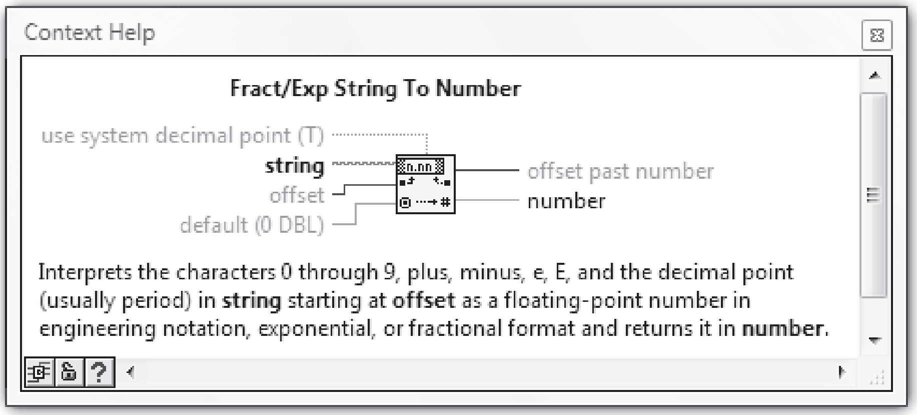
Command: CONF:VOLT:DC 10,0.00001;:INIT;:FETC?

Response: +5.13802910E+00

error out
status ✓
code 0
source

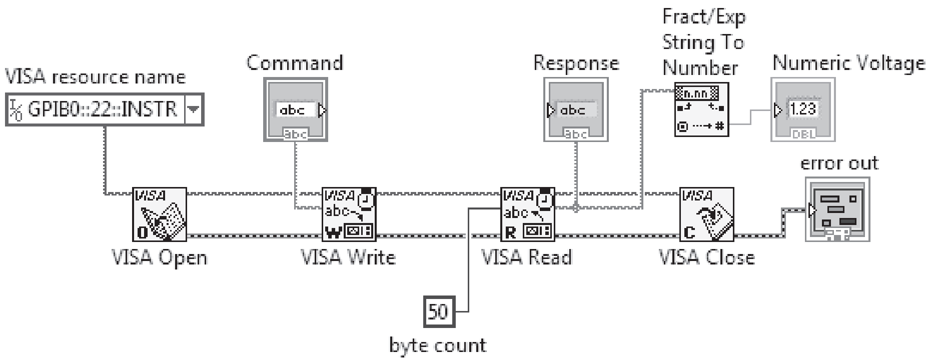
Note that, while extra digits are displayed, the value within **Response** is only accurate to the fifth decimal place.

As shown above, the Agilent 34401A reports its data samples in the form of an ASCII character string using the exponential format SD.DDDDDDDDESDD, where S is a positive or negative sign, D is a numeric digit, and E is an exponent. For future reference, note that the string that represents a data sample is 15 bytes long. If you want to use this reading as input to a mathematical calculation (a common situation), you will need to convert the string representation into a numerical format. Such conversion operations can be easily accomplished in LabVIEW using the array of conversion icons found in **Functions>>String**. In the present case, use **Fract/Exp String To Number** in **Functions>>Programming>>String>>String/Number Conversion**. The Help Window for this icon is given next.

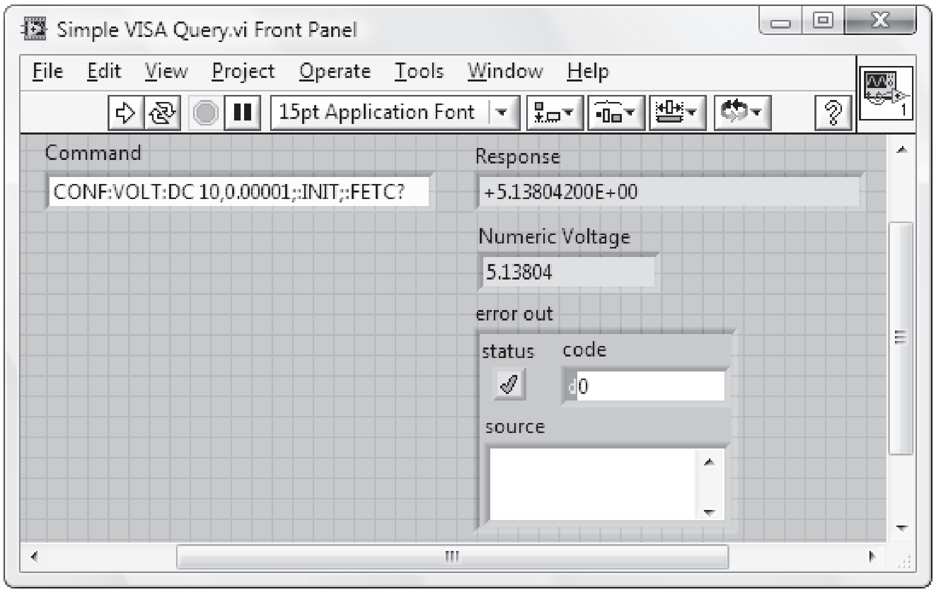


Interprets the characters 0 through 9, plus, minus, e, E, and the decimal point (usually period) in **string** starting at **offset** as a floating-point number in engineering notation, exponential, or fractional format and returns it in **number**.

Place a **Numeric Indicator** on the front panel of **Simple VISA Query** and label it **Numeric Voltage**. Use **Display Format…** in this indicator's pop-up menu to make its **Digits of precision** equal to *5*, and disable **Hide trailing zeros**. Then modify the block diagram as follows.



Run the VI to verify that the string-to-number conversion icon performs as expected.


Simple VISA Query.vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font

Command

CONF:VOLT:DC 10,0.00001;:INIT;:FETC?

Response

+5.13804200E+00

Numeric Voltage

5.13804

error out

status ✓ code 0

source

13.12 SYNCHRONIZATION METHODS

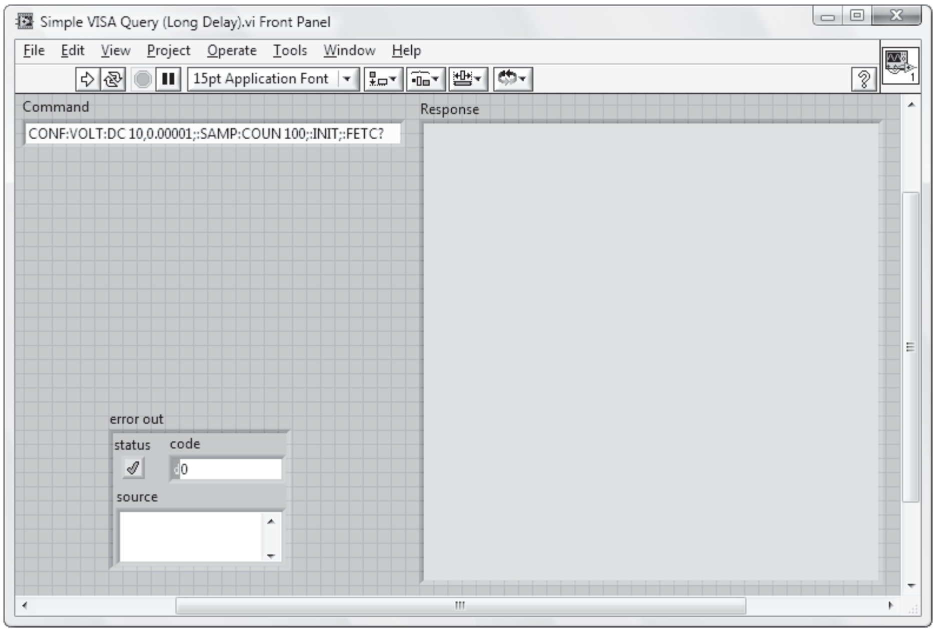
Although most ASCII commands are completed quickly after being received by a programmable instrument, some commands start a process that requires a significant amount of time (such as acquiring a large amount of data or moving an object from Point A to Point B). The time required for such processes must be taken into account when writing a data acquisition program, else, upon execution, the program may request data before it is available, induce undesirable motion, or cause some other chaotic outcome.

As an example, in its default configuration, the Agilent 34401A multimeter acquires one data sample after receipt of the *INITiate* command, then stores this measured value in its internal memory. However, through use of the *SAMPle:COUNt <Space><Value>* command, the multimeter can be instructed to take and store multiple data samples upon receiving *INITiate*. The Agilent 34401A is configured to acquire 100 DC voltage samples with 6½-digit resolution via the following concatenated string of commands:

CONF:VOLT:DC<Space>10,0.00001;:SAMP:COUN<Space>100;:INIT;:FETC?

The *FETCh?* command will load the 100 acquired samples from the multimeter's internal memory (which, by the way, can hold up to a maximum number of 512 measured values) into the instrument's interface-related output buffer.

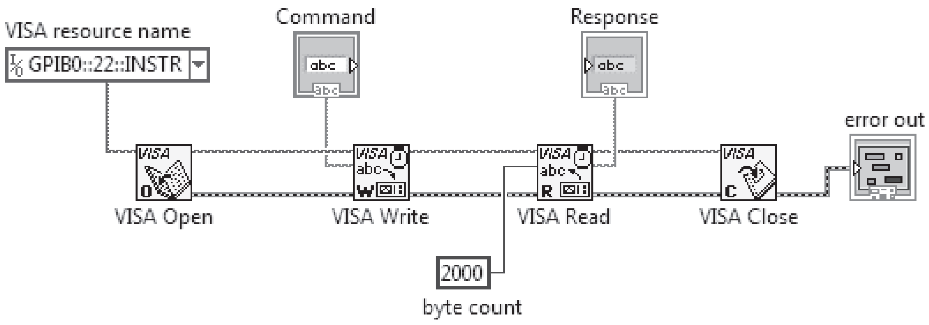
Let's write a VI that uses the given command string to gather a sequence of 100 voltage samples. Open **Simple VISA Query**, then use **Save As…** to create a new VI called **Simple VISA Query (Long Delay)**. Delete **Numeric Voltage** from the front panel and enlarge **Response** so that it can display a very long string (which is the concatenation of 100 voltage values). Type the command given above into the **Command** control. Once entered into **Command**, you can keep this command permanently loaded there by selecting **Edit>>Make Current Values Default**.


Simple VISA Query (Long Delay).vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font
Command
Response
error out
status
code
source

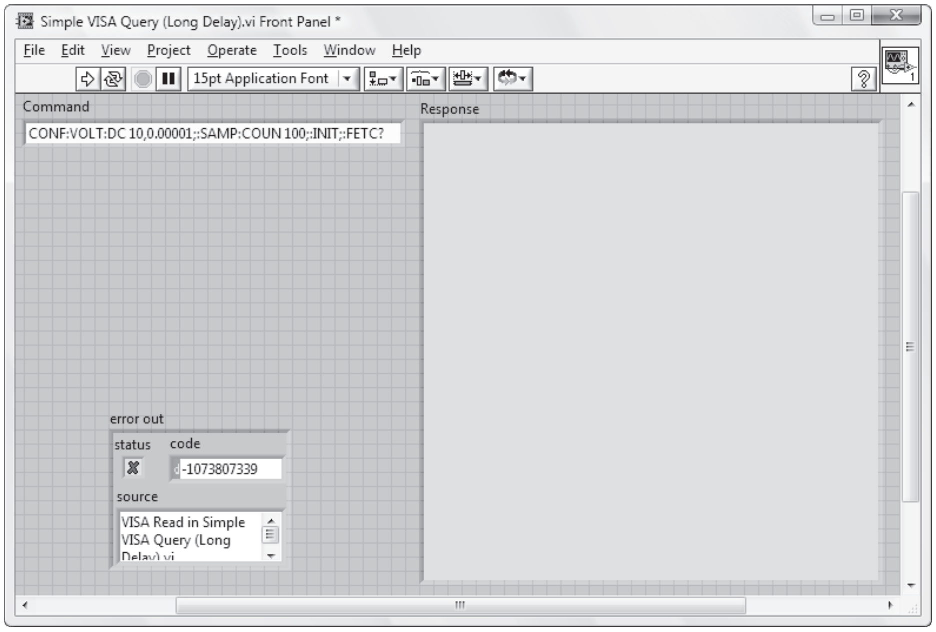
CONF:VOLT:DC 10,0.00001;:SAMP:COUN 100;:INIT;:FETC?

0

Switch to the block diagram. Delete **Fract/Exp String To Number**. After the given command string is written to the multimeter by **VISA Write**, **VISA Read** will receive a string containing the 100 voltage samples. Since each voltage sample is reported as a 15-byte string and a (single) delimiting ASCII character will be needed to separate each sample, this 100-sample string is expected to be about (100 × 15) + 100 × 1600 bytes long. Input an integer larger than 1600 to **byte count** as shown.



Run **Simple VISA Query (Long Delay)**. Count down the seconds 3. ..2. ..1. .. Disappointed? You will find that your VI does not display even one voltage sample (let alone the expected 100 values), but rather produces an error.

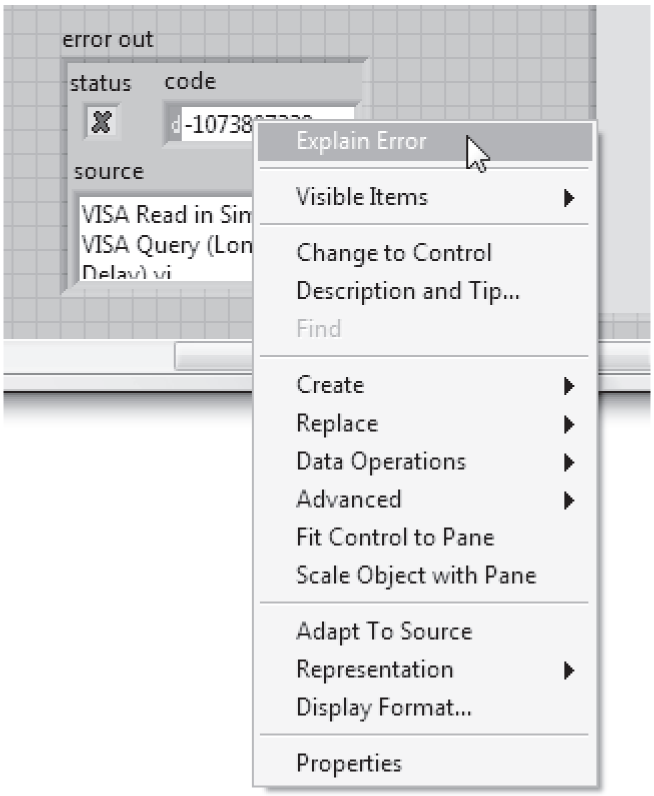

Simple VISA Query (Long Delay).vi Front Panel *

Command
CONF:VOLT:DC 10,0.00001;:SAMP:COUN 100;:INIT;:FETC?

