Tutorial: Local Variable, Global Variable, and Race Conditions

Publish Date: Jan 09, 2013

Overview

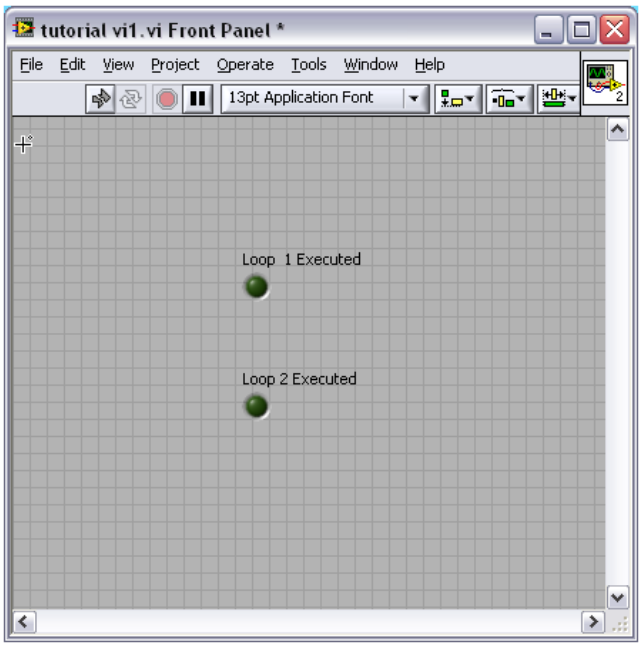
In NI LabVIEW software, the order of execution is controlled by the flow of data (data flow) though wires rather than the sequential order of commands. This allows you to create a block diagram with simultaneous (parallel) operations. When you have parallel loop structures, you cannot use wires to communicate data between the two loops because data flow prevents parallel operation. To overcome this, you must use variables. With variables, you can circumvent normal data flow by passing data from one place to another without connecting the two places with a wire. In LabVIEW, variables take many forms. This tutorial explores the local and global variable as well as race conditions, which can result from the improper use of variables.

Table of Contents

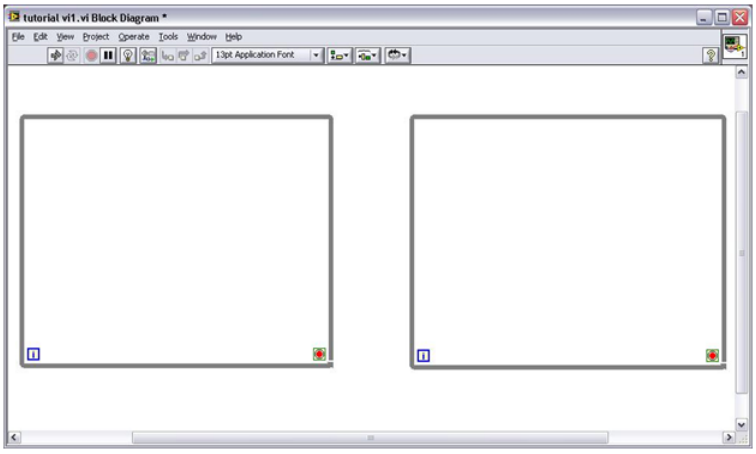
Necessity of Variables in LabVIEW

The following steps demonstrate the need for using variables in LabVIEW.

1. Open a blank VI.
2. Save the VI as **Parallel Loops.vi**.
3. Right-click to open the **Controls** palette and navigate to **Modern»Boolean»Round LED**.
4. Place two **Round LED** Boolean indicators on the front panel.
5. Rename the indicators by double-clicking each text label **Boolean** and typing the title **Loop 1 Executed** and **Loop 2 Executed**.

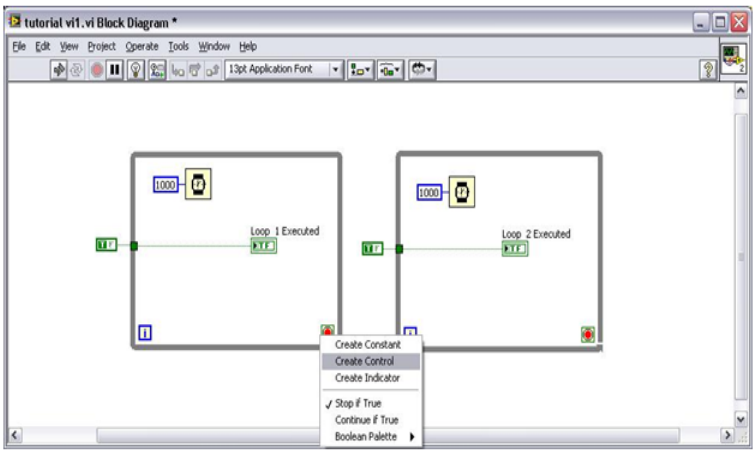


6. From the menu bar, select **Window»Show Block Diagram**.
7. Right-click to open the **Functions** palette and navigate to **Programming»Structures»While Loop**.
8. Place two while loops on the block diagram.

