LabVIEW™ Core 2 Exercises

Course Software Version 2014
November 2014 Edition

Trademarks

Refer to the *NI Trademarks and Logo Guidelines* at ni.com/trademarks for more information on National Instruments trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group.

TETRIX by Pitsco is a trademark of Pitsco, Inc.

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

EtherCAT® is a registered trademark of and licensed by Beckhoff Automation GmbH.

CANopen® is a registered Community Trademark of CAN in Automation e.V.

DeviceNet™ and EtherNet/IP™ are trademarks of ODVA.

Go!, SensorDAQ, and Vernier are registered trademarks of Vernier Software & Technology. Vernier Software & Technology and vernier.com are trademarks or trade dress.

Xilinx is the registered trademark of Xilinx, Inc.

Taptite and Trilobular are registered trademarks of Research Engineering & Manufacturing Inc.

FireWire® is the registered trademark of Apple Inc.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox™ and Target Language Compiler™ are trademarks of The MathWorks, Inc.

Tektronix®, Tek, and Tektronix, Enabling Technology are registered trademarks of Tektronix, Inc.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc.

The ExpressCard™ word mark and logos are owned by PCMCIA and any use of such marks by National Instruments is under license.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products/technology, refer to the appropriate location: **Help»Patents** in your software, the patents.txt file on your media, or the *National Instruments Patent Notice* at ni.com/patents.

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments website at ni.com/info and enter the Info Code feedback.

Exercise 1-1 Weather Station UI VI with Local Variables

Goal

Use a local variable to write to and read from a control.

Scenario

You have a LabVIEW project that implements a temperature weather station. The weather station acquires a temperature every half a second, analyzes each temperature to determine if the temperature is too high or too low, then alerts the user if there is a danger of a heat stroke or freeze. The VI logs the data if a warning occurs.

Two front panel controls determine the setpoints—the temperature upper limit and the temperature lower limit. However, nothing prevents the user from setting a lower limit that is higher than the upper limit.

Use a local variable to set the lower limit less than the upper limit if the user sets a lower limit that is higher than the upper limit.

Design

Your task is to modify a VI so that the lower limit is set less than the upper limit when necessary.

State Definitions

The following table describes the states in the state machine.

State	Description	Next State
Acquisition	Set time to zero, acquire data from the temperature sensor	Analysis
Analysis	Read front panel controls and determine warning level	Data Log if a warning occurs, Time Check if no warning occurs
Data Log	Log the data in a tab-delimited ASCII file	Time Check
Time Check	Check whether time is greater than or equal to .5 seconds	Acquisition if time has elapsed, Time Check if time has not elapsed

Changing the value of the lower temperature limit control should happen after the user has entered the value but before the value determines the warning level. Therefore, make the modifications to the VI in the Acquisition or Analysis state, or place a new state between the two.

Before determining which option to use, review the content of the Acquisition and Analysis states:

1. Open `Weather Station.lvproj` in the `<Exercises>\LabVIEW Core 2\Weather Station` directory.
2. Open **Weather Station UI.vi** from the **Project Explorer** window.
3. Review the contents of the Acquisition and Analysis states, which correspond to the Acquisition and Analysis cases of the Case structure.

Design Options

You have three different design options for modifying this project.

Option	Description	Benefits/Drawbacks
1	Insert a Case structure in the Acquisition state to reset the controls before a local variable writes the values to the cluster.	Poor design: the acquisition state has another task added, rather than focusing only on acquisition.
2	Insert a new state in the state machine that checks the controls and resets them if necessary.	Ability to control when the state occurs.
3	Modify the Temperature Warnings subVI to reset the controls.	Easy to implement because functionality is already partially in place. However, if current functionality is used, one set of data is always lost when resetting the lower limit control.

This exercise implements Option 2 as a solution.

New State Definitions for Option 2

Read the upper and lower limit controls in the Range Check state, instead of the Analysis state. Table describes the states in the new state machine. You have already implemented the Acquisition, Analysis, Data Log, and Time Check states. In this exercise, you add the Range Check state. The VI reads the **Upper Limit** and **Lower Limit** controls in the Range Check state, instead of the Analysis state. The Range Check state also resets the **Lower Limit** control lower than the upper limit if the **Upper Limit** control is less than the lower limit.