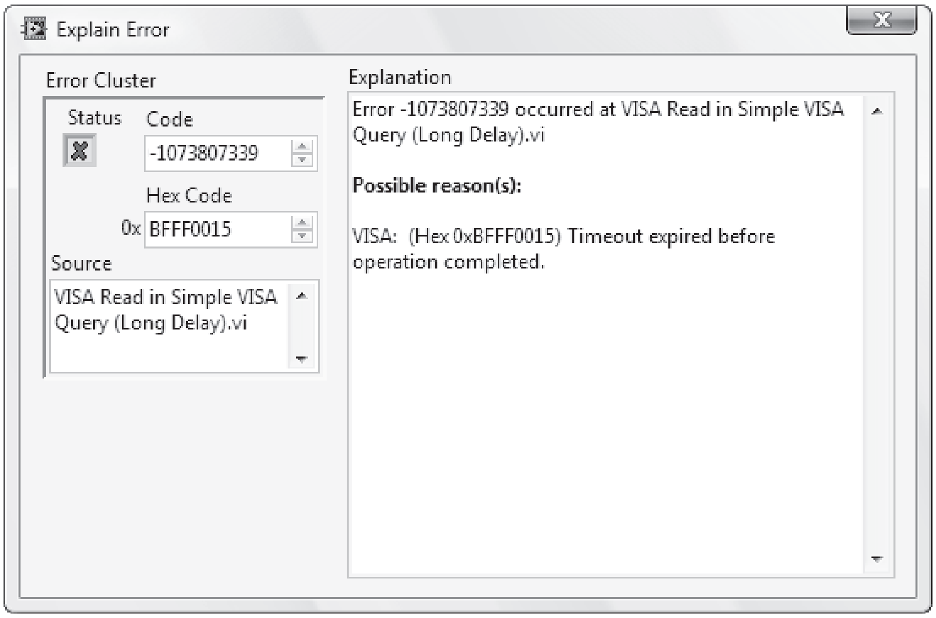
Response

error out
status code
-1073807339
source
VISA Read in Simple VISA Query (Long Delay).vi

To find out what produced this error, pop up on the **code** indicator within the **error out** cluster and select **Explain Error**.



A dialog box appears, where we are told that a "timeout expired" at **VISA Read** before the requested operation (i.e., take 100 data samples) could be completed.



-1073807339

0x BFFF0015

VISA Read in Simple VISA Query (Long Delay).vi

Error -1073807339 occurred at VISA Read in Simple VISA Query (Long Delay).vi

Possible reason(s):

VISA: (Hex 0xBFFF0015) Timeout expired before operation completed.

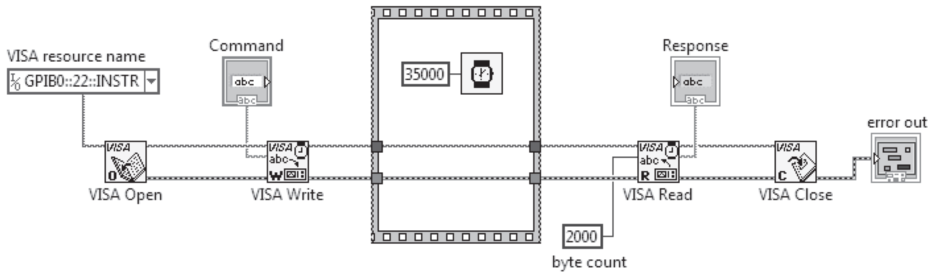
After some head scratching and checking of the Agilent 34401A user manual, the following explanation then emerges for the error we observed when running **Simple VISA Query (Long Delay)**. Simply stated, voltage sampling takes time. In its default configuration, it takes the Agilent 34401A multimeter 10 power line cycles (PLC) for each voltage sample. Additionally, the Agilent 34401A has an autozero feature, which is enabled by default. This feature operates as follows: After each voltage measurement, the multimeter internally disconnects the input signal and takes a zero reading. The instrument then subtracts the zero reading from the preceding measured value to prevent offset voltages in the multimeter's internal circuitry from affecting measurement accuracy. Since the zero reading also takes 10 PLC, each complete voltage sample by the multimeter takes 20 PLC. Assuming this instrument is plugged into a 60 Hz power source (that is, 60 PLC per second), 100 voltage samples will take about

$$100 \times \left(\frac{20\ \text{PLC}}{60\ \text{PLC}/\textbf{sec}} \right) \times 33.3\ \textbf{sec}$$

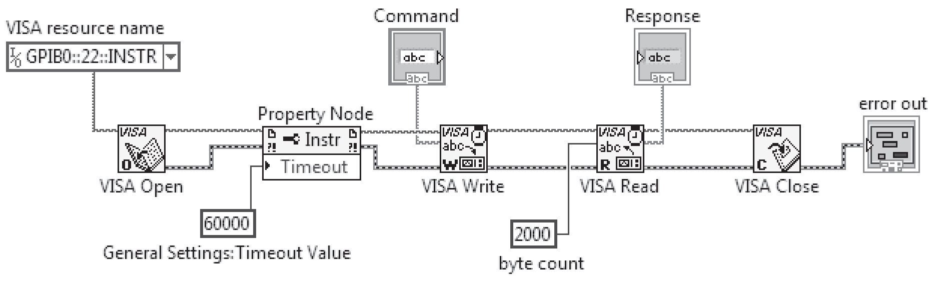
There's the problem! A few moments ago, we found that the default timeout value for **VISA Read** is 3 seconds, but the measurement we have initiated takes over 30 seconds.

Thus, long before the requested data is available, **VISA Read** terminates the execution of **Simple VISA Query (Long Delay)**.

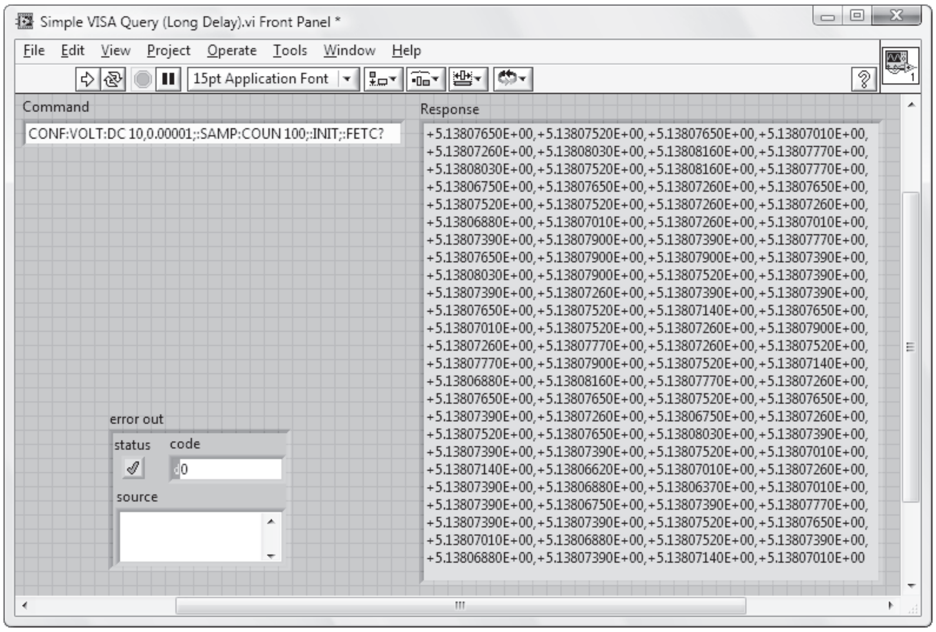
There are a couple of crude solutions to this dilemma. First, on the block diagram of **Simple VISA Query (Long Delay)**, you can insert a single-frame **Sequence Structure** into the VISA execution chain that simply contains a **Wait (ms)** icon, wired to produce a delay of about 33 seconds between the issuance of the data-taking command and the order to read the gathered data samples. The resulting diagram would appear as follows.



Second, for a slightly more elegant fix, you can use a **Property Node** to change the **Timeout** value for **VISA Read** from its default value (on my system) of 3000 ms to something larger than 33 seconds. Use this approach to modify the block diagram of **Simple VISA Query (Long Delay)** as shown below. Here, the **Timeout** value is chosen to be 60 seconds.



Return to the front panel of **Simple VISA Query (Long Delay)**. With the command to perform 100 samples programmed into **Command**, run the VI. About 33 seconds later you should see something like the following front panel. Note that the delimiter used by the Agilent 34401A to separate neighboring data values is a comma.


Simple VISA Query (Long Delay).vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font
Command
Response
error out
status
code
source

```
CONF:VOLT:DC 10,0.00001;:SAMP:COUN 100;:INIT;:FETC?
```

```
+5.13807650E+00,+5.13807520E+00,+5.13807650E+00,+5.13807010E+00,
+5.13807260E+00,+5.13808030E+00,+5.13808160E+00,+5.13807770E+00,
+5.13808030E+00,+5.13807520E+00,+5.13808160E+00,+5.13807770E+00,
+5.13806750E+00,+5.13807650E+00,+5.13807260E+00,+5.13807650E+00,
+5.13807520E+00,+5.13807520E+00,+5.13807260E+00,+5.13807260E+00,
+5.13806880E+00,+5.13807010E+00,+5.13807260E+00,+5.13807010E+00,
+5.13807390E+00,+5.13807900E+00,+5.13807390E+00,+5.13807770E+00,
+5.13807650E+00,+5.13807900E+00,+5.13807900E+00,+5.13807390E+00,
+5.13808030E+00,+5.13807900E+00,+5.13807520E+00,+5.13807390E+00,
+5.13807390E+00,+5.13807260E+00,+5.13807390E+00,+5.13807390E+00,
+5.13807650E+00,+5.13807520E+00,+5.13807140E+00,+5.13807650E+00,
+5.13807010E+00,+5.13807520E+00,+5.13807260E+00,+5.13807900E+00,
+5.13807260E+00,+5.13807770E+00,+5.13807260E+00,+5.13807520E+00,
+5.13807770E+00,+5.13807900E+00,+5.13807520E+00,+5.13807140E+00,
+5.13806880E+00,+5.13808160E+00,+5.13807770E+00,+5.13807260E+00,
+5.13807650E+00,+5.13807650E+00,+5.13807520E+00,+5.13807650E+00,
+5.13807390E+00,+5.13807260E+00,+5.13806750E+00,+5.13807260E+00,
+5.13807520E+00,+5.13807650E+00,+5.13808030E+00,+5.13807390E+00,
+5.13807390E+00,+5.13807390E+00,+5.13807520E+00,+5.13807010E+00,
+5.13807140E+00,+5.13806620E+00,+5.13807010E+00,+5.13807260E+00,
+5.13807390E+00,+5.13806880E+00,+5.13806370E+00,+5.13807010E+00,
+5.13807390E+00,+5.13806750E+00,+5.13807390E+00,+5.13807770E+00,
+5.13807390E+00,+5.13807390E+00,+5.13807520E+00,+5.13807650E+00,
+5.13807010E+00,+5.13806880E+00,+5.13807520E+00,+5.13807390E+00,
+5.13806880E+00,+5.13807390E+00,+5.13807140E+00,+5.13807010E+00
```

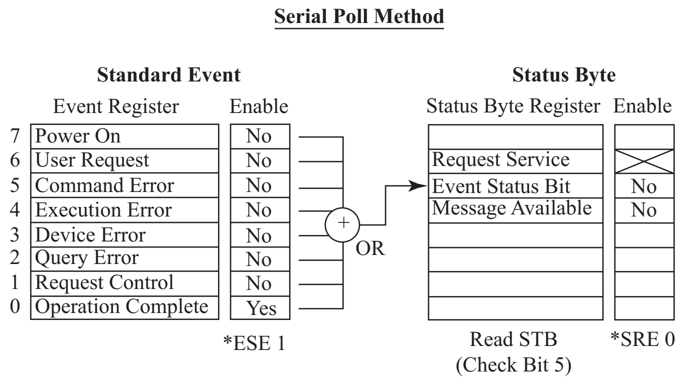
status: ✓ code: 0

In the preceding example, we found that with a detailed knowledge of the measurement process being implemented, it was possible to troubleshoot a malfunctioning VISA-based VI. Please note that lack of communication (in particular, the GPIB device not correctly knowing when the instrument's data will be available) is the root problem that led to the malfunction.

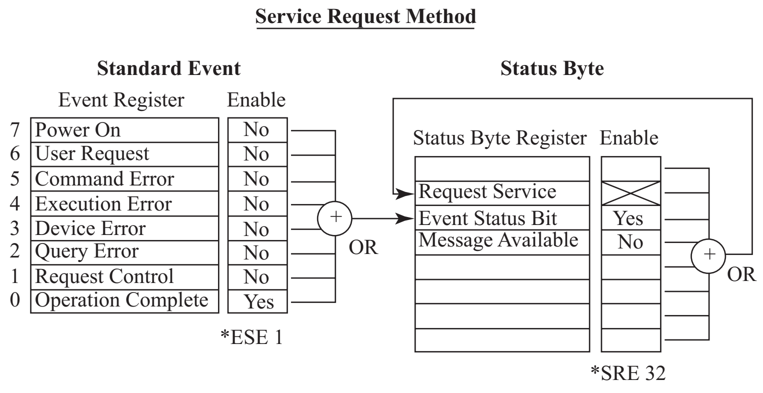
Fortunately, powerful tools exist that allow one to monitor the status of tasks being performed by a programmable instrument. For IEEE 488.2 compliant instruments, these tools are the Standard Event Status Register (SESR) and Status Byte Register (SBR) that were discussed at the beginning of this chapter. With proper use of the SESR and SBR, many potential data-taking glitches, such as the one just experienced, can be avoided.

The status reporting capabilities of the SESR and SBR can be employed in several ways. We will explore two commonly used techniques—the Serial Poll and Service Request Methods. The core operation for both of these methods is the same—the completion of an assigned task triggers the Operation Complete (OPC) bit in the Standard Event Status Register to be set, which in turn sets the Event Status Bit (ESB) of the Status Byte Register.

In the Serial Poll Method, the setting of ESB is detected by directly checking the Status Byte Register, whose state is obtained by serial polling the instrument. The complete step-by-step process of this method is shown in the following diagram.



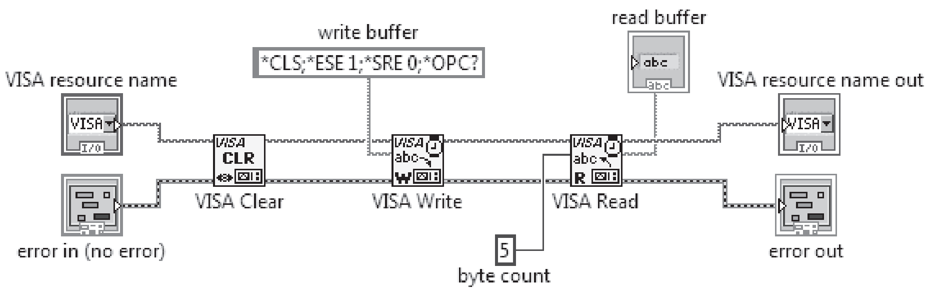
In the Service Request Method, the Status Byte Register is configured such that, when its ESB is set, the Request Service bit is induced to be set also. This action then causes the instrument to assert a SRQ, which alerts the GPIB device that the assigned operation is complete. This method is pictured here.



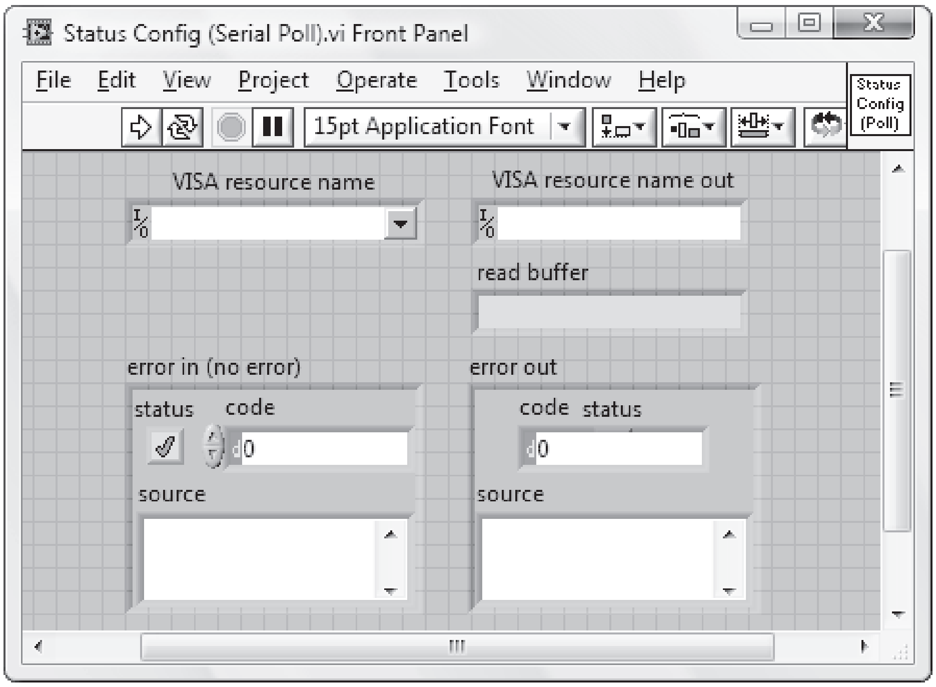
We'll write VIs that implement both of these approaches to status reporting.