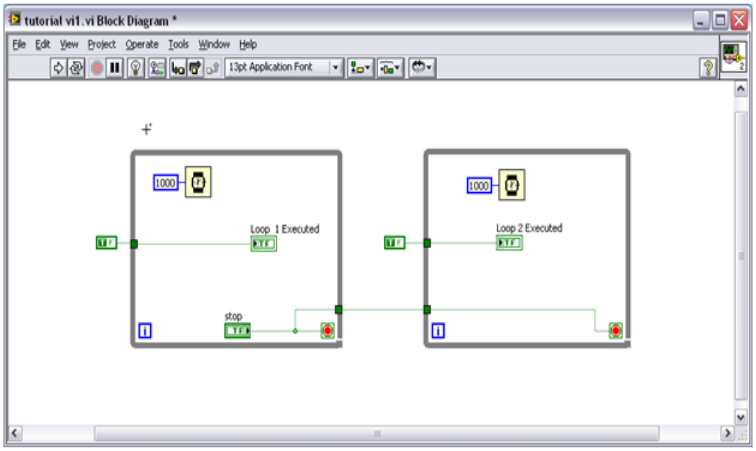


9. Right-click to open the **Functions** palette and navigate to **Programming»Timing»Wait (ms)**.
10. Place one Wait (ms) VI in each while loop.
11. Right-click on the input terminal of one Wait (ms) VI (milliseconds to wait) and select **Create»Constant**.

12. Double-click the Numeric Constant and type “1000.”
13. Right-click on the input terminal of the second Wait (ms) VI (milliseconds to wait) and select **Create»Constant**.
14. Double-click the Numeric Constant and type “1000.”
15. Place one of the Boolean indicators into each while loop.
16. Right-click to open the **Functions** palette and navigate to **Programming»Boolean»Boolean Constant**.
17. Place one Boolean constant outside each while loop.
18. Wire each Boolean constant to one of the Boolean indicators in each loop.
19. Right-click the Loop Conditional Terminal and select **Create»Control**.

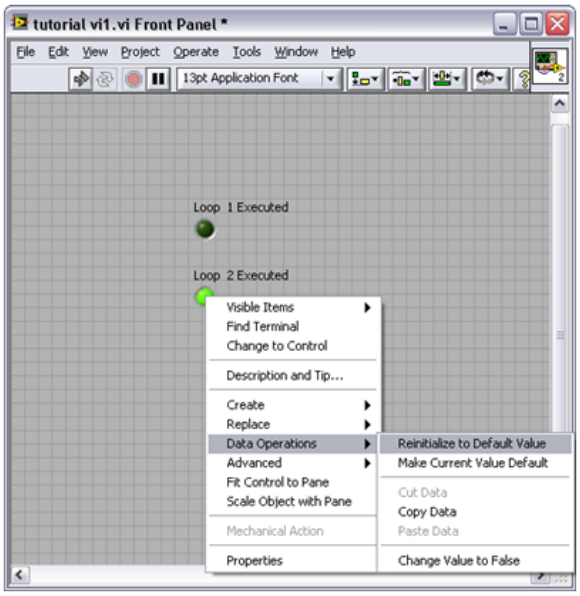


20. Run an additional wire from the newly created Boolean Stop Control to the Loop Conditional Terminal of the second loop.



Run this VI. Note that the **Loop 1 Executed** indicator immediately displays a value of true while **Loop 2 Executed** remains false. Press the **Stop** Boolean control. Note that although you have the stop button wired to the loop conditional terminals of both loops, only the first loop immediately stops when you press the Stop button control. Once the first loop stops, the true value is passed out o the first while loop to the second while loop. The second while loop can then execute. The **Loop 2 Executed** Boolean indicator should now display a value of true. After a wait of 1 second (1000 ms a true value is passed to the Loop Conditional Terminal, which stops the second while loop. Because there is no more code to execute, the VI stops executing.

