IVI State Caching for Groups of Attributes

by
Ian Dees
Application Software Engineer
Anritsu Company

Category:
Telecom

Products Used:
LabWindows™/CVI
Instrument Driver Development Wizard
PCMCIA-GPIB card

The Challenge:
Write an instrument driver capable of caching the values of multiple attributes that are set by the same command. The driver should avoid sending redundant command strings to the instrument, but must also make sure the instrument always has the correct values.

The Solution:
Using the advanced features of LabWindows/CVI, create shared read and write callback functions, which decide when to send commands to the instrument based on a set of hidden state variables.

Introduction
Anritsu is using LabWindows/CVI to write an Interchangeable Virtual Instruments (IVI) driver for computer-based control of Optical Spectrum Analyzers. One of IVI’s benefits is state caching, through which the driver optimizes performance by remembering the state of the instrument. This feature is difficult to implement for some instrument attributes, which can only be accessed by complex commands that set a group of attributes at once. Anritsu and National Instruments collaborated on an innovative solution, which is able to deliver the performance boost of state caching even in difficult cases when a single command sets many attributes.

Many Attributes Sharing One Command
When a command sets two attributes at once, the Instrument Driver Developers Guide recommends considering one of them to be “dominant” (2-20). When the driver writes the dominant attribute, it picks a reasonable default value for the subordinate one. This method works well for attribute pairs, but larger commands complicate the picture. Imagine a command that sets five attributes at once. We would have to select a dominant attribute, a “second-place” attribute, a “third-place” attribute, and so forth. For each combination of attributes one through four, we would need to choose a default value for the fifth. The attributes would always have to be set in the same order, lest one default value overwrite another.

National Instruments Driver Support engineer Edward Zhu recommends a new approach: create an integer attribute representing the state of the entire group. This “state attribute” is hidden: the user must not be able to put the driver into an incorrect state. The IVI engine cannot cache this variable, because the write callback must be called every time the state attribute is changed. Both properties are easy to set in LabWindows/CVI (Figure 1).
The state attribute can have one of four values: “1 means no updates (instrument I/O) required and defer all future updates, 2 means future updates required, 3 means update now, and 4 is the do nothing state.” (Zhu) Figure 2 shows how our driver will manage the attributes based on the state attribute.

To manage the transitions between states, we’ll define three callback routines:

- **The shared read callback** is called by the IVI engine whenever a user-level attribute is read. The driver simply has to “do an instrument I/O, parse the data string, and update the cached values of all the attributes accordingly” (Zhu).

- **The shared write callback** is called when an attribute is changed. This function sets the state attribute to inform the driver that the values in memory no longer match those in the instrument (Zhu).

- **The state attribute write callback** (Figure 3, parts omitted for brevity) is only invoked when our state-caching code changes the state attribute. If the driver is in state 3 (i.e., if some attributes have changed), this callback sends all the attribute values to the instrument (Zhu).
The state-transition behavior is hidden inside a function the user can call to set all the attributes at once. At the beginning of this high-level function (Figure 4), we set the state attribute to 1 to hold communications until all attributes have been set. Next, we call the appropriate SetAttribute function on each attribute. At the end of the function, we set the state attribute to 3, so that all the attributes will be sent to the instrument (Zhu).

**Finishing Touches**

We can make writing high-level configuration functions easier by moving the state transition logic into the callback functions. To simplify the process, let’s reexamine the state definitions (Figure 5):

State 1: None of the attributes have been changed (the attributes are “clean”). If any of them change in the near future, don’t send any I/O (the attributes are “locked”).

State 2: At least one attribute has changed (the attributes are “dirty”). The attributes are still locked, so do not send any I/O.

State 3: The attributes are dirty and have just been unlocked. Send the new values to the instrument.

State 4: The attributes are clean and unlocked.
Figure 5: Modified State Diagram

We have replaced the integer state attribute with a pair of Boolean state attributes, Locked and Dirty, both of which share the state attribute write callback. Our high-level function (Figure 6) is now simpler: we set Locked to True before changing our attributes. When we’re done, we set Locked to False, and the state attribute write callback (Figure 7) will check the value of the Dirty and Locked flags to decide whether or not to send any communication.

```c
// Lock the group, so that no I/O will happen until we unlock it.
Ivi_SetAttribute(ViBoolean(vi, "EXAMP_ATTR_TEST\_LOCKED", 0, VI_TRUE));

// Set the values of the attributes.
Ivi_SetAttribute(ViInt32(vi, "EXAMP\_ATTR\_TEST1", 0, test1));
Ivi_SetAttribute(ViInt32(vi, "EXAMP\_ATTR\_TEST2", 0, test2));
Ivi_SetAttribute(ViInt32(vi, "EXAMP\_ATTR\_TEST3", 0, test3));
Ivi_SetAttribute(ViInt32(vi, "EXAMP\_ATTR\_TEST4", 0, test4));
Ivi_SetAttribute(ViInt32(vi, "EXAMP\_ATTR\_TEST5", 0, test5));

// Unlock the group and send any changed attributes.
Ivi_SetAttribute(ViBoolean(vi, "EXAMP\_ATTR\_TEST\_LOCKED", 0, VI_FALSE));
```

Figure 6: New High-Level Configuration Function
In the shared read callback Figure 8), after we query the instrument, some of the retrieved values may be older than the cached values we're holding. For each attribute, we call AttributeIsCached to see whether or not its in-memory value has changed; if the attribute has not changed, we call SetAttribute to update it with the new value from the instrument. At the end of the shared read callback, a quick check of the “attributeId” parameter tells us which attribute we must return.
**Further Improvements**

In most cases, the attributes in a group are more diverse than in our five-integer example. Instead of sharing one read callback and one write callback, the group might share one integer read callback, one integer write callback, one floating-point read callback, and so on. These shared callbacks could call the same implementation function.

Developers who are interested in more complex behavior could easily extend this state-caching scheme. For example, if one attribute’s range depends on another attribute’s value, more sophisticated range checking could be added into the callbacks. Another potential improvement would be to turn off our state-caching scheme when the user wants to disable state caching for the entire driver.

**Works Cited**


Zhu, Edward. [Instrument.Driver@ni.com]. “RE: IVI Question.” Private e-mail message to Ian Dees. [ian.dees@anritsu.com]. 20 October 1999.