Table 1-3. State Descriptions for Option 2

State	Description	Next State
Acquisition	If you have hardware installed, acquire data from the temperature sensor on channel AI0 and read front panel controls. Otherwise, simulate this acquisition.	Range Check
Range Check	Read front panel controls and set the lower limit to 1 less than the upper limit if the upper limit is less than the lower limit.	Analysis
Analysis	Determine warning level.	Data Log if a warning occurs Time Check if no warning occurs
Data Log	Log the data in a tab-delimited ASCII file	Time Check
Time Check	Check whether time is greater than or equal to .5 seconds	Acquisition if time has elapsed Time Check if time has not elapsed

Implementation

1. Open `Weather Station.lvproj` from the `<Exercises>\LabVIEW Core 2\Weather Station` directory.
2. Add the Range Check state to the state machine.
   a. In the **Project Explorer** window, navigate to **Supporting Files** and open **Weather Station States.ctl**.
   b. Right-click the **States** control and select **Edit Items** from the shortcut menu.
   c. Insert an item and modify the item to match Table 1-4. Be careful not to add an empty listing.

Table 1-4. States Enumerated Control

Item	Digital Display
Acquisition	0
Range Check	1
Analysis	2
Data Log	3
Time Check	4

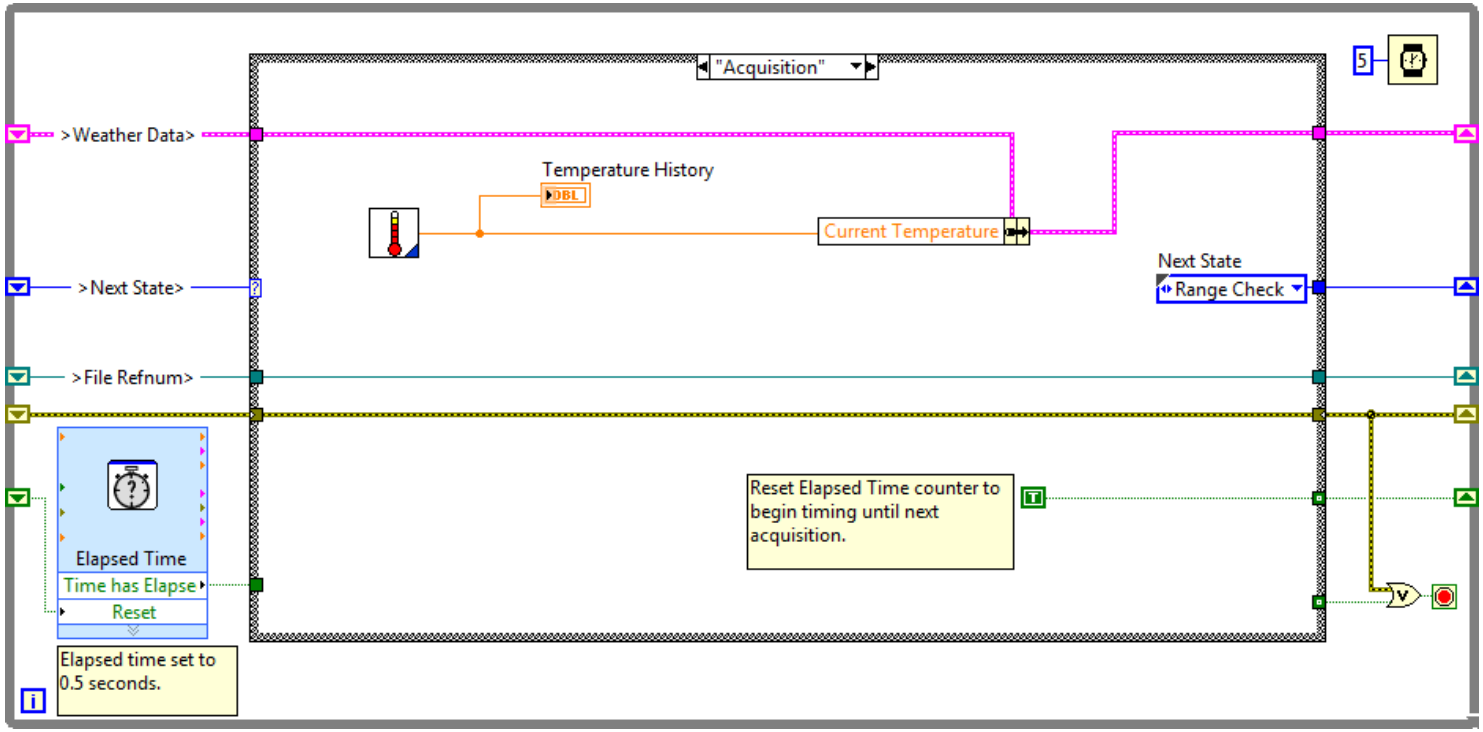
   d. Save and close the control.
   e. Open **Weather Station UI.vi** from the **Project Explorer** window.

