PHYC 500: Introduction to LabView

M.P. Hasselbeck, University of New Mexico

Exercise 14 (v 1.2) Producer-Consumer Loops

This LabView project re-visits Exercise 12, in which a State Machine monitors a simulated temperature source and flashes a warning LED when the temperature is out of a specified range. The State Machine is useful because it is very easy to program logical branching to different operations (states). A limitation is that it can only be in one state at any given time. This is a problem if an application requires continuous collection of data but the VI is busy doing something else. In Exercise 12, no temperature data is taken for 3 seconds while the LED is flashing. Ignoring a temperature sensor for *any* period of time could be a serious problem in a real experiment. A solution is to use LabView's capability to run asynchronous, parallel loops. This will be developed and implemented in this exercise; A Producer-Consumer structure will run simultaneously with a loop that shares data only via Local Variables. It eliminates the various race conditions associated with Local Variables.

The starting point is the VI that was created for Exercise 12. Open its block diagram and use it as a reference. In a new blank VI, create two parallel While Loops and stack them vertically. In the top loop, copy the temperature VI, the numerical temperature indicator, and the Waveform Chart from the Acquire state. Create a Boolean control button for the Conditional Stop. Add a Wait function to run the loop at 10 Hz. In the second loop, copy the range logic from the Analyze state and also copy the code from the Alarm state that causes the Front Panel LED to flash three times (500 ms on-off). Make the necessary wiring connections. The LED flashing operation should go in a Case Structure that is controlled by the OR gate. Set the Wait time in the second loop to run at 20 Hz.

The temperature data in the lower loop enters via a Local Variable. Right-click on the temperature indicator in the first loop and create a Local Variable. Change it to read and wire it inside the second loop. Create a Local Variable for the Stop button Boolean and wire this to the conditional stop of Loop 2. The mechanical action of the button should be changed to 'Switch until released' (press and hold the button to stop the VI). This is because the default latching mechanism cannot be handled with Local Variables. Place a shift-register on the second loop and use it to count the number of alarm events.

The Front Panel should look essentially the same as what was built for Exercise 12. When you run this VI, you will see that it continues to collect temperature data even when the LED alarm is signaling. There is, however, a potential problem. If the alarm range is set tighter, eg. reduced from 15—25 degrees to 16—24 degrees, you will notice that an occasional out-of-range alarm event is missed. The reason is that the second While Loop is busy flashing the LED instead of reading the Local Variable. This is called a “race condition”. It is the primary danger when using Local Variables in LabView code running simultaneous operations.

There is actually a second race condition present. Because the second loop is running at a rate that is two times faster than the top loop, it will sometimes be reading the temperature data *twice*! When the Local Variable is polled repeatedly, there is no way of knowing whether its value has been updated. Grabbing the same data point more than once while missing other data completely is almost never a good idea. Note that parallel loops are not always necessary to have a race condition.

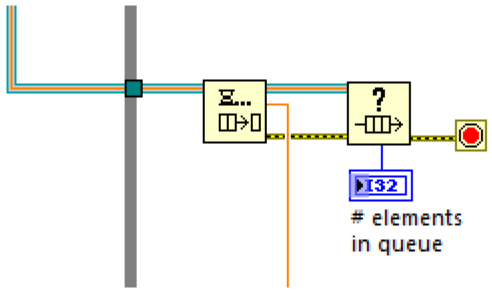
One way to address the race condition is to use queues. The most common structure for implementing queues in LabView is the Producer-Consumer structure. There is a template for this built into LabView, but here the structure will be built up in the open VI. Right-click in the Block Diagram, and select Synchronization: Queue Operations. Pin this palette open.

The top loop will be modified to function as the producer. It is configured as follows: Add two shift registers to this loop. Place 'Obtain Queue' on the outside left of the loop, place 'Enqueue Element' inside the loop, and place 'Release Queue' on the right outside of the loop. Connect them together left-to-right using the error cluster by wiring it through the While Loop using the lower shift-register (i.e. do not create new tunnels). If this doesn't make sense, examine the Producer/Consumer Design Pattern template for guidance.

The next step is to configure the queue for a specific data type. Here, we will be queuing temperature data that is in DBL floating point format. Place a DBL constant on the **element data type** input terminal of 'Obtain Queue'. This constant can have any value; its sole purpose is to define the data representation of the queue. Wire **queue out** to the upper shift-register, then to 'Enqueue Element'. Connect **queue out** to the right shift-register terminal and then to 'Release Queue'. Consult the Producer/Consumer template if this is confusing. Notice the wire containing the queue data: it is orange (indicating DBL data) surrounded by dark green. The inner wire changes color to match the data type of the queue. Wire the temperature data to the **element** input terminal of 'Enqueue Element'.

Now create the consumer loop. Make a copy of the second loop and place it directly under the first two. One way to do this is to select the second loop and drag it while pressing the CTRL key. You can also copy-paste. You should now have 3 loops. In the new loop, delete the two Local Variables and the Wait function. Two duplicate indicators will appear on the Front Panel for a second LED and an alarm event counter. Place these adjacent to the original indicators and clearly label them so they can be distinguished from operations in the second loop, i.e. the loop containing Local Variables.

Inside the consumer loop, place 'Dequeue Element' and 'Get Queue Status' side-by-side. Make a connection between the **queue out** terminal of 'Obtain Queue' and the **queue** terminal of 'Dequeue Element' inside the consumer loop. This will create a tunnel on the consumer loop. The **element** output terminal on 'Dequeue Element' is wired to the analysis code just as the Local Variable in the second loop. Create an indicator to show the number of elements in the queue and wire the error cluster to the conditional stop. This portion of the consumer loop Block Diagram is shown below:



The consumer loop will only execute when data is present in the queue, so it does not require a Wait function. When the producer loop releases the queue, an error will appear in the consumer loop that stops it.

Both the Local Variable loop and consumer loop are trying to process the same temperature data stream. When you run this VI, however, you should observe that the consumer loop counts alarm events that the Local Variable loop misses. (Check that the temperature range for triggering the alarm is the same in both loops) Depending on the rate at which the LED alarm is activated, temperature data may begin accumulating in the queue as shown on the corresponding indicator. This data will eventually get processed by the consumer loop, but it is completely lost to the Local Variable Loop (middle loop). You may also observe that this loop may count the same alarm event twice. You can characterize the performance of these looping operations by varying the alarm temperature range along with the polling rates (set by the Wait functions).