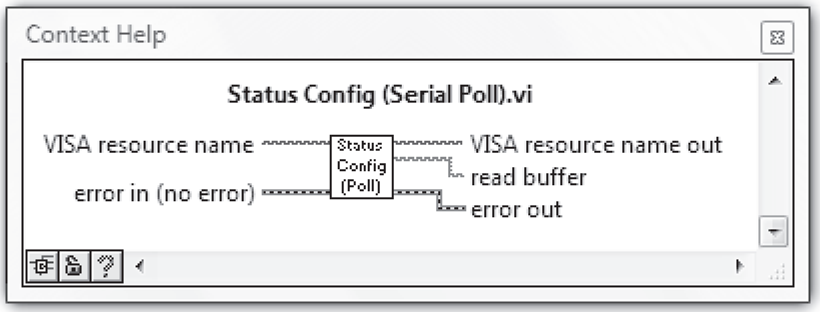
13.13 MEASUREMENT VI BASED ON THE SERIAL POLL METHOD

Let's try the Serial Poll Method first. To configure the Agilent 34401A for status reporting using the Serial Poll Method, write the following VI called **Status Config (Serial Poll)** and save it in **YourName\Chapter 13**. First, code the VI's block diagram as shown next. Use the autocreation feature in pop-up menus to create all of the constants, controls, and indicators.

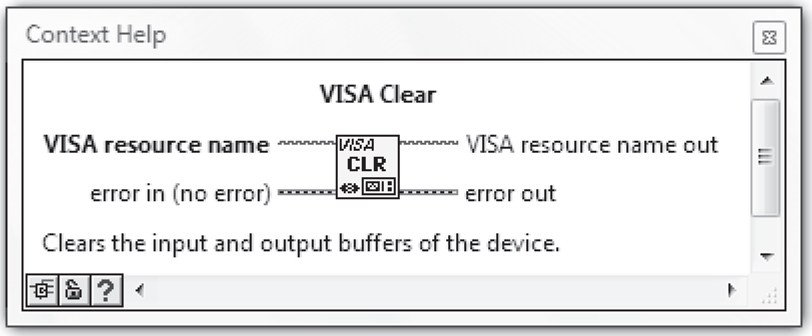


Switch to the front panel and arrange the object logically. Design an icon and assign the connector terminals consistent with the Help Window shown.


Status Config (Serial Poll).vi Front Panel



Here's how the VI works, assuming that the instrument referenced by **VISA resource name** is the Agilent 34401A multimeter. Within the chain of VISA icons, **VISA Clear** (found in **Functions>>Instrument I/O>>GPIB**) executes first. The Help Window for this icon is shown below.



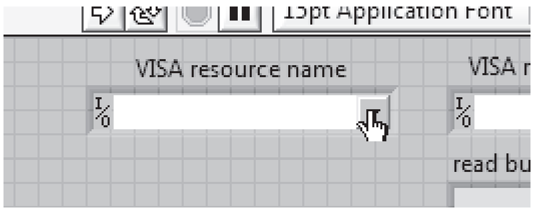
Clears the input and output buffers of the device.

Although not an absolute necessity for inclusion in **Status Config (Serial Poll)**, this VI performs the precautionary action of "clearing" the Agilent 34401A. **VISA Clear** instructs the multimeter to abort all measurements in progress, disable its triggering circuitry, clear its interface-related output buffer, and prepare to accept a new command string.

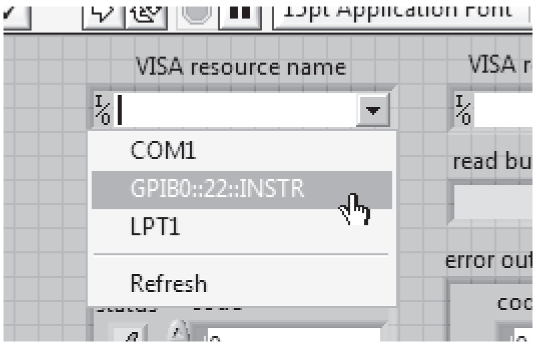
Next, **VISA Write** sends the concatenated command string **CLS;*ESE 1;*SRE 0;*OPC?* to configure the Agilent 34401A for status reporting using the Serial Poll Method. Note that since the component strings are all IEEE 488.2 common commands, leading colons are not required in the concatenation. In this sequence of commons, **CLS* clears the contents of the SESR and SBR. As described in the beginning of this chapter, **ESE 1* enables the SESR's OPC bit to set the ESB in the Status Byte Register and **SRE 0* disables the instrument from asserting a SRQ. Then, **OPC?* requests the instrument to return a *"1"* to the instrument's output buffer after this command is completed. This last command is included simply as a method of checking that the entire sequence of commands has been executed.

Finally, **VISA Read** reads the contents of the instrument's output buffer. If all goes well, there should be a single ASCII character *1* read into the computer.

Test drive your VI as follows. Click on **VISA resource name** control's menu button with the .

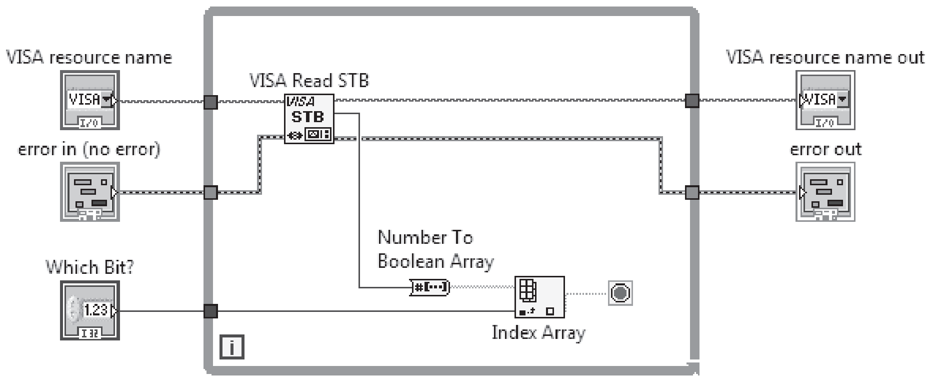


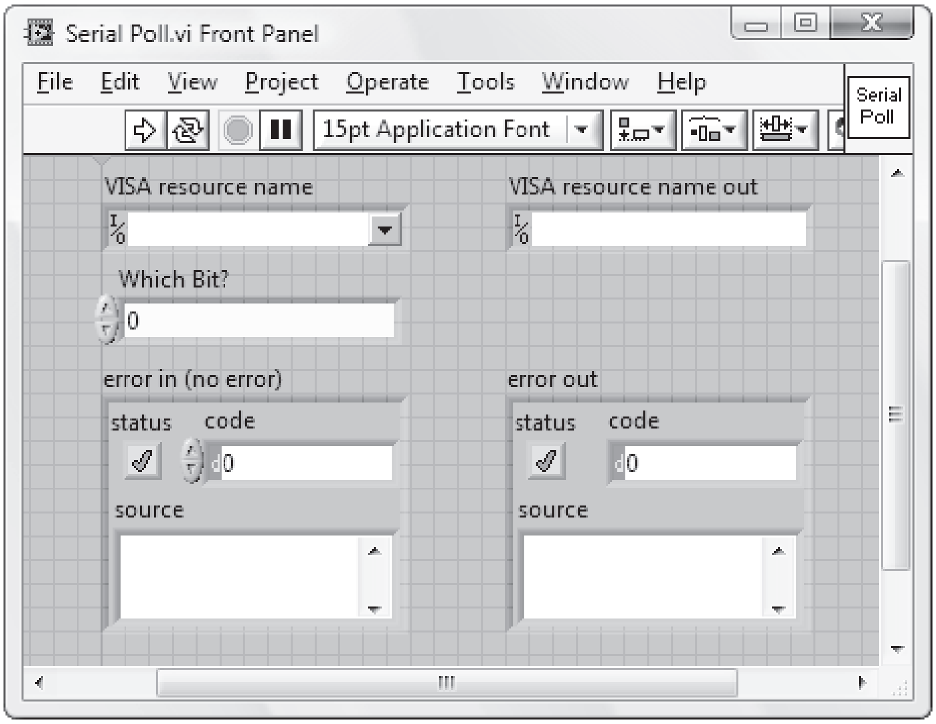
Then, from the list presented, select the VISA resource name for your computer-controlled instrument.

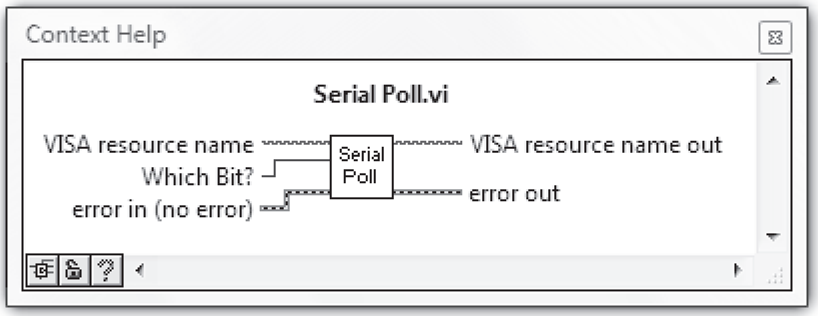


Then, run **Status Config (Serial Poll)**. Upon completion, does the **Buffer Reading** string indicator display an ASCII character *1*?

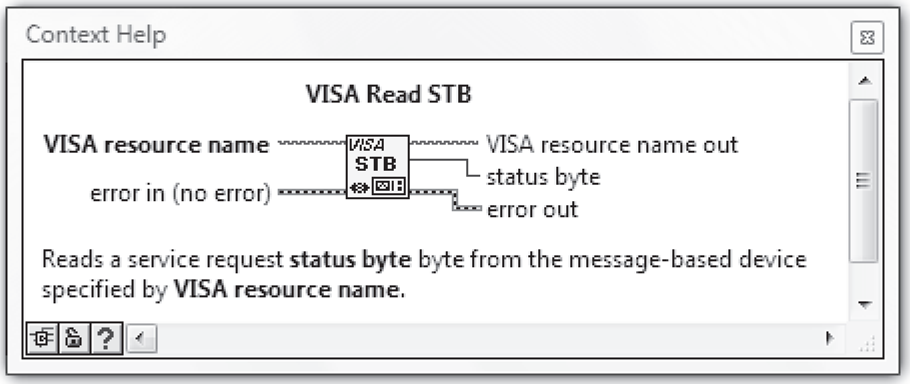
Next, construct a VI called **Serial Poll** which continuously reads the Status Byte Register of an instrument until a given bit is set. A suggested coding of **Serial Poll** is shown in the following diagrams, and explanations of the unfamiliar icons are in the subsequent paragraphs. Save **Serial Poll** in **YourName\Chapter 13**.





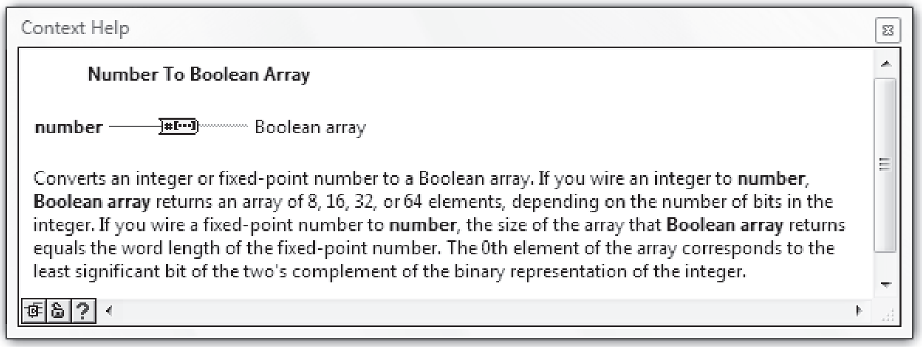


VISA Read STB, found in **Functions>>Instrument I/O>>VISA**, is the workhorse of this VI. With each iteration of the While Loop, its **status byte** output returns the current values of the SBR's eight bits in the form of an integer. For example, if the SBR's fifth bit (ESB) is set, then **status byte** outputs the integer *32*, since $00100000_2 = 32_{10}$. The Help Window for **VISA Read STB** is shown next.



Reads a service request **status byte** byte from the message-based device specified by **VISA resource name.**

The individual bits of **status byte** can be checked through the use of **Number To Boolean Array** (found in **Functions>>Programming>>Boolean** with its Help Window shown next). This VI creates an array of TRUE and FALSE values that mirror the sequence of zeros and ones (starting from the least-significant bit) in the binary representation of the integer input **number**. For example, if **number** equals the decimal integer *48*, then the **Boolean array** output will be [F, F, F, F, T, T, F, F], since 4810 = 00110000_2. **Index Array** can then be used to ascertain the value of a particular element in this array. **Serial Poll**'s While Loop will continue to iterate until **Which Bit?** becomes TRUE.



Converts an integer or fixed-point number to a Boolean array. If you wire an integer to **number**, **Boolean array** returns an array of 8, 16, 32, or 64 elements, depending on the number of bits in the integer. If you wire a fixed-point number to **number**, the size of the array that **Boolean array** returns equals the word length of the fixed-point number. The 0th element of the array corresponds to the least significant bit of the two's complement of the binary representation of the integer.

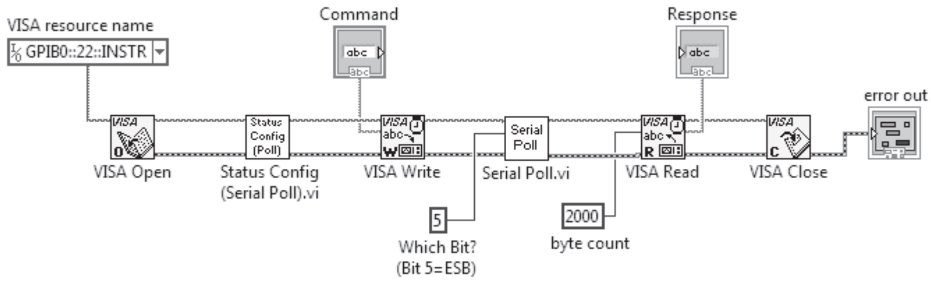
Run **Serial Poll** under **Highlight Execution** and, through your observations, gain a better understanding of its operation. Remember to input values for **VISA resource name** and **Which Bit?** on the front panel. When run in this isolated manner, the VI will most likely never be able to exit the While Loop, so you'll have to stop it using the **Abort Execution** button in the toolbar.

We're finally ready to write **VISA Query (Serial Poll)**. This top-level program implements serial polling to synchronize the GPIB activities necessary in acquiring 100 voltage samples using an Agilent 34401A multimeter.

