

You can create versatile software to control an instrument, as this LabView example for a Keithley 2306 battery simulator shows.



DEVELOP SOFTWARE for TEST INSTRUMENTS

BY BRIAN MCLEAN, SEMTECH

Software lets you automate test equipment, and if your software has a quality GUI (graphical user interface), test operators will be able to use it to test just about anything. Software not only lets you put a custom face to an instrument, but it also lets you develop functions that your instruments may lack.

Developing a versatile GUI takes planning and careful thought. To show you, I'll explain how I developed a GUI with LabView from National Instruments for a Keithley Instruments Model 2306 battery/charger simulator. (You can download the code for LabView versions 8.0, 8.2, 8.5, 8.6, and 2009 from the online version of this article, www.tmworld.com/2009_11). I use the code to test several devices, including linear single-cell lithium-ion battery chargers and high-efficiency charge-pump LED drivers.

Make a plan

Before you start programming, you must decide on the scope of your code as well as how robust and fast it should be. When defining the scope of the code, remember that some instruments use thousands of commands, but you may not want to include the complete set of functions if an application needs only a few. Knowing the core functionality requirements of an application will help you prioritize your development. For a power supply, for example, you'll have to develop a

routine to enable the supply's power outputs. Such a function is necessary, whereas changing a software button's color might be a luxury.

You should also determine if the test instrument can perform a required function or if you will need to implement that function in software. My code adds a datalogging option that the 2306 can't perform on its own.

To create robust code, include initialization routines as well as error-detection and error-handling routines. These will let users properly configure an instrument and recover from unexpected conditions.

Throughput is important in most test applications, so you must consider how quickly the equipment responds to the user input through the GUI. Some instruments, for example, respond in a few milliseconds

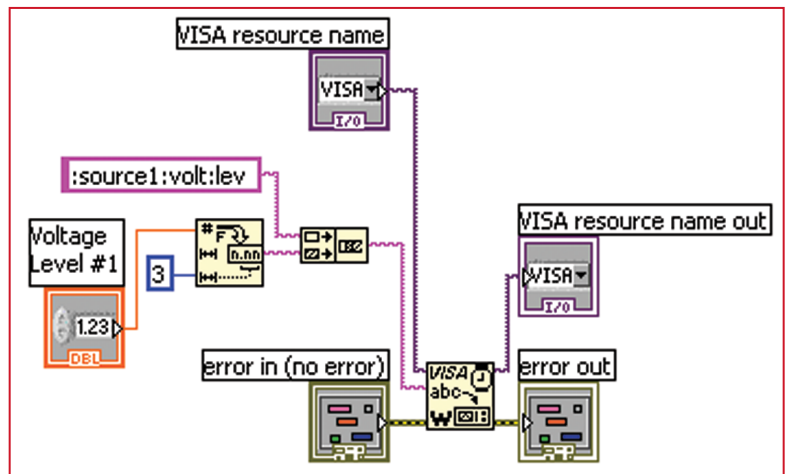


FIGURE 1. This LabView code sets the 2306's power supply for 3-V output.

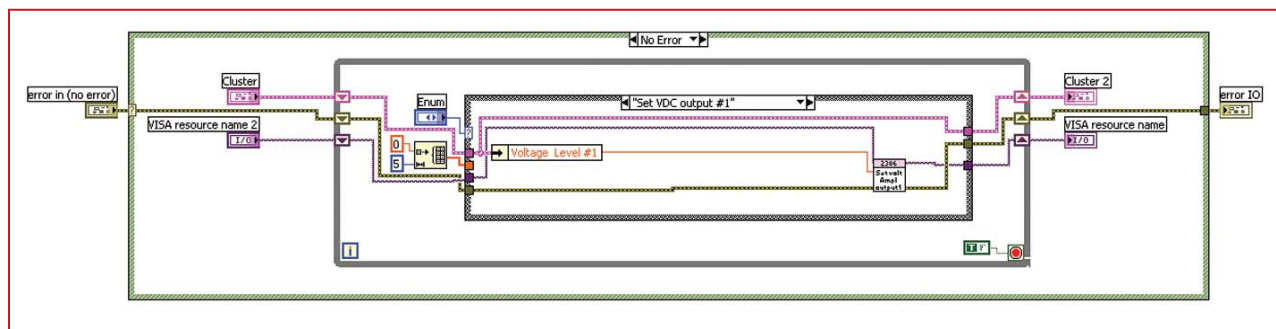


FIGURE 2. The case structure in the center resides in a while loop.

to commands, and others take several hundred milliseconds.

The response time is also determined by the bus that the PC uses to communicate with the instrument. Instrument buses include RS-232, GPIB, PXI, USB, and Ethernet. Each has unique characteristics concerning bandwidth and latency. What might work when communicating with GPIB might not work as well with USB or Ethernet because of their higher latency.

Developing instrument software is essentially a three-part process. You should start by developing or downloading the low-level drivers. Then, design a front panel for the main routine, and finally, write the code.