Note The Run arrow is broken as there as some unwired terminals. You complete the wiring of these terminals in the following steps.

   f. On the block diagram of the Weather Station UI VI, right-click the state machine Case structure and select **Add Case for Every Value** from the shortcut menu. Because the enumerated control has a new value, a new case appears in the Case structure.

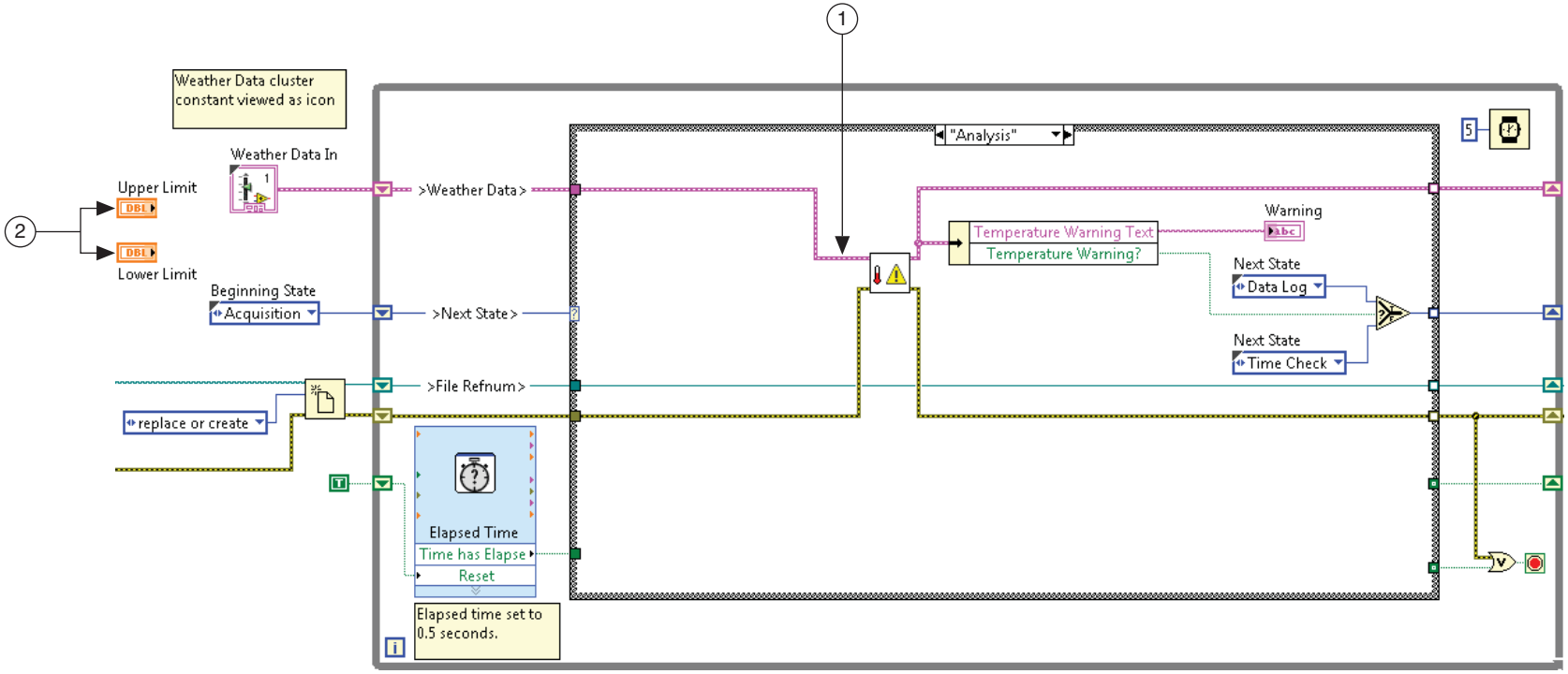
3. Set the **Next State** enum in the Acquisition case to Range Check as shown in Figure 1-1.