Open **Simple VISA Query (Long Delay)**, then use **Save As…** to create **VISA Query (Serial Poll)**. The front panel can remain unchanged. If it's not already there by default, type the following command into the **Command** control.

*CONF:VOLT:DC<Space>10,0.00001;:SAMP:COUN<Space>100;:INIT;*OPC;:FETC?*

Switch to the block diagram and modify it as shown below with **Status Config (Serial Poll)** and **Serial Poll** used as subVIs.

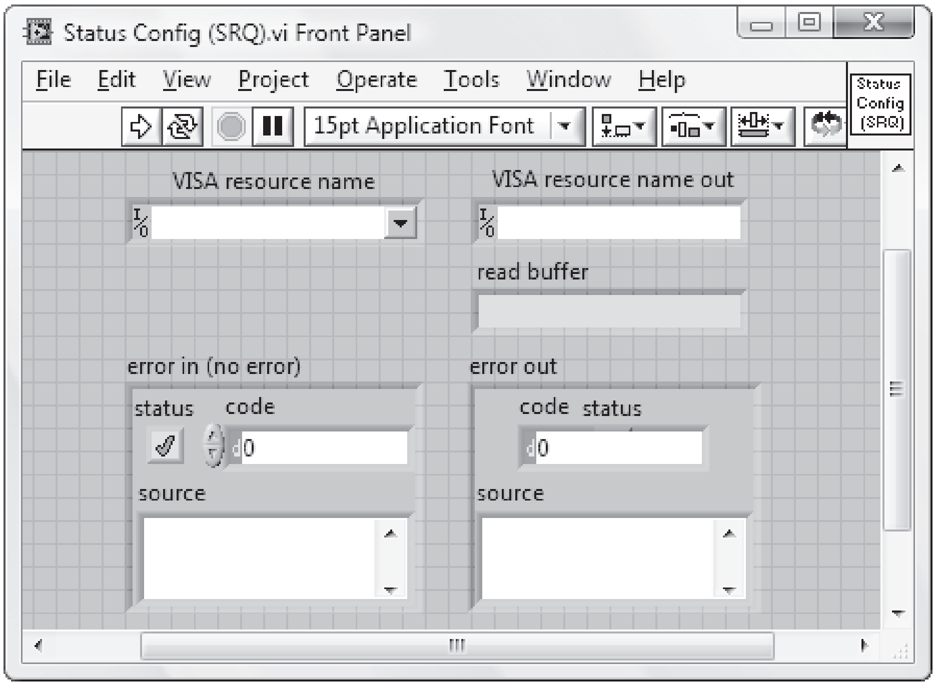


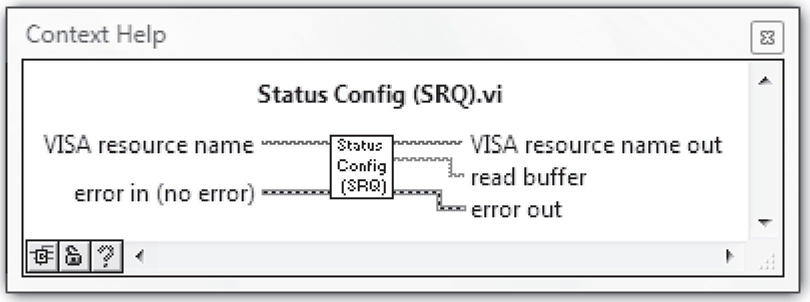
Here is how this diagram works: The concatenated command string is sent to the instrument by **VISA Write**. After configuring the multimeter for the desired DC Voltage measurement function, the acquisition process is begun by the *INIT* command. The succession of 100 samples is acquired and temporarily stored in the multimeter's internal memory. After the hundredth sample is obtained, **OPC* instructs the instrument to set its SESR's OPC bit (which, in turn, sets the SBR's ESB), then *FETC?* loads the contents of the internal memory into the instrument's output buffer. At that point, **Serial Poll** detects the setting of ESB, which then triggers the instrument's output buffer to be read by **VISA Read**. One might be tempted to write the concatenated command with **OPC* after *FETC?*, rather than sandwiched between *INIT* and *FETC?*, as above. It is best, however, to avoid sending **OPC* after a query (a query is a command like *FETC?* that ends in a question mark) as such commands cause a message to be loaded into an instrument's output buffer. If the message exceeds the finite size of the output buffer, as happens in our present situation, the query must be immediately followed by **VISA Read** as the program executes in order to read the long message string over the bus successfully.

Return to the front panel, save your work, and then run **VISA Query (Serial Poll)**. Does the VI obtain the requested 100 DC voltage samples successfully? If so, try running it again with **Highlight Execution** activated for both **VISA Query (Serial Poll)** and its subVI **Serial Poll**. This exercise will illustrate the weakness of the Serial Poll Method, namely, the large volume of interface bus traffic required by this technique. During the 30-odd seconds while the 100 data samples are being gathered, the instrument is polled countless times by the GPIB device so that its status can be continuously monitored. While effective, the Serial Poll Method is rather inefficient because of its excessive use of the interface bus and processor time.

13.14 MEASUREMENT VI BASED ON THE SERVICE REQUEST METHOD

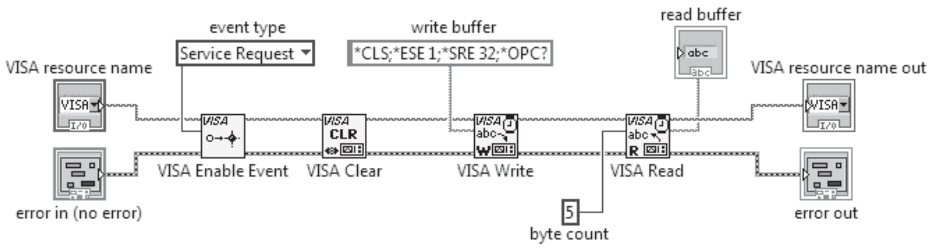
The Service Request Method provides status reporting with a minimum of interface bus activity. To configure the 34401A for status reporting using the Service Request Method, open **Status Config (Serial Poll)**, then create **Status Config (SRQ)** using **Save As…** and save it in **YourName\Chapter 13**. The front panel and terminal assignments can remain as is, but the icon should be redesigned as shown here.





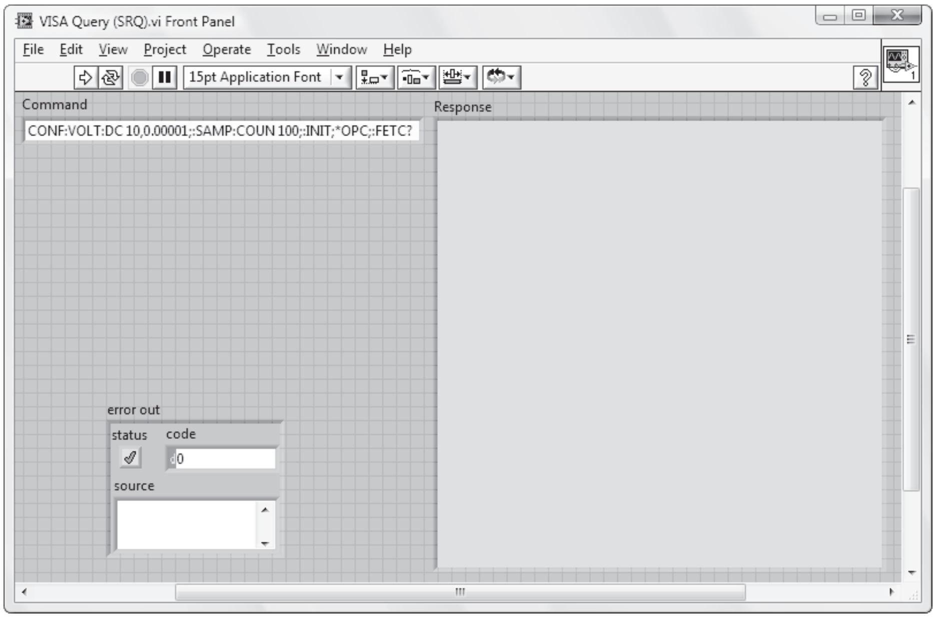
Only two modifications of the block diagram are needed. First, by changing **SRE 0* to **SRE 32* in the command string sent to the instrument, the Agilent 34401A will assert the

SRQ line when the SBR's fifth (ESB) bit is set. The already present **ESE 1* command configures the instrument to set the ESB in response to the SESR's OPC (Operation Complete) bit being set. Second, in order for VISA icons to detect service request (SRQ) events during this VISA session, **VISA Enable Event**, with **Service Request** wired to its **event type** input, must be included in the diagram as shown. **VISA Enable Event** is found in **Functions>>Instrument I/O>>VISA>>VISA Advanced>>Event Handling.**



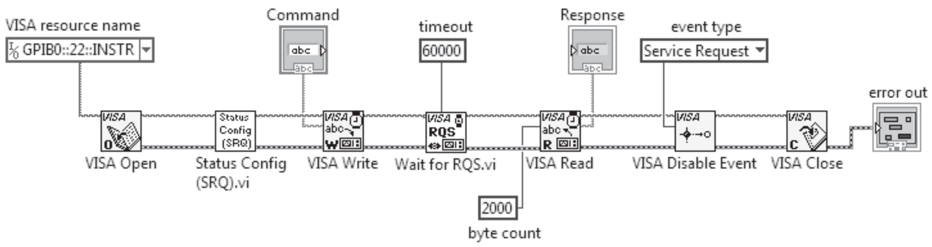
Save your work as you close this VI.

Open **VISA Query (Serial Poll)**, then use **Save As…** to create a new VI named **VISA Query (SRQ)**, and store it in **YourName\Chapter 13**. The front panel is fine as is.


VISA Query (SRQ).vi Front Panel

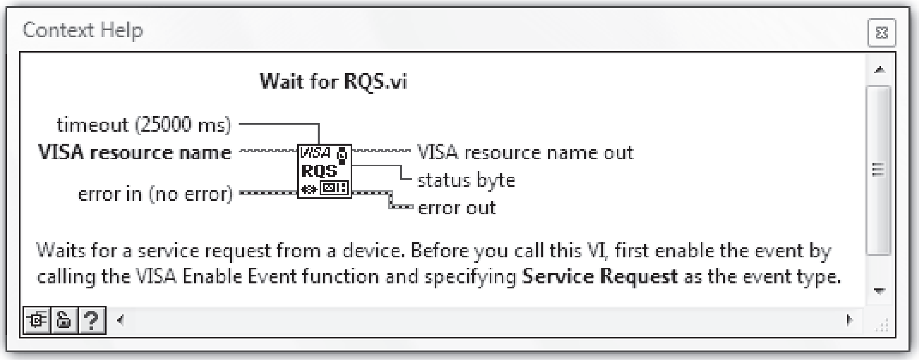
Command: CONF:VOLT:DC 10,0.00001;:SAMP:COUN 100;:INIT;*OPC;:FETC?

Switch to the block diagram and modify it as shown next.



Here, **VISA Disable Event**, found in **Functions>>Instrument I/O>>VISA>>VISA Advanced>>Event Handling**, must be included in order disable VISA servicing of SRQ events before the VISA session is closed.

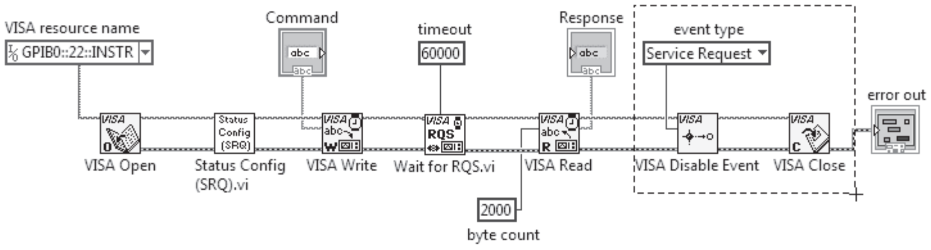
Wait for RQS.vi, also found in **Functions>>Instrument I/O>>VISA>>VISA Advanced>>Event Handling** (Help Window shown below), sits idly until the instrument denoted by **VISA resource name** asserts a SRQ. However, there is a limit to the patience of this icon. It will only wait up to a total time of **timeout**, with a default value of 25000 ms = 25 seconds. Because our measurement requires over 33 seconds, a constant larger than 33,000 must be wired to the **timeout** input of **Wait for RQS.vi**, as shown in the above diagram.



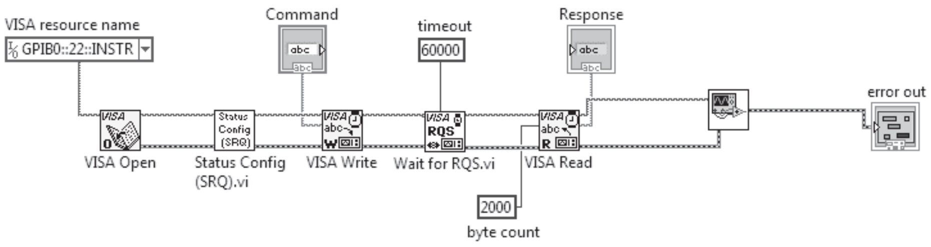
Waits for a service request from a device. Before you call this VI, first enable the event by calling the VISA Enable Event function and specifying **Service Request** as the event type.

Save your work, then run **VISA Query (SRQ)**. Does it successfully acquire the requested 100 DC voltage samples? Do you understand the operation of this program and how the Service Request Method manages to work with a minimum of interface bus activity?

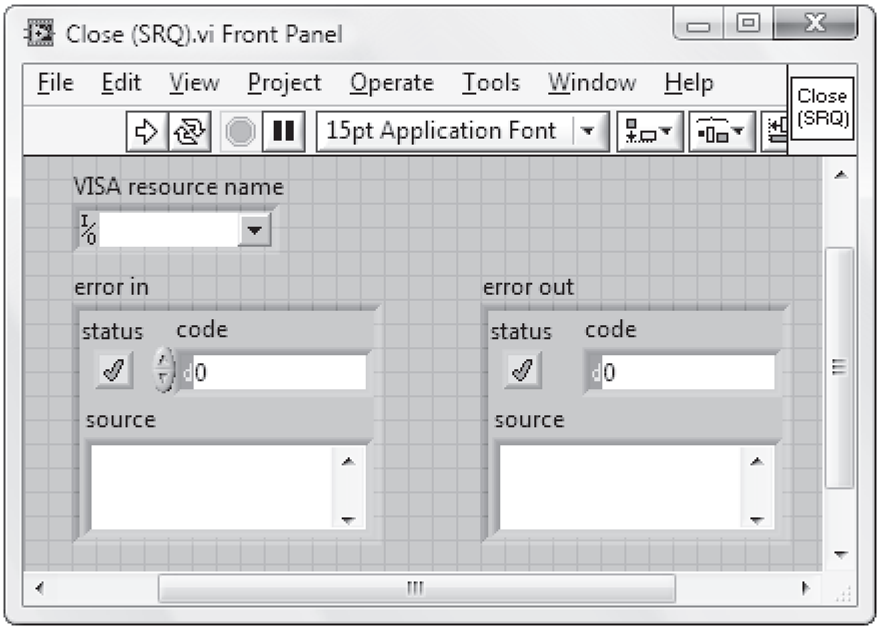
To simplify the block diagram of **VISA Query (SRQ)**, you might consider packaging **VISA Disable Event** and **VISA Close** together in a subVI called **Close (SRQ)**, since both of these icons are involved in closing down the service request–based VISA session. To accomplish this feat easily, simply create a highlighting box around the two icons using the ↖.