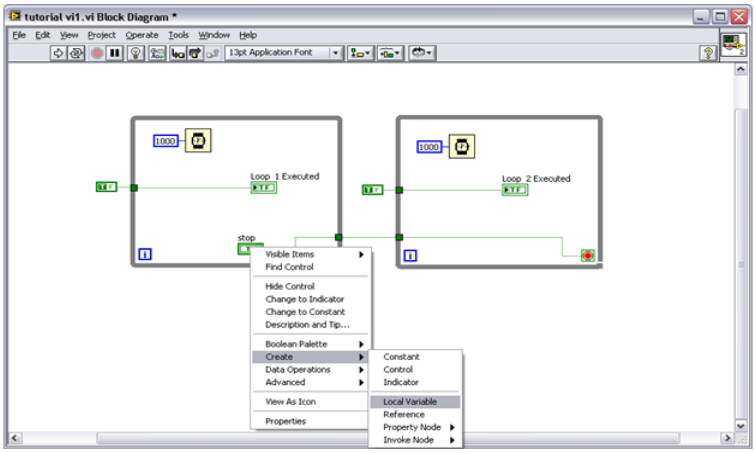
Tip: The two Boolean indicators retain their most recent value after the VI has stopped running. With the program you have created, these two indicators retain a true value. You can set this back to false by right-clicking on the indicator (on either the block diagram or the front panel) and selecting **Data Operations»Reinitialize to Default Value**.



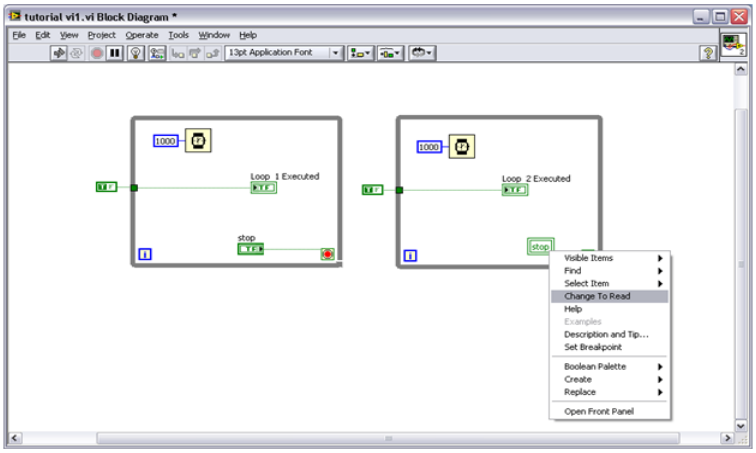
Implementing a Local Variable

Using the same VI you just created, implement local variables so you can execute both while loops simultaneously as well as stop both while loops simultaneously using a single Boolean control.

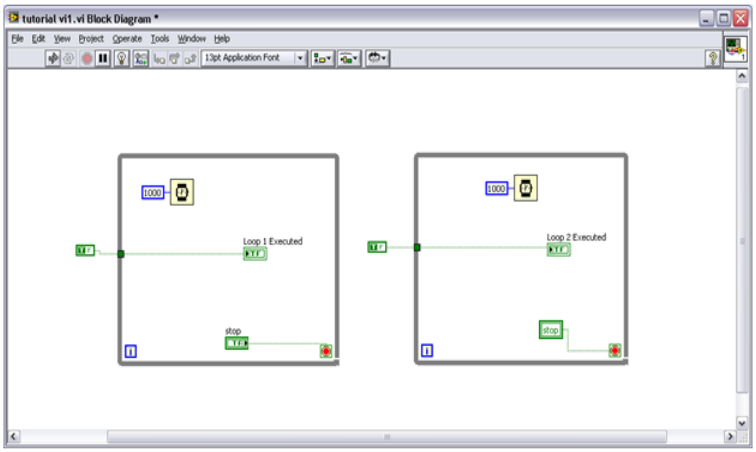
1. Change the milliseconds to wait constant to 10 for both Wait (ms) VIs.
2. Delete the wire running from the Stop Boolean control to the Loop Conditional Terminal of the second while loop.
3. On the block diagram, right-click the Stop Boolean control and select **Create»Local Variable**.