I recommend a modular architecture for your code, as it will let you develop routines for each instrument function and then call the routines as needed. In LabView, a software-based instrument is called a VI (virtual instrument). Each function can reside in a separate VI, called a subVI. Any VI can be a subVI if it's called by a higher-level routine.

Build or download?

The lowest level subVI in an application is the instrument driver, which sends commands to an instrument through the communications bus. You must decide if you can use an existing driver or if you have to develop your own. Check the equipment manufacturer's Website or other online repositories for available drivers. If you find an appropriate driver, you should try working with it before creating a new one—an existing driver may have all the functions you need and can save you development time.

Be sure to examine the driver code carefully, though. You may find drivers

that are created for an instrument model that is similar to yours but is not the exact model. The driver may work with your instrument, but it may not include functions you need or it may deliver different results on your instrument. For example, some drivers have an initialization stage that resets the instrument every time you call the driver. That could produce an undesirable outcome if the test specifications require that you maintain a specified output level from a power supply throughout a test.

These and other issues with publicly available drivers may force you to develop your own. For the Model 2306 VI, I wrote my own driver VIs using LabView. I relied heavily on the instrument's programming manual, because it defines the proper syntax for the instrument commands.

For example, the Keithley programming manual specifies a command “:source1:volt:lev 3” for setting the voltage level on output #1 to 3V. **Figure 1** shows the code in a LabView block diagram. The code, which is a simple write function to the instrument, will become a subVI, called by a higher-level VI as needed.

After I created the driver subVIs, such as those for setting voltage level, enabling outputs, and setting currents limits, I packaged them in code called a functional global. Packaging all the subVIs in a functional global VI (in this case, Keithley 2306 Functional Global.vi) lets you employ the drivers in many routines, because all of the commands are in one file. A functional global VI is a LabView case structure contained within a while loop (**Figure 2**). The case structure at the center of the diagram shows a case

that sets VDC output #1 on the 2306 (the instrument has two outputs).

I use the Keithley 2306 Function Global.vi routine in all code that controls the instrument. By default, all VIs and subVIs, including the functional global VI, are of a non-reentrant type. A non-reentrant VI prevents other VIs in your code from gaining access to a function while the VI is executing. This helps to prevent a race condition, which could occur when two or more parts of your code try to use the same routine at the same time.

Designing the user interface

After I created the drivers, I developed the front-panel GUI, which is the top-level VI for the instrument. **Figure 3** shows the graphical front panel of the 2306.

You should make your GUIs aesthetically pleasing and easy to use. Avoid using colors such as pinks or reds that are hard on the eyes. Simple shades of gray often draw the fewest complaints from users. Having controls clustered by functionality also makes GUIs easier to use.

(continued)

ON **TMWorld.com**

GET THE GUI CODE

The online version of this article contains the code for the GUI the author developed for the Keithley Instruments 2306 battery/charger simulator. Code is available for LabView versions 8.0, 8.2, 8.5, 8.6, and 2009. The online article also includes a block diagram that shows the run state in the GUI routine.

www.tmworld.com/2009_11

The GUI in Figure 3 uses easy-to-read controls and indicators. By grouping information into tabs (“Voltage and Current,” “Output Z,” “Protection,” and so forth) I was able to reduce the overall screen size of the GUI, making it more manageable. The GUI displays both of the instrument’s power-supply outputs. It also contains voltage and current indicators on the right side of the panel. The user can set the voltage outputs and set limits on the output current. Another control lets users set voltage step sizes. This GUI doesn’t have all the functionality of the 2306, but it has enough to make it a useful engineering tool.

Letting the user choose the step size for the voltage and current level is a useful feature when the performance of the device under test changes at a certain voltage or current level. Controlling the voltage step size gives the user more control over the point where this event occurs as a voltage increases or decreases.

Sometimes, you need a feature that’s not built into your instrument. I needed to store test data, so I used LabView to create a datalogging function that lets users store data to the computer’s hard drive. **Figure 4** shows the “Data Logging” tab of the 2306 GUI.

Start coding

Once I finished the design of the GUI, I was ready to create the underlying code. For the 2306 VI, I wanted a smooth response to user inputs. When the user presses a button or increments a control, that specific event should appear instantly. A GUI shouldn’t appear to hang if a user quickly and repeatedly presses a button. The GUI must handle all the events in the order selected and constantly give updates of its outputs. I also wanted the GUI to use as little of the host processor’s resources as possible.

After experimenting with many program architectures, I selected one that uses LabView event handlers, notifiers, and parallel loops, because I didn’t want my code to constantly poll the front-panel VI for changes to the controls. A polling architecture consumes more processing power than an event-oriented architecture.

The overall program architecture is a state machine composed of case struc-

ture and parallel loops. Using parallel loops lets you increase your program’s efficiency because the LabView compiler will place each loop in a different thread on your operating system.

The top loop—the consumer loop—stays idle until an event occurs. The loop has a case structure inside a while loop. Inside the while loop there is a “Wait on Notification” function. This function will select a case from the case

structure based on the message it receives from the “Send Notification” function in the event-handling loop. The case structure has numerous functions inside it, including the functional global VI that calls instrument drivers.