Figure 1-1. Weather Station UI VI with Local Variables—Completed Acquisition State



4. Modify the Analysis case as shown in Figure 1-2.

Figure 1-2. Weather Station UI VI with Local Variables—Completed Analysis State

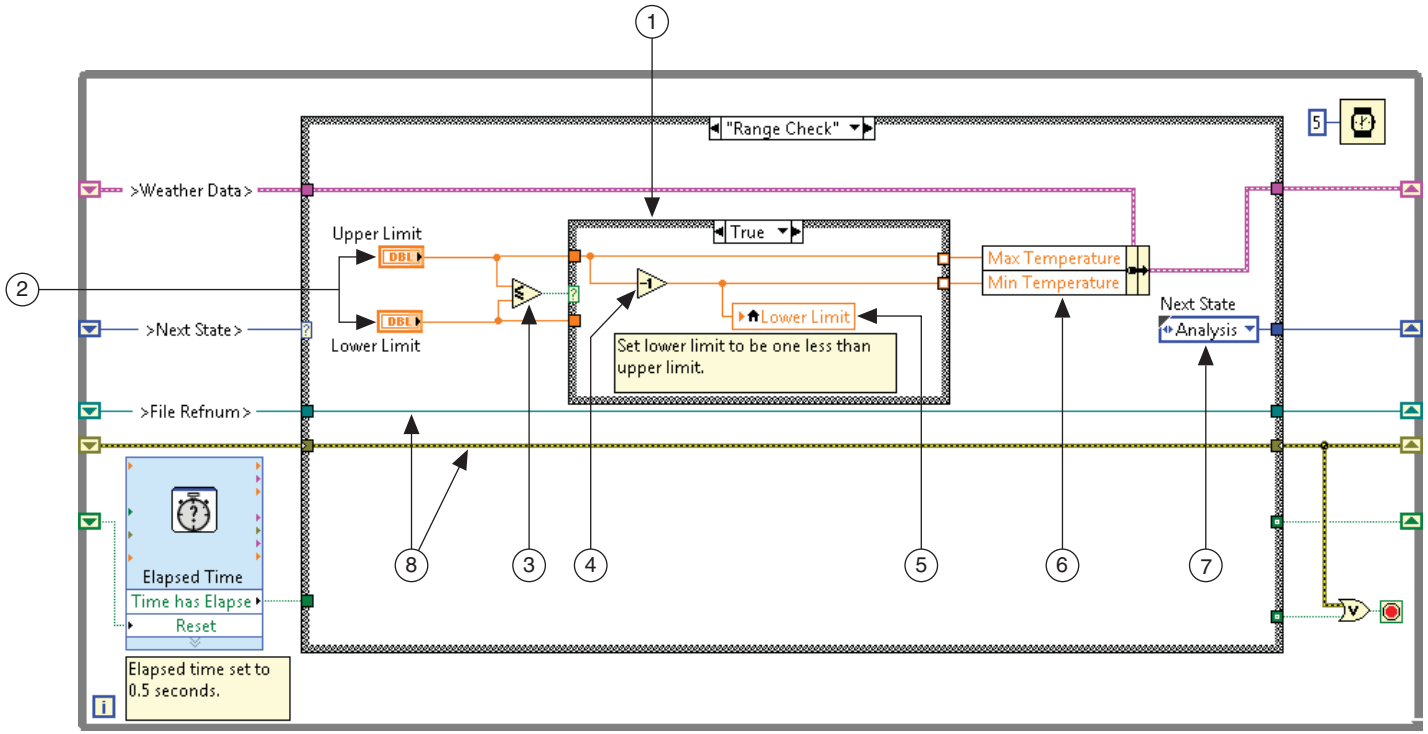


1 Delete the Bundle By Name function and wire the Weather Data wire directly to the Temperature Warnings VI. Press <Ctrl-B> to delete the broken wires from the **Upper Limit** and **Lower Limit** controls.
2 Move the **Upper Limit** and **Lower Limit** controls outside the While Loop.

5. Complete the Range Check True state as shown in Figure 1-3.

   When the **Upper Limit** control value is less than or equal to the **Lower Limit** control value, use a local variable to write the value, upper limit - 1, to the **Lower Limit** control.