4. Place the local variable in the second while loop.
5. Right-click the local variable and select **Change to Read**.

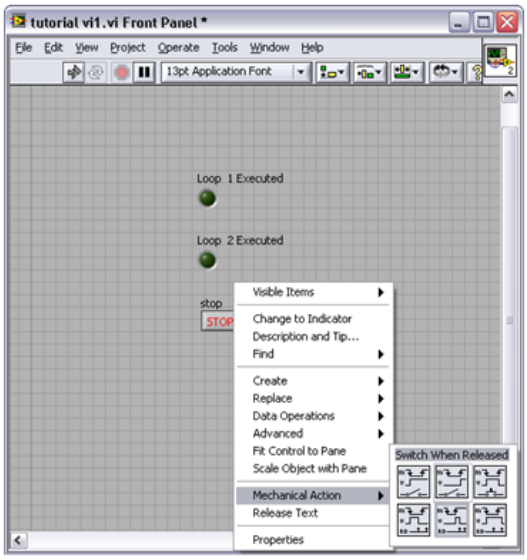


6. Wire the output of the local variable to the input of the loop conditional terminal.

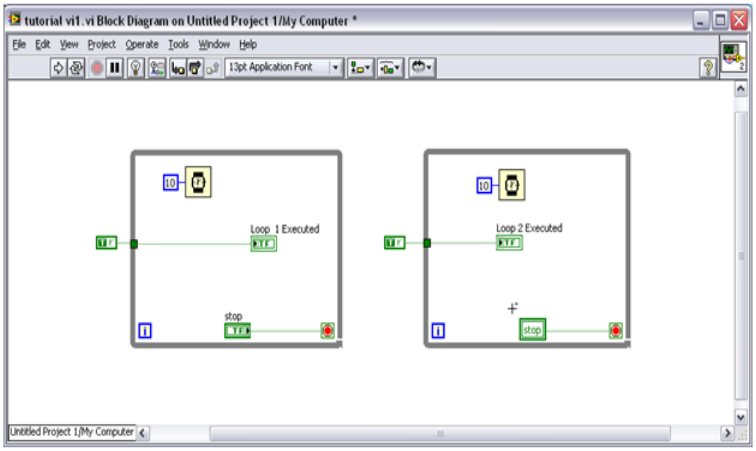


Note: At this point, you are not able to run the VI because Boolean controls associated with a local variable cannot use latch mechanical action. For example, **Latch When Released** changes the control value only after you release the mouse button within the graphical boundary of the control. When the VI reads it once, the control reverts to its default value. A switch mechanical action does not revert back to its default value.

7. On the front panel, right-click the Stop Boolean control and select **Mechanical Action»Switch When Released**.



8. Run the VI.

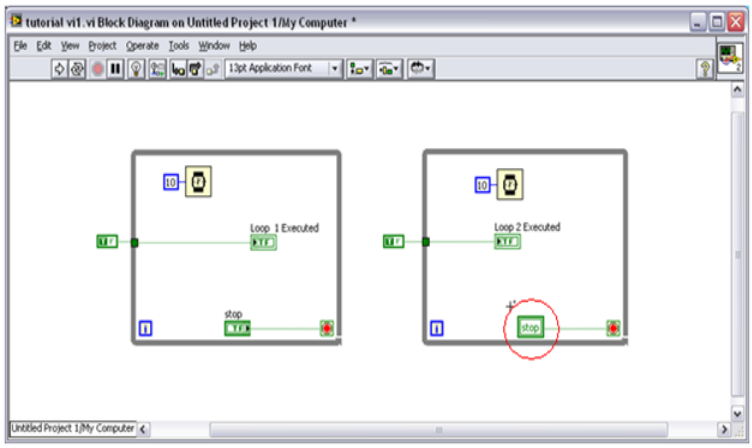


Notice that the Boolean indicators associated with each loop immediately display true when you run the VI, and both loops stop running when you press the Stop Boolean control.

Implementing a Global Variable

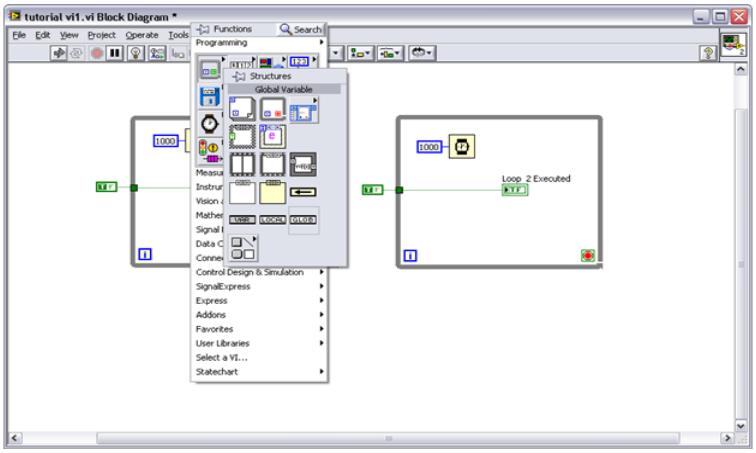
You can use local variables only to pass data between two different locations in a single VI. If you require control over two separate VIs running in parallel or a VI and subVI, you must use a global variable.