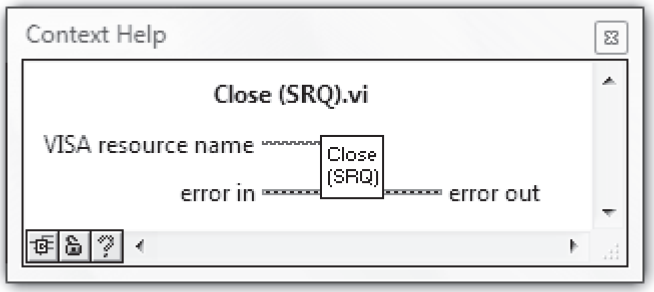


Then select **Edit>>Create subVI**. A new subVI icon will appear wired on your diagram.

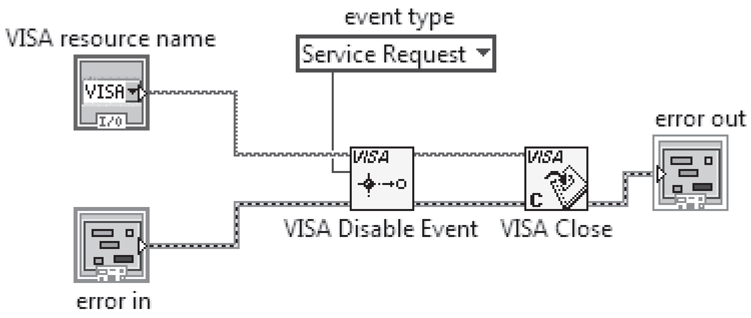


Double-click on this new icon to open it. Then, relabel the front-panel objects appropriately, design an icon, and assign the connector consistent with the Help Window shown (using the 4 × 2 × 2 × 4 pattern). Save this VI under the name **Close (SRQ)** in **YourName\Chapter 13**.

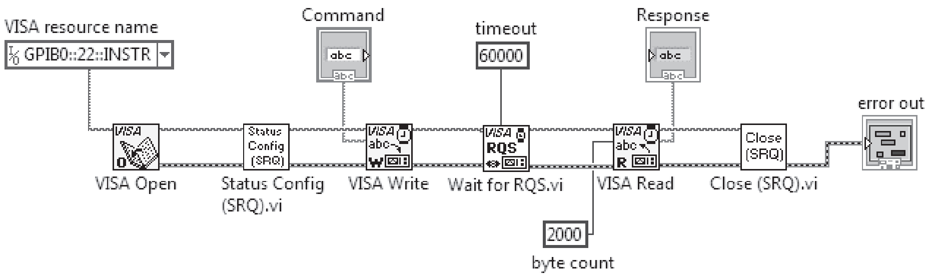




Switch to the block diagram of **Close (SRQ)**. It should appear as follows.



Close **Close (SRQ)**, and return to the block diagram of **VISA Query (SRQ)**. You may have to delete the originally created subVI and load a new copy of **Close (SRQ)** there using **Functions>>Select a VI…** After that, the finished block of **VISA Query (SRQ)** will appear as shown next. Try running this VI to verify that it functions correctly.



13.15 CREATING AN INSTRUMENT DRIVER

An instrument driver is a collection of modular software routines that perform the operations required in the computer control of a programmable instrument. These operations include configuring, triggering, status checking, and sending data to and receiving data from the instrument. Above, **Status Config (Serial Poll)** and **Status Config (SRQ)** are examples of configuration VIs that would be useful to include as part of the Agilent 34401A software driver. You will now write another configuration VI, this time one that prepares the multimeter for taking a desired measurement.

The Agilent 34401A is capable of implementing 11 types of measurement functions: DC and AC voltage, DC voltage ratio (ratio of voltage at two different inputs), DC and AC current, 2- and 4-wire resistance (2-wire is the "normal" method for measuring resistance; the more involved 4-wire technique is necessary only when measuring very small resistance samples), frequency and period of an AC signal, continuity, and diode check. To gain experience with some of the LabVIEW tools available for developing instrument drivers, let's write a driver that offers the choice of configuring the Agilent 34401A for either a DC voltage, AC voltage, or 2-wire resistance measurement. You, of course, can be more ambitious and write your VI to control up to all 11 possible measurement functions.

Referring to the Agilent 34401A user manual, we find that our driver must allow a user to select one of the following three possible commands in order to configure the instrument for the desired measurement function:

```
CONFigure:VOLTage:DC <Space> <Range>, <Resolution>
CONFigure:VOLTage:AC <Space> <Range>, <Resolution>
CONFigure:RESistance <Space> <Range>, <Resolution>
```

Here, the possible values of <*Range*> for both the DC and AC voltage measurements are 0.1, 1, 10, 100, and 1000 Volts. For the resistance measurement, the allowed <*Range*> values are 100, 1k, 10k, 100k, 1M, 10M, and 100M ohms. In all cases, the measurement precision may be 4½, 5½, or 6½ digits, which corresponds to <*Resolution*> being 10^{-4}, 10^{-5} or 10^{-6} times the <*Range*> value, respectively.

We will write two programs called **Range and Resolution Decoder** and **Command String**, which will allow a user to construct the desired command string using front-panel controls. On **Range and Resolution Decoder**, given range and resolution choices from a user-friendly front-panel listing of the multimeter's available offerings, the program will convert these choices to the double-precision floating-point numeric format needed in **Command String**. **Command String** will construct the appropriate ASCII command string to be sent to the Agilent 34401A, based on selections made on its front-panel controls.

Create a new VI named **Range and Resolution Decoder** and save it in **YourName\Chapter 13**. Place four **Emun** controls (found in **Controls>>Modern>>Ring & Emun**) on the front panel and label them **Function, Voltage Range**, **Resistance Range**, and **Resolution**, respectively. Pop up each **Enum**, select **Edit Items…** and then program each with the items given in the following list.

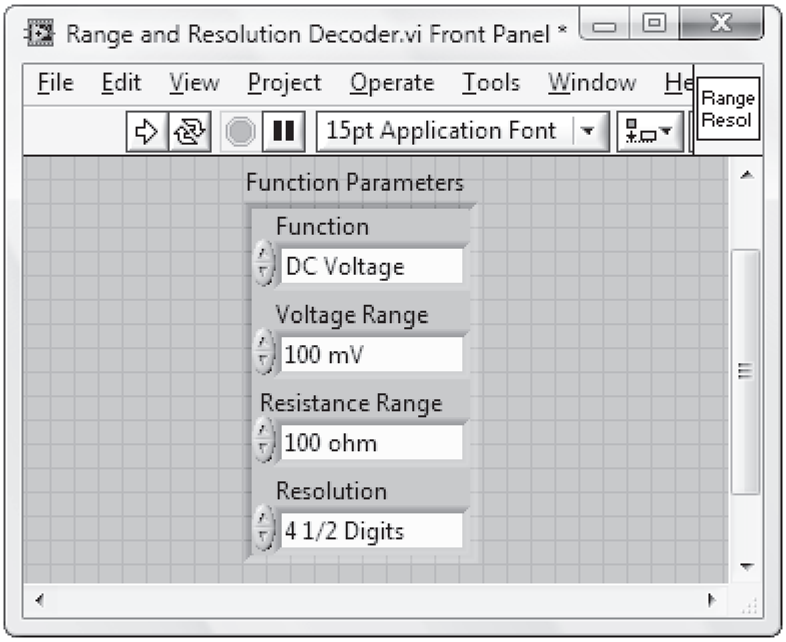
Function: *DC Voltage*, *AC Voltage*, *Resistance*

Voltage Range: *100 mV*, *1 V*, *10 V*, *100 V*, *1000 V*

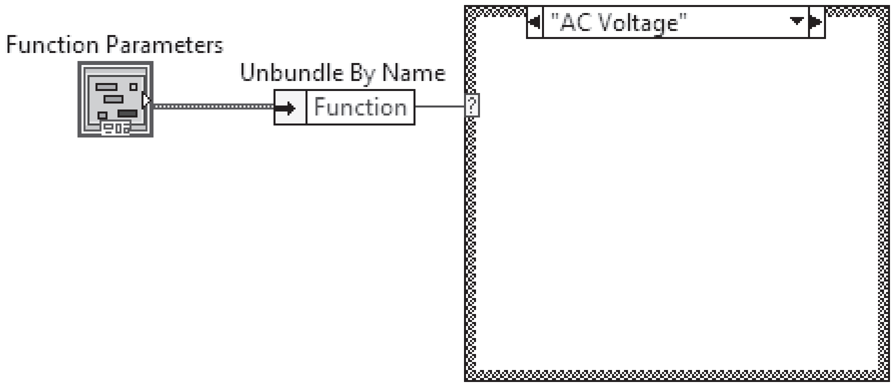
Resistance Range: *100 ohm*, *1 kohm*, *10 kohm*, *100 kohm*, *1 Mohm, 10 Mohm*, *100 Mohm*

Resolution: *4 1/2 Digits*, *5 1/2 Digits*, *6 1/2 Digits*

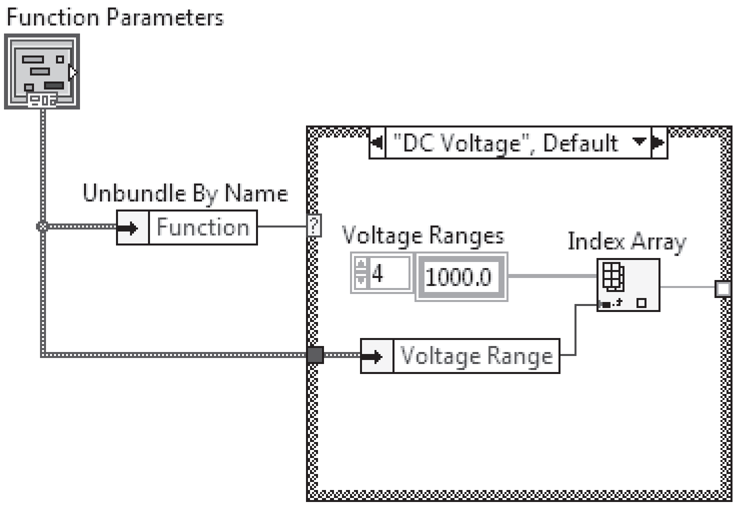
Then place these four **Enum** controls in a **Cluster** shell (found in **Controls>>Modern>>Array, Matrix & Cluster**) labeled **Function Parameters** as shown next.



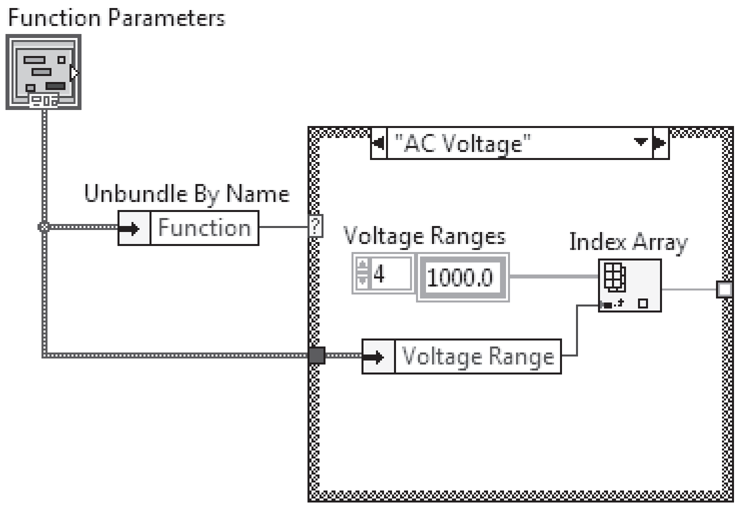
Switch to the block diagram, place a **Case Structure** there, and complete the code shown. Pop up on the Case Structure and select **Add Case for Every Value**, and then verify that it has three cases labeled **DC Voltage**, **AC Voltage**, and **Resistance**.



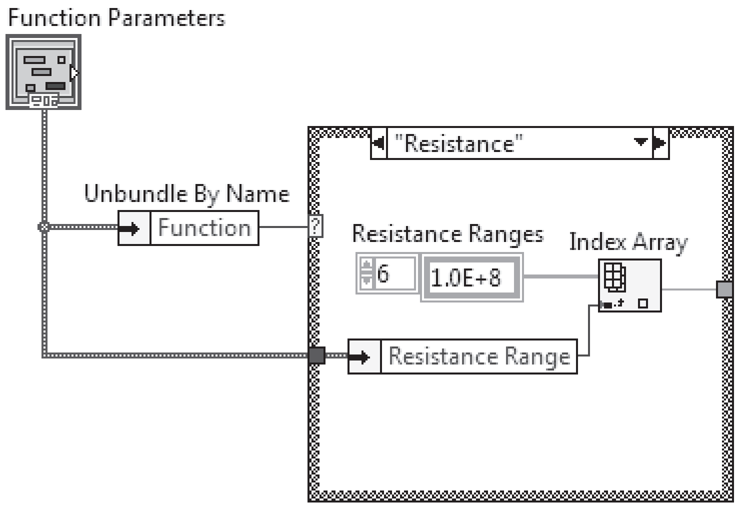
Select the **DC Voltage** case, and then place an **Index Array** icon within it. Pop up on Index Array's **n-dimension array** input and select **Create>>Constant** to create an **Array Constant** and label it **Voltage Ranges**. Next, program the index-0 through index-4 elements of this **Array Constant** as *0.1*, *1.0*, *10.0*, *100.0*, and *1000.0*, respectively. The **Array Constant** then will serve as a *look-up table* of the multimeter's allowed voltage ranges, given as double-precision floating-point numbers. Complete the code for the **DC Voltage** case shown below. Here, the integer associated with a selected **Voltage Range** on the front-panel Enum control provides the index of the desired look-up table element. This element is then output by **Index Array**.



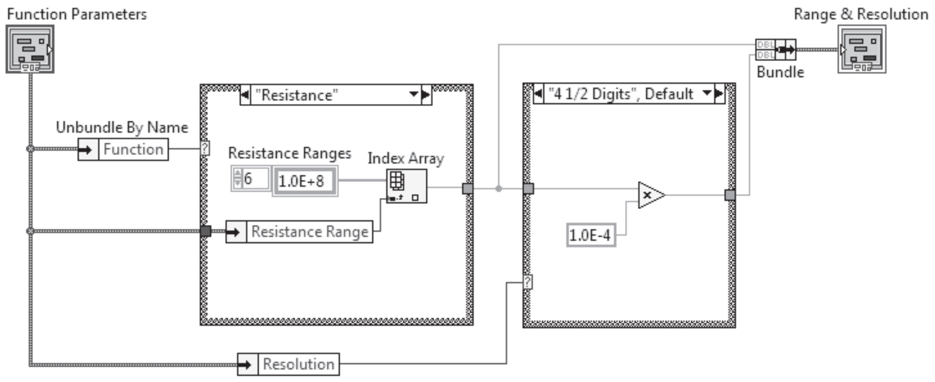
Clone the **Voltage Ranges** Array Constant (mouse-click, while holding down <*Ctrl*>), and place the copy somewhere on the block diagram. Then switch to the **AC Voltage** case and (using your cloned **Voltage Ranges**), write the code shown.