The second loop from the top is the event loop. It has an event structure inside of a while loop. When the user clicks on a front-panel control, the event loop generates a trigger and, using the “Send Notification” function, it sends a message with the new data to the top loop to perform the required function.

If, for example, the user increments one of the voltage controls, the event structure will detect the change and send a notifier call with the new data and the function to the top consumer loop.

The third loop is a producer loop. This loop constantly calls the read-measurement state of the Keithley 2306 Functional Global.vi and displays the voltage levels, current limits, and the instrument’s DMM (digital multimeter) inputs of both channels. I inserted a small delay in this loop to prevent it from consuming too much of the processor’s resources. This loop also sends data to the datalogging loop.

The final loop, the datalogging loop, also consists of a case structure inside a while loop. Inside the true case is another while loop. When the user selects datalogging, the code will save data to

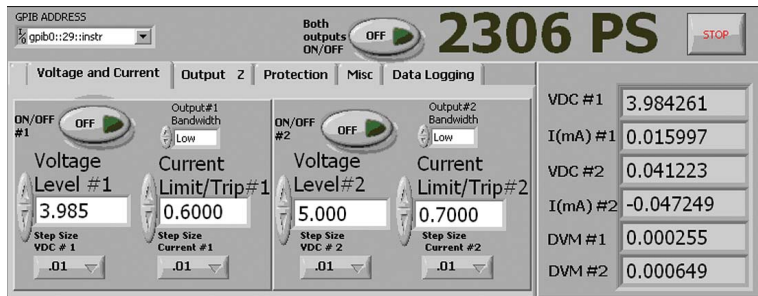


FIGURE 3. The Keithley 2306 GUI has tabs that minimize its size while providing access to instrument functions.

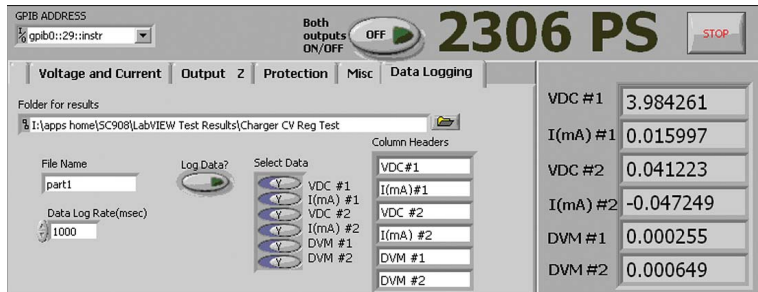


FIGURE 4. The “Data Logging” tab lets a user store data to a file on the host computer.

tures inside while loops. The online version of this article contains a block diagram that shows the run state in the 2306 GUI routine (www.tmworld.com/2009_11). This state runs constantly until the user presses the stop button. The VI contains several other states (not shown in the block diagram) that perform initialization and error handling. Event handling gives my GUI the needed quick response to changes to the front-panel controls. The event structure, when not being triggered by a front-panel event, is idle and uses very little processing power.

The online diagram shows the four parallel loops used in the program. They include a consumer loop that calls various control functions, a producer loop that displays data, an event loop, and a

a file. The user can select a file name and folder to store the data. The user can specify, in milliseconds, how often the code will save data to the file.

Uses for the VI

I use the Keithley 2306 to test several devices, including a linear single-cell lithium-ion battery charger. When developing a tester for a battery charger, you should know how various types of new batteries perform. One of the use cases I built into the Keithley 2306 GUI is a battery-discharge test. To run a battery-discharge test, an operator performs the following steps:

1. Connect battery to channel #2.
2. Set voltage level #2 to around 2.4VDC.
3. Set current limit #2 to about 100 mA.
4. Turn log data feature on.
5. Turn on output #2.

After several hours of testing, the VI will have collected enough data for you to create battery-discharge curves with a spreadsheet. Without this routine, an operator would have to collect the data by hand, and the level of detail in the data would be impossible to obtain.

I also use the 2306 GUI to test a high-efficiency charge-pump LED driver. The software maintains a constant current level in an LED even though battery voltage may vary. When the battery voltage reaches certain thresholds, the driver will change modes (1X, 1.5X, or 2X) to ensure LEDs get a constant current level.

Finding the exact transition point is critical to creating product specifications and to understanding the behavior of the part. Using the step function in the GUI to increment or decrement the supply voltage in small steps lets application engineers isolate the mode-transition point. Stepping the power supply's

output in equal steps is difficult to do with just the instrument's front-panel controls. With the automated test that the VI provides, application engineers can monitor the output levels and watch for a transition point.

My GUI has made it easier for operators to access the functionality built into the Keithley 2306. Before I deployed it, though, I thoroughly tested it by using both invalid settings as well as the expected setting. Be sure to do the same with any software you write—using unexpected settings will help you verify how robust the code is before it is put to use in the lab. T&MW

Brian McLean is a test engineer at Semtech and is a National Instruments Certified LabView developer. He designs and creates custom automated test systems to validate new semiconductor designs. He trained in electronics with the US Navy and has 20 years of experience working with electronic systems and software. BMcLean@semtech.com.