1. Select **File»New Project**.
2. When prompted to add the open VI to the new project, select **Add.** The open VI should be the VI you created in the previous section of this tutorial.
3. Delete the local variable from the second while loop.



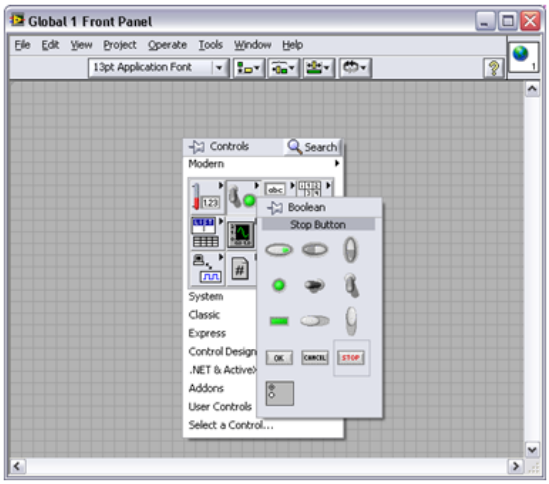
4. Within the block diagram, right-click to open the **Functions** palette.

5. Navigate to **Structures»Global Variable**.



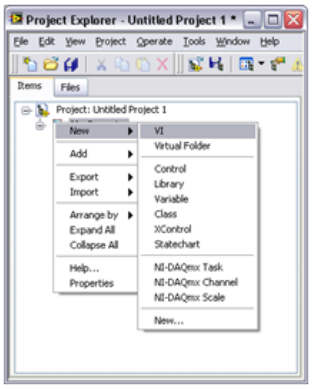
6. Double-click the global variable to open its front panel.

7. Right-click the global variable front panel and navigate to **Modern»Boolean»Stop Button**.



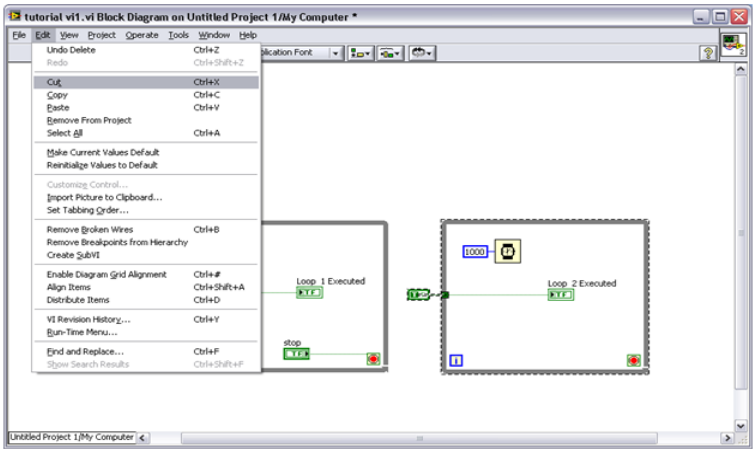
8. Save the global variable as Global 1.vi and close its front panel.