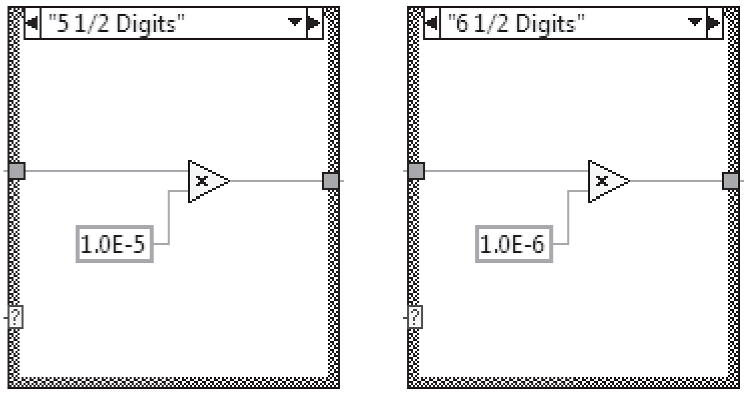


Finally, switch to the **Resistance** case and program it as shown. Here, the index-0 through index-6 elements of the **Resistance Ranges** Array Constant are *1.0E2*, *1.0E3*, *1.0E4*, *1.0E5*, *1.0E6*, *1.0E7*, *1.0E8*, respectively.

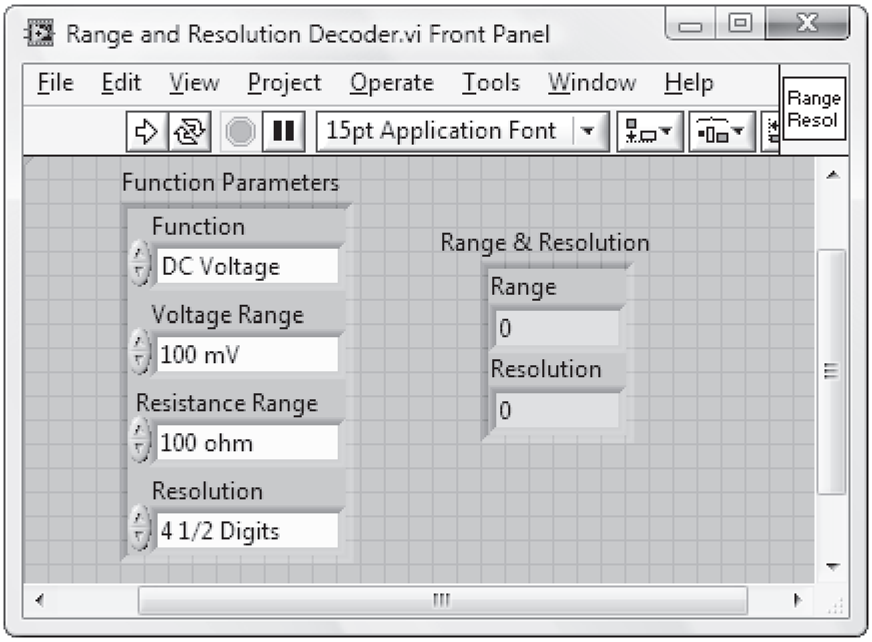


Add a second **Case Structure** and complete the diagram as shown next. Remember to pop up on the Case Structure and select **Add Case for Every Value**. The **Range & Resolution** indicator cluster is created by popping up on **Bundle** and using **Create>>Indicator**.



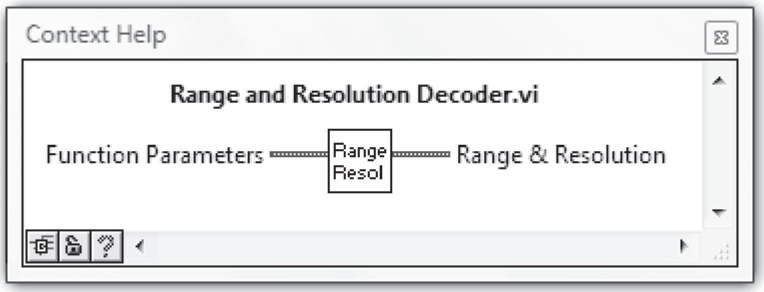


Return to the front panel. Within the **Range & Resolution** indicator cluster, label the top and bottom **Numeric Indicator** as **Range** and **Resolution**, respectively. Design an icon and assign the connector terminals as shown. Save your work.


Range and Resolution Decoder.vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font
Range Resol

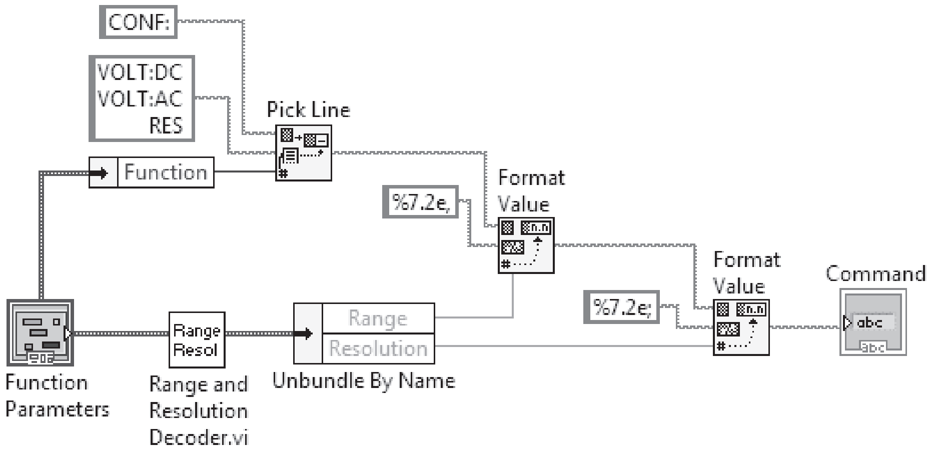
Function Parameters
Function: DC Voltage
Voltage Range: 100 mV
Resistance Range: 100 ohm
Resolution: 4 1/2 Digits

Range & Resolution
Range: 0
Resolution: 0

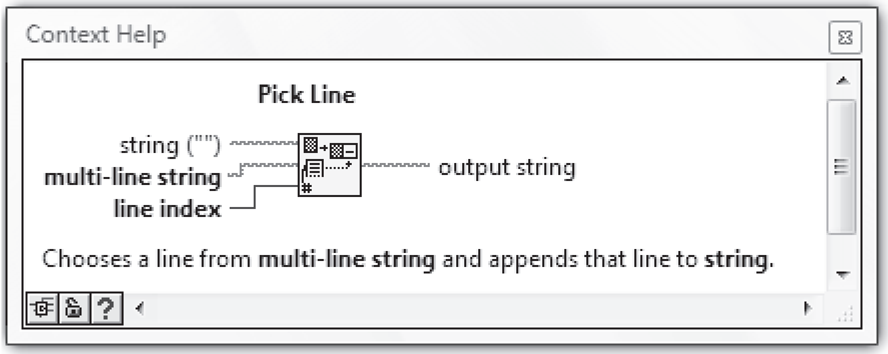


Run **Range and Resolution Decoder** and verify that it functions properly. For example, with **Function**, **Voltage Range**, and **Resolution** equal to *DC Voltage*, *10 V*, and *6 1/2 Digits*, respectively, **Range** and **Resolution** should equal *10.0* and *0.00001*.

Next, open a blank VI, and save it under the name **Command String** in **YourName\Chapter 13**. Switch to the block diagram and write the following code, which constructs the desired ASCII command string. The **Function Parameters** control cluster and **output string** string indicator are made using the autocreation feature in pop-up menus.

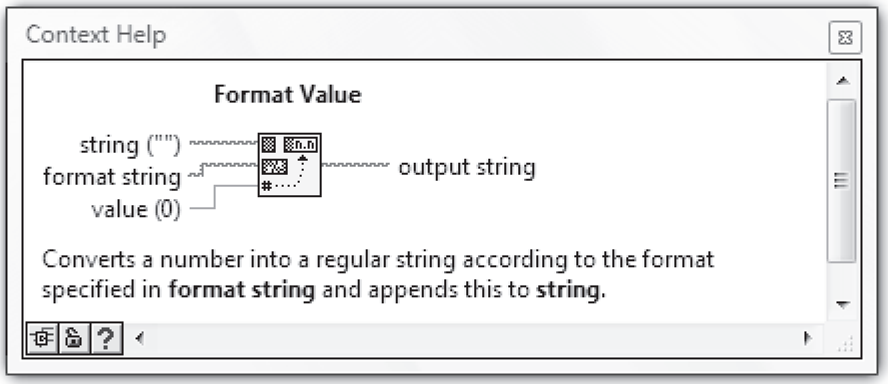


This diagram constructs the desired command string in a three-step process. First, all three possible commands begin with the keyword *CONF:*, so this sequence of ASCII characters is wired to the **string** input of **Pick Line** (found in **Functions>>Programming>> String>>Additional String Functions** with its Help Window given below). The value of the **line index** input (an integer given by the front-panel **Function** Enum control) then selects which of the three possible lines programmed into the **String Constant** wired to **multi-line string** is to be appended to ***CONF***:. Create the three lines in this **String Constant** by the following sequence of keystrokes: *VOLT:DC<Space> <Enter>VOLT:AC<Space> <Enter>RES<Space>*. Be sure to include the *<Space>* character at the end of each command string. You can make the invisible space and line feed characters visible by popping up on the **String Constant** and selecting **"\" Codes Display**. The correct entry will then appear as *VOLT:DC\s\nVOLT:AC\s\nRES\s*.



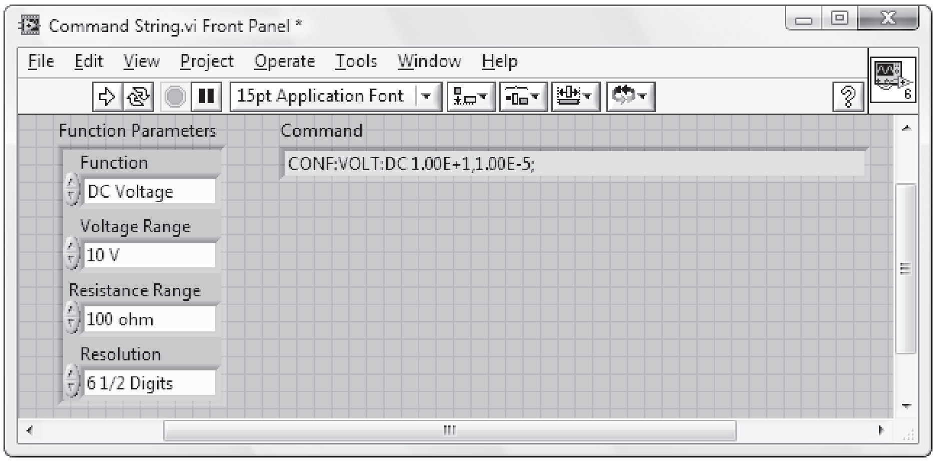
Chooses a line from **multi-line string** and appends that line to **string.**

Format Value, from **Functions>>Programming>>String/Number Conversion** (Help Window shown next), then is used to attach two more string fragments, each with embedded ASCII-coded numbers that program the <*Range*> and <*Resolution*> settings of the multimeter. This icon takes the number at the **value** input, and converts it to an ASCII string representation with the format defined at the **format string** input. This ASCII string is appended to **string** and presented at **output string**. In the above diagram, the scientific notation format *%7.2e* (see section 5.5) is used for both <*Range*> and <*Resolution*> parameters. Note a comma (,) and semicolon (;) follow <*Range*> and <*Resolution*>, respectively.



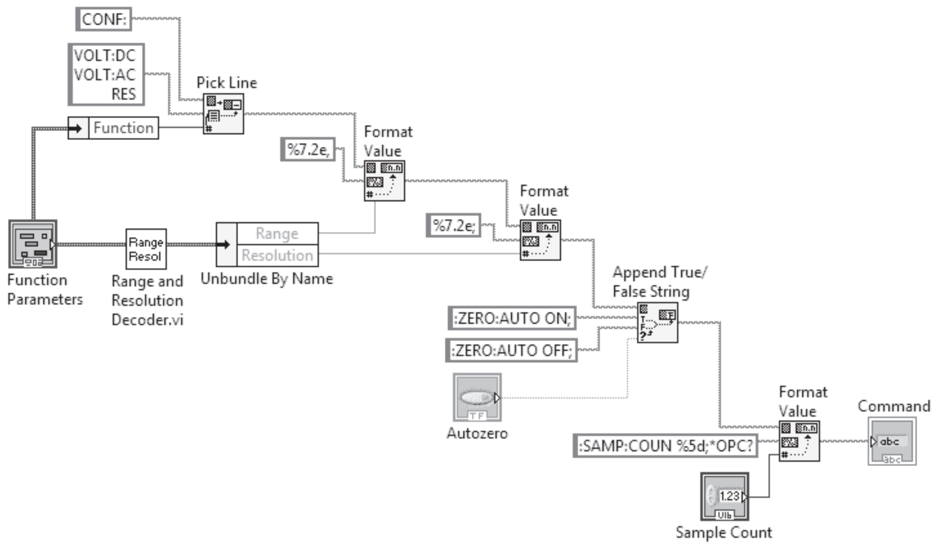
Converts a number into a regular string according to the format specified in **format string** and appends this to **string**.

Switch to the front panel and change the label of the String Indicator from **output string** to **Command**. Run the VI with a given choice of the controls within **Function Parameters**, and verify that the correct command string appears in the **Command** indicator. Save your work.


Command String.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font
Function Parameters
Function: DC Voltage
Voltage Range: 10 V
Resistance Range: 100 ohm
Resolution: 6 1/2 Digits
Command: CONF:VOLT:DC 1.00E+1,1.00E-5;

Add a **Push Button** (found in **Controls>>Modern>>Boolean**) and a **Numeric Control** to the front panel and label them **Autozero** and **Sample Count**, respectively. Change the representation of **Sample Count** to **U16**.

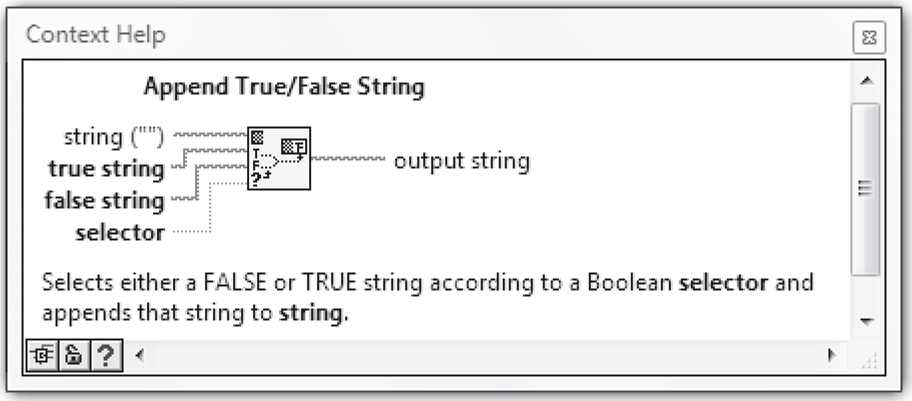
Switch to the block diagram, then include code to control the multimeter's autozeroing feature and to program the desired number of data samples to be taken. The *%5d* format in the *SAMPle:COUNt* command specifies a five-place decimal integer because the maximum allowed value for *SAMPLe:COUNt* (according to the Agilent 34401A user manual) is 50000. The format string entry for this command should be *:SAMP:COUN<Space>%5d*.