9. In the Project Explorer window, right-click My Computer and select **New»VI**.



10. Save the New VI as Second Loop.vi.

11. In Parallel Loops.vi, select the second while loop as well as the constant and wires running into the while loop.

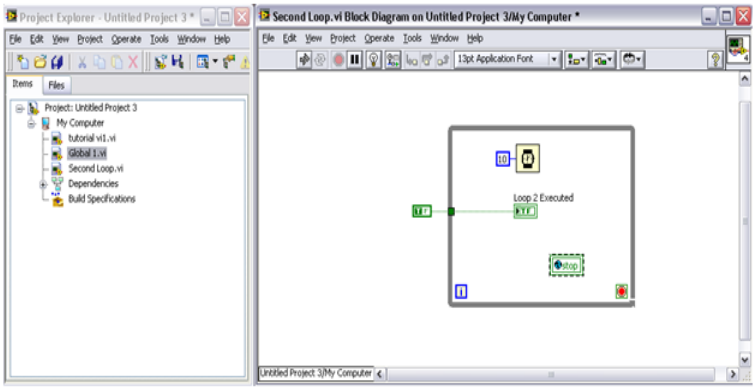


12. Select **Edit»Cut**.

13. Open the block diagram of Second Loop.vi and select **Edit»Paste**.

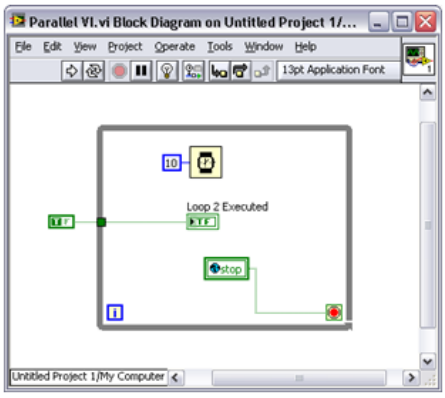
14. Save Parallel Loops.vi as Parallel Global.vi.

15. From the Project Explorer window, drag and drop Global 1.vi onto the block diagram of Second Loop.vi.



16. Right-click the global variable icon and select **Change to Read**.

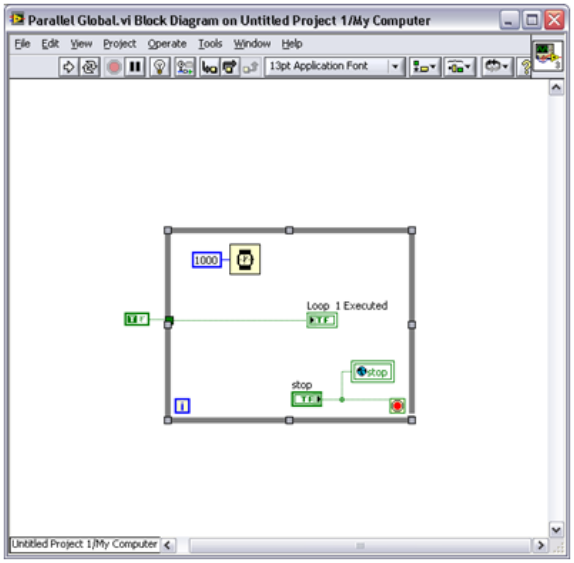
17. Wire the Stop global variable to the Loop Conditional Terminal.



18. Save the VI.

19. Open Parallel Global.vi.

20. Place the Stop global variable inside the while loop and wire it to the Stop Boolean control.



21. Run both VIs.

22. Press the Stop button on Parallel Global.vi – note that both VIs stop executing simultaneously.

Race Conditions

A race condition is when the execution timing or order of a program unintentionally affects an output or data value. This is the same condition that can occur within text-based programming. Howeve dataflow programming prevents race conditions. In instances where data flow breaks down, such as in parallel programming, race conditions can occur.

Open the attached Race Condition.lvproj and open counter 1.vi. This VI uses shared variables to pass data between the two loops. Shared variables are similar to global variables except they also contain error in and out terminals and have the ability to be published across the network. In this VI, you are simply using the shared variables to communicate data locally between the two loops.

The two loops increment the same variable in each iteration. The expected result of running this VI is **Total Count,** which is the sum of **Count 1** and **Count 2**. In actuality, **Total Count** displays a value less than the sum of **Count 1** and **Count 2**. The behavior worsens the longer the code is run. This is due to a race condition present in the VI. In a single processor computer, this code is actually running sequentially, but LabVIEW and the operating system switch between tasks rapidly so the code can essentially operate in parallel. Note that both loops perform the following operations:

- Read the shared variable
- Increment the value read
- Write the incremented value to the shared variable

Now consider what happens if the loop operations happen to be scheduled in the following order:

1. Loop 1 reads the shared variable
2. Loop 2 reads the shared variable
3. Loop 1 increments the value it read
4. Loop 2 increments the value it read
5. Loop 1 writes the incremented value to the shared variable
6. Loop 2 writes the incremented value to the shared variable

When this occurs, both loops write the same incremented value to the shared variable. This generates a race condition and can cause problems if you expect the result to be an exact sum of **Coun 1** and **Count 2**.

More information

- Exercise (Download)
- Video
- Variables
- Modules Home
- FIRST Community