Autozero can be either turned on and off with the following commands:

ZERO:AUTO<Space>ON
ZERO:AUTO<Space>OFF

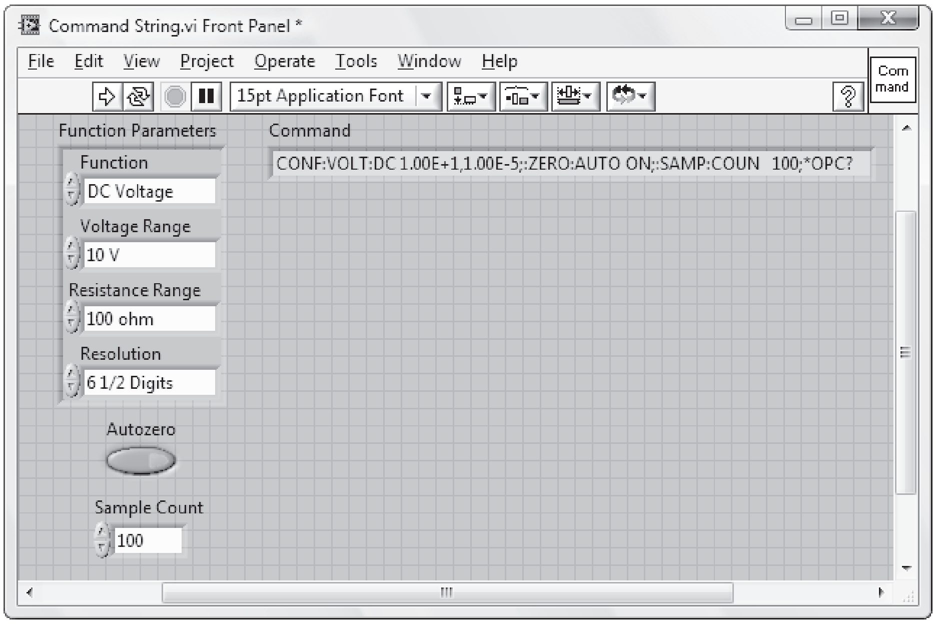
Append True/False String (found in **Functions>>Programming>>String>>Additional String Functions**), whose Help Window follows, provides an easy way to select which of these two choices is concatenated to the command string. Remember to include the leading colon and final semicolon in the **false string** and **true string** entries to assure proper command concatenation.



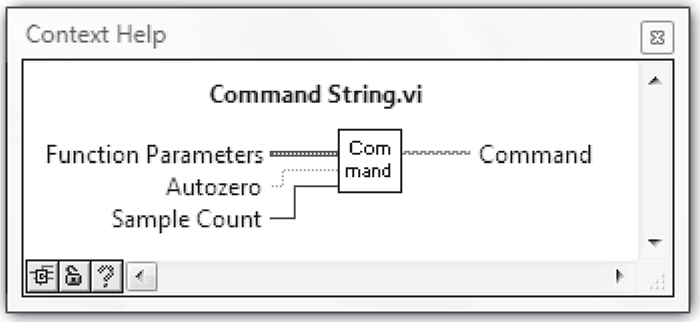
Selects either a FALSE or TRUE string according to a Boolean **selector** and appends that string to **string.**

To guarantee that the instrument fully processes the sent command string before exiting the configuration VI (which you will write in a moment), conclude the string with **OPC?*.

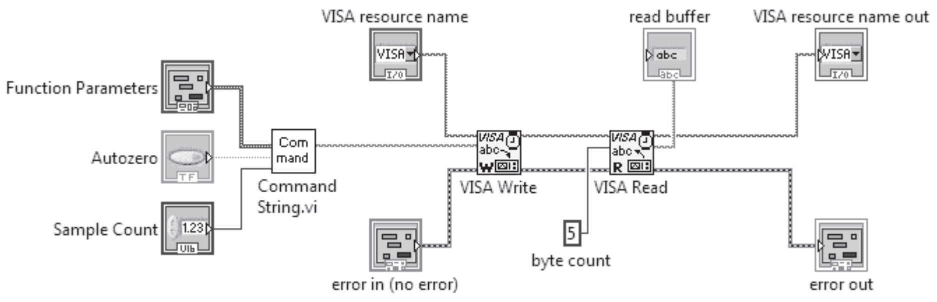
Return to the front panel. Run the VI with a given choice of the front-panel controls to verify that the correct command string appears in the **Command** indicator. Then design an icon and assign the connectors consistent with the following Help Window. Save your work as you close the VI.


Command String.vi Front Panel *
File Edit View Project Operate Tools Window Help
15pt Application Font
Com mand

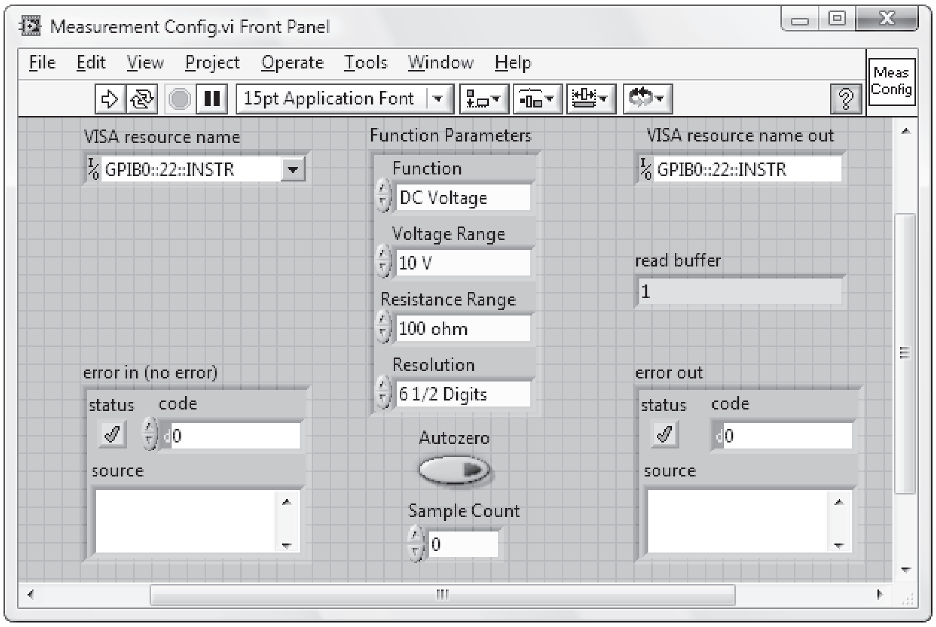
Function Parameters
Function: DC Voltage
Voltage Range: 10 V
Resistance Range: 100 ohm
Resolution: 6 1/2 Digits
Autozero
Sample Count: 100
Command: CONF:VOLT:DC 1.00E+1,1.00E-5;:ZERO:AUTO ON;:SAMP:COUN 100;*OPC?



Finally, create a VI named **Measurement Config** and save it in **YourName\Chapter 13**. Switch to the block diagram and code it as shown. This VI will write the command string to the instrument. When this diagram runs, the **read buffer** indicator will display an ASCII "1" if the command string was successfully read by the instrument.



Switch to the front panel, arrange the objects there as you wish. Then design an icon and assign the connector terminals consistent with the Help Window shown below. Save your work.


Measurement Config.vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font
Meas Config

VISA resource name: GPIB0::22::INSTR

Function Parameters
- Function: DC Voltage
- Voltage Range: 10 V
- Resistance Range: 100 ohm
- Resolution: 6 1/2 Digits

Autozero

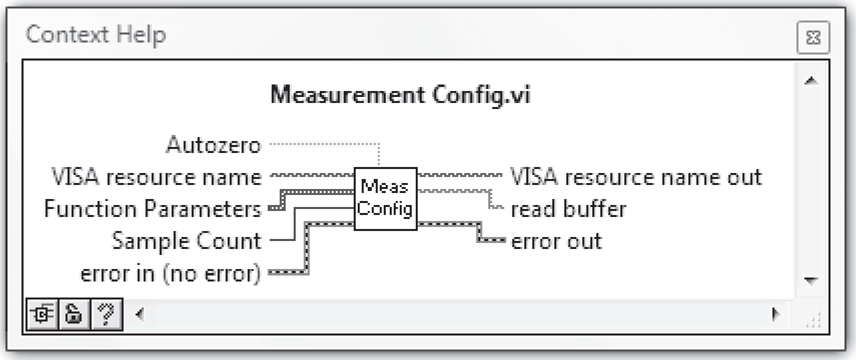
Sample Count: 0

VISA resource name out: GPIB0::22::INSTR

read buffer: 1

error in (no error): status ✓ code 0 source

error out: status ✓ code 0 source



Input the VISA resource name for your instrument into the **VISA resource name** control, and then run **Measurement Config** with front-panel control settings shown above so that the following command is sent to the instrument:

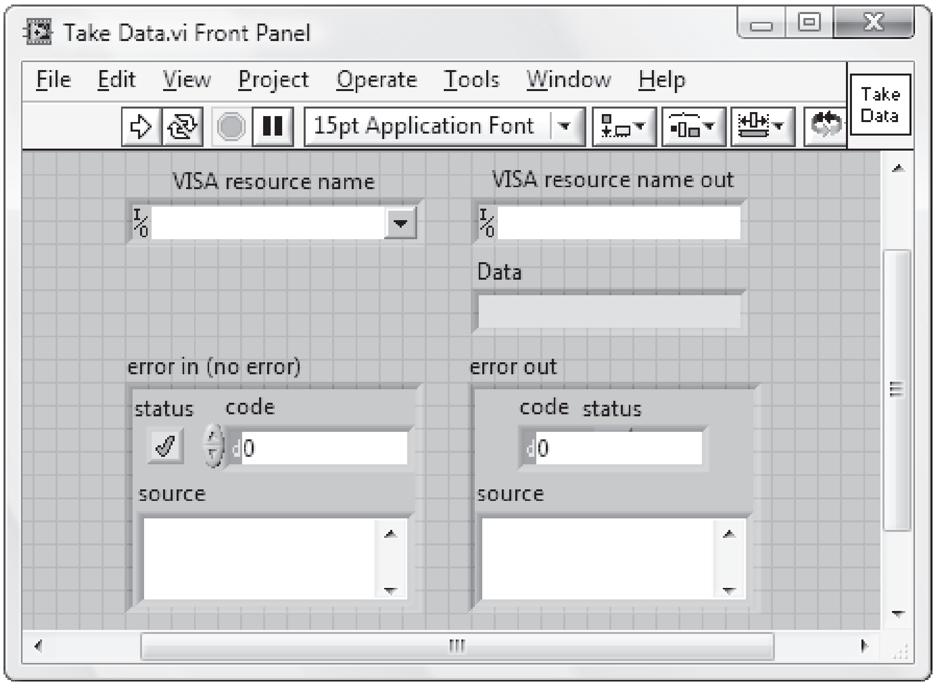
*CONF:VOLT:DC<Space>1.00E+1,1.00E-5;:ZERO:AUTO<Space>ON;:SAMP:COUN<Space> 5;*OPC?*

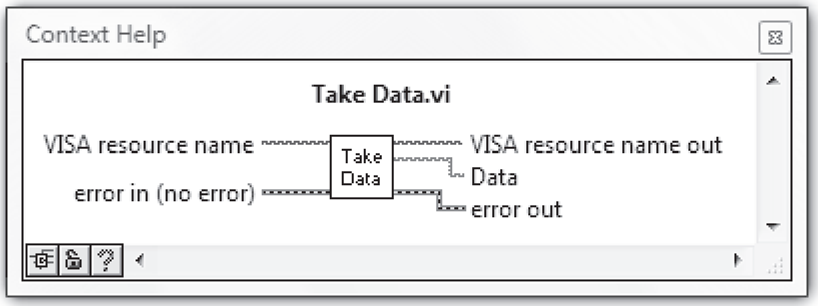
If the command is successfully sent over the GPIB, an ASCII "1" will appear in **output buffer**. If the Agilent 34401A beeps, there is most likely an error in the sent command. Open the front panel of **Command String**, then run **Measurement Config** again. Check that the concatenated command in the **Command** indicator on **Command String**'s front panel has a form as given above; make sure all of the colons, semicolons, and spaces are included. If there is an error, correct it on the block diagram of **Command String**.

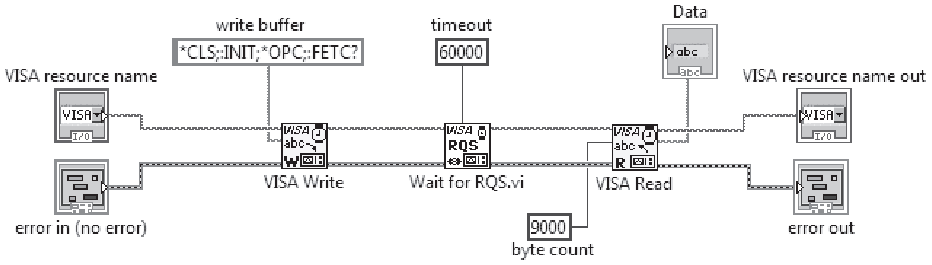
After running **Measurement Config,** the multimeter will be left in *remote mode*. You can switch to *local mode* by pressing the instrument's front-panel SHIFT/LOCAL key. The Agilent 34401A can then be triggered (equivalent to sending the INIT command over the GPIB) with the SINGLE/TRIG button. A star (*) annunciator will blink on the instrument's front-panel display as it acquires each voltage sample. Does this annuciator blink **Sample Count** times after the SINGLE/TRIG button is depressed?

Save **Measurement Config** as you close it.

Write a final modular VI for your Agilent 324401A instrument driver called **Take Data,** as shown below, and save it in **YourName\Chapter 13**. The leading **CLS* command assures that all bits in the SESR and SBR register are set to zero, prior to each data-taking process.





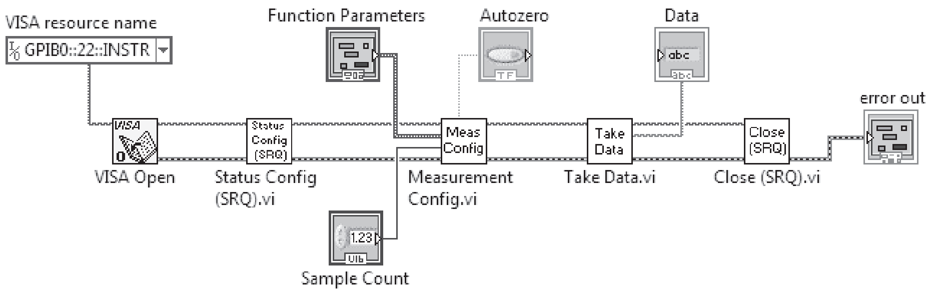


Note: **Take Data** cannot be run independently without generating an error. However, if you first run one of the other VIs that you have written (can you figure out which one?), then **Take Data** can be run successfully.

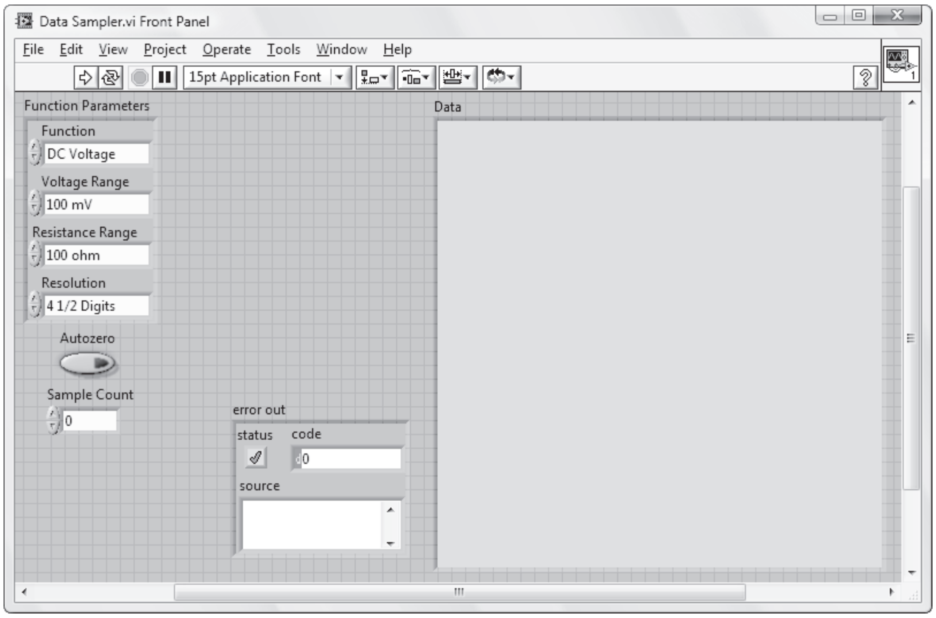
13.16 USING THE INSTRUMENT DRIVER TO WRITE AN APPLICATION PROGRAM

Ultimately, the merit of an instrument driver is measured by the ease with which you can use it to write an *application program* to fulfill some specialized need in your laboratory work. Let's quickly write an application program called **Data Sampler** that can be configured to take a multisample voltage or resistance measurement.

With **VISA Query (SRQ)** open, use **File>>Save As...** to create **Data Sampler** and save it in **Your Name\Chapter 13**. Rewrite the block diagram using your modular driver software as shown below.



Return to the front panel and arrange the object there as desired. Save your work.


Data Sampler.vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font

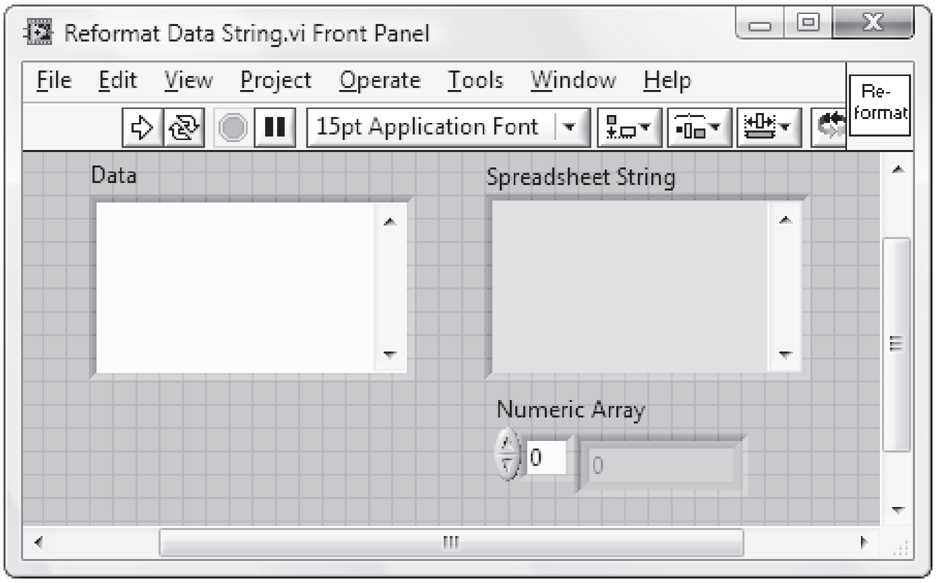
Function Parameters
Function: DC Voltage
Voltage Range: 100 mV
Resistance Range: 100 ohm
Resolution: 4 1/2 Digits
Autozero
Sample Count: 0
error out
status ✓ code 0
source
Data

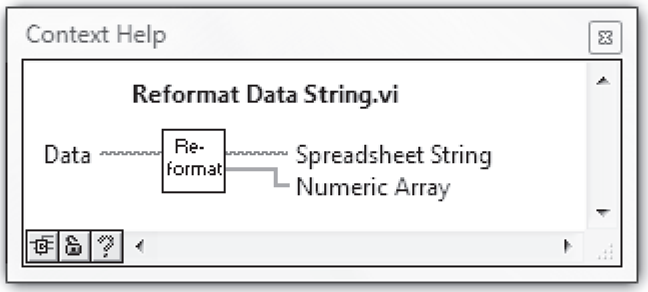
Run **Data Sampler** with various choices of front-panel settings, and then pat yourself on the back for a job well done.

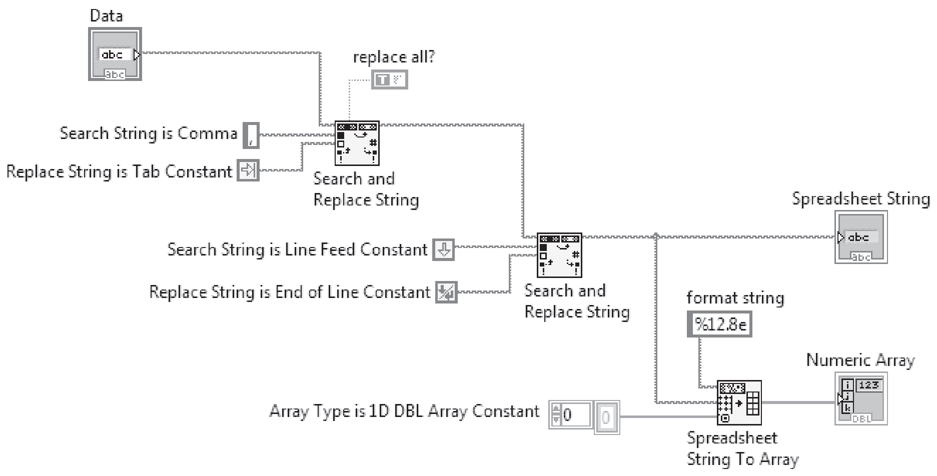
Now that you know some of what goes into writing an instrument driver, here's some very good news. In many cases, the LabVIEW instrument driver you will need for a particular instrument in your laboratory has already been written and is available for your use free of charge. National Instruments provides an extensive library of downloadable instrument drivers at *http://www.natinst.com/idnet/*. You can also access this resource within LabVIEW by selecting **Tools>>Instrumentation>>Find Instrument**

Drivers… Most of these drivers are written using VISA icons and so, by using the interface-appropriate VISA resource name for your instrument, can be used to communicate over various interface buses—RS-232, GPIB, Ethernet, and USB. LabVIEW itself comes equipped with the VISA driver for the Agilent 34401A Digital Multimeter in **Functions>>Instrument I/O>>Instrument Drivers>>Agilent 34401**. Take a look at some of these subVIs and see if you can decipher them.

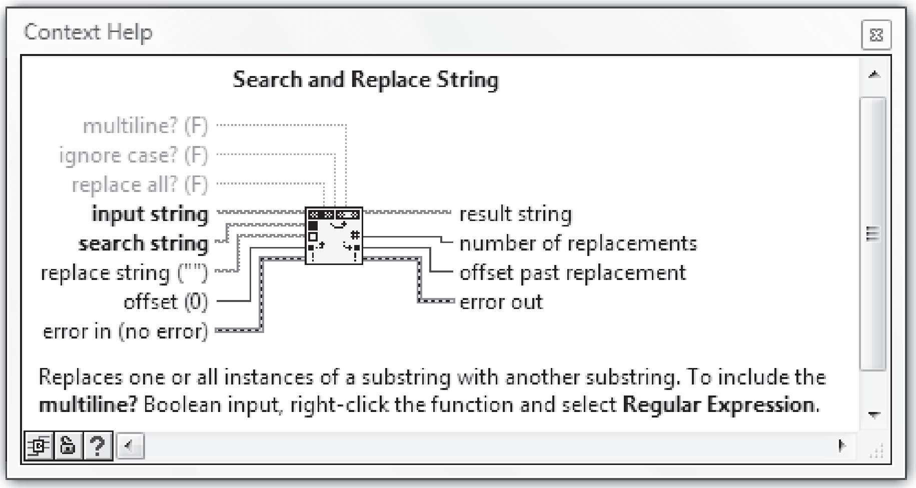
Finally, a useful Agilent 34401A instrument driver utility would perform the following task: Take the instrument's **Data** string (data samples delimited by commas and the string terminated by a LF character) and convert it to a numeric array and a spreadsheet format. One manifestation of that utility, called **Reformat Data String,** is shown next. On the front panel, the string control and string indicator have been resized and scrollbars have been activated by selecting **Visible Items>>Vertical Scrollbar** in the pop-up menus.


Reformat Data String.vi Front Panel
File Edit View Project Operate Tools Window Help
15pt Application Font
Re-format
Data
Spreadsheet String
Numeric Array
0
0




Data
replace all?
Search String is Comma
Replace String is Tab Constant
Search and Replace String
Search String is Line Feed Constant
Replace String is End of Line Constant
Search and Replace String
Spreadsheet String
format string
%12.8e
Numeric Array
Array Type is 1D DBL Array Constant
Spreadsheet String To Array

If interested, try writing **Reformat Data String**. It implements the **Search and Replace String** icon found in **Functions>>Programming>>String** (Help Window shown below) to coerce the original **Data** string into the spreadsheet format (by replacing comma delimiters and the LF terminator with tabs and an EOL, respectively). Do you understand how it works? Once written, include this program as a subVI in **Data Sampler** and watch it perform its magic.



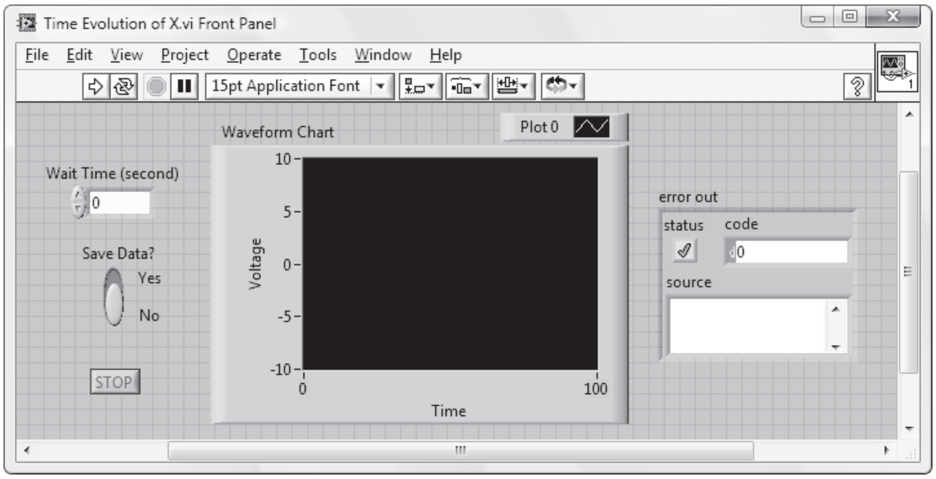
Replaces one or all instances of a substring with another substring. To include the **multiline?** Boolean input, right-click the function and select **Regular Expression**.

DO IT YOURSELF

Assume that you have a widget in your laboratory that is providing you with some interesting information about *X*, where *X* might be the position of an object or the intensity of a light source. Additionally, say, the widget provides this information about *X* in the form of a "voltage-code," that is, it produces an output voltage *V* that is some known function of *X*. Then, with an Agilent 34401A multimeter and an appropriate application VI (called it **Time Evolution of X**), you can monitor *X* (via measurement of *V*) as a function of time.

Using your Agilent 34401A instrument driver programs as subVIs, write **Time Evolution of X**. When run, this top-level VI continuously obtains a single DC voltage sample every **Wait Time** seconds (where **Wait Time** is given by the value on the similarly named front-panel control) until the front-panel **Stop Button** is clicked. While running, the VI provides real-time graphing of the *Voltage versus Time* data on a **Waveform Chart** with the Chart's *Time* axis properly calibrated. The front panel also provides the option of storing the all of the accumulated data in a spreadsheet file with *Time* and *Voltage* in the spreadsheet's first and second column, respectively. Define *Time* = 0 at the moment that the first voltage sample is acquired.

The front panel of **Time Evolution of X** should appear as shown below. All needed parameters without a front-panel control should be input on the block diagram. After building this VI, run it to observe a time-dependent voltage input (e.g., from a function generator) to the multimeter and save the resulting *Voltage versus Time* data in a spreadsheet.



A helpful tip: The *Time* axis of the Waveform Chart can be calibrated using a Property Node. Pop up on the Chart's icon terminal and select **Create>>Property Node>> X Scale>>Offset and Multiplier>>Multiplier**. Then set the **Multiplier** property appropriately.

Problems

1. Thermocouples are widely used as temperature sensors. A thermocouple is constructed by joining the ends of two dissimilar metals, for example, a copper and a constantan wire for a Type T thermocouple. This junction produces a millivolt-level voltage, which has a well-documented temperature dependence, where the temperature is measured relative to a "cold junction" reference temperature. Conveniently, this cold junction can be provided by a compact electronic device called a Cold Junction Compensator (CJC), which effectively makes the reference temperature equal to 0°C.

   Connect a thermocouple to a CJC and then connect the plus and minus output of the CJC to the HI and LO Voltage Inputs of the Agilent 34401A Multimeter. Then, write a program called **Thermocouple Thermometer (VISA)** that, every 250 ms until a Stop Button is clicked, reads the thermocouple voltage, converts this value to the corresponding temperature in Celsius, and then display this temperature in a front-panel indicator.

   To convert the thermocouple voltage to its corresponding temperature, use **Convert Thermocouple Reading.vi**, which is found in **Programming>>Numeric>>Scaling**, with its **CJC Voltage** input wired to *0* (the **CJC Sensor** and **Type of Excitation** inputs can be left unwired). Program **Thermocouple Type** for your particular type of thermocouple (e.g., T).

   Run **Thermocouple Thermometer (VISA)** and use it to measure room temperature as well as the temperature of your skin.
2. As written in this chapter, **Serial Poll** is flawed in that, if the bit being monitored in the Status Byte Register is never set, this VI will loop endlessly. With **Serial Poll** open, use **Save As…** to create a new VI called **Serial Poll with Timeout**. Then modify the block diagram so that, if the bit being monitored is not set within 10 seconds, the While Loop is stopped.
3. When LabVIEW is installed on your computer, a driver for the Agilent 34401A Multimeter is included. This driver is found in **Functions>>Instrument I/O>>Instrument Drivers>> Agilent 34401**. Use the icons from this "built-in" driver to write a program called it **Time Evolution of X (Built-In Driver)**, which carries out the task described in this chapter's **Do It Yourself** project. The icon VI Tree gives a helpful overview of the "built-in" driver.
4. Regardless of the chosen resolution, the Agilent 34401A Multimeter always reports data sample values with eight digits to the right of the decimal point. Thus, some of these decimal-place values are not significant. With **Take Data** open, use **File>>Save As…** to create a new VI called **Take Data (Accurate Resolution)**. Add a **Function Parameters** front-panel control to this new VI (so that the selected resolution setting can be input) and then modify the block diagram appropriately so that the **data** output reports values with the actual resolution selected (e.g., 4½ digits, if that is the selected resolution).
5. Use the **Instrument I/O Assistant** Express VI to query the Agilent 34401A Multimeter. Place an **Instrument I/O Assistant** (found in **Functions>>Express>>Input**) on the block diagram of a VI called **Simple VISA Query (Express)**. When this Express VI's dialog window opens, select the desired instrument, and then click on **Add Step**. In the **Add Step** dialog window that appears, double click on **Query and Parse**. In the Enter a command box, type

CONF:VOLT:DC 10,0.00001;:INIT;:FETC?

and then click **Run this step**. The command will be sent to the Agilent 34401A Multimeter and its string response will be displayed. Click the **Auto parse** button to convert the response string to numeric format and then close the dialog box by clicking the **OK** button. When returned to the block diagram, simply create an indicator for the icon's **token** output terminal.

Run **Simple VISA Query (Express)** and demonstrate that it successfully obtains a DC Voltage sample from the Agilent 34401A Multimeter.