Figure 1-3. Weather Station UI VI with Local Variables—Range Check True Case

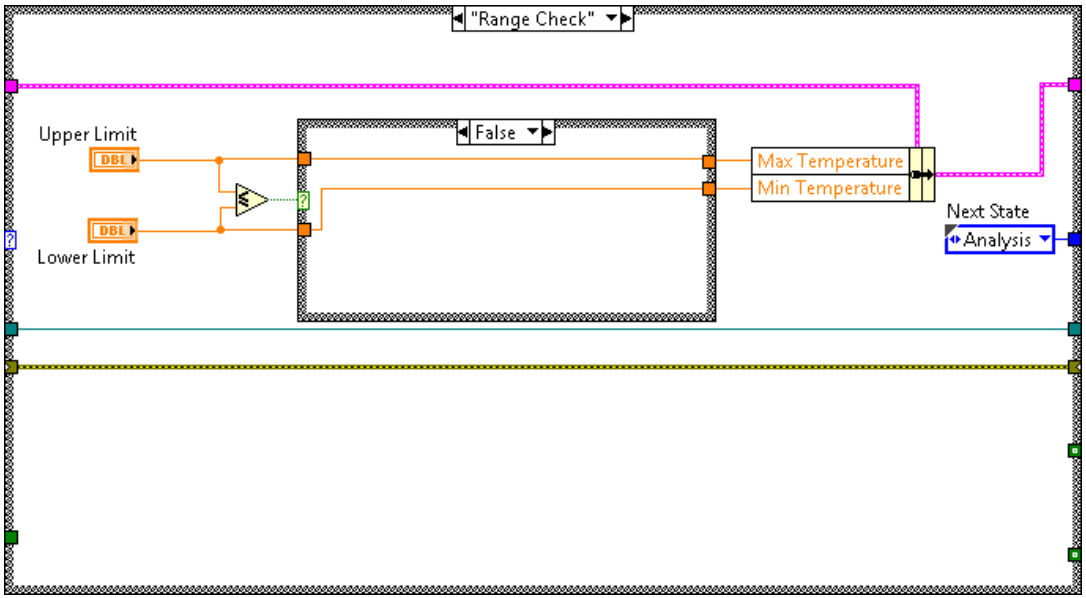


1 **Case Structure**—Place a Case structure inside the Range Check case.
2 Move the **Upper Limit** and **Lower Limit** controls into the Range Check case.
3 **Less or Equal?**—Compares upper limit and lower limit values. Because the Less or Equal? function is wired to the case selector of the inner Case structure, when the upper limit is less than or equal to the lower limit, the True case executes.
4 **Decrement**—Subtracts 1 from the value of the **Upper Limit** control so the True case writes a lower value to the **Lower Limit** control.
5 **Lower Limit** local variable—Right-click the **Lower Limit** control and select **Create»Local Variable**. Place the local variable in the True case.
6 **Bundle by Name**—Expand to display two elements and use the Operating tool to select the correct cluster elements.
7 **Next State**—Create a copy of the Beginning State type def control and set the next state to **Analysis**.
8 **Complete Wiring**—Wire the File Refnum and the error wire.

6. Create the Range Check False state as shown in Figure 1-4.

   If the **Upper Limit** control value is not less than or equal to the **Lower Limit** control value, the False case executes and the values are passed, unchanged, through to the temperature cluster.

Figure 1-4. Weather Station UI VI with Local Variables—Range Check False State



7. Save the VI and the project.

Test

1. Run the VI.
2. Enter a name for the log file when prompted.
3. Enter a value in the **Upper Limit** control that is less than the value in the **Lower Limit** control. Does the VI behave as expected?
4. Stop the VI when you are finished.
5. Close the VI and the project.

End of Exercise 1-1

Exercise 2-1 Concept: Comparing Queues With Local Variables

Goal

In this exercise, you run and examine a prebuilt producer/consumer design pattern VI that transfers data that a producer loop generates to consumer loops using local variables and queues.

Description

The following sections describe how the Queues vs Local Variables VI does the following.

- Creates a queue.
- Queues data that the producer loop generates.
- Dequeues data in the consumer loop.
- Waits for the queue to empty before exiting the VI.
- Uses local variables to read and display data from the producer loop.

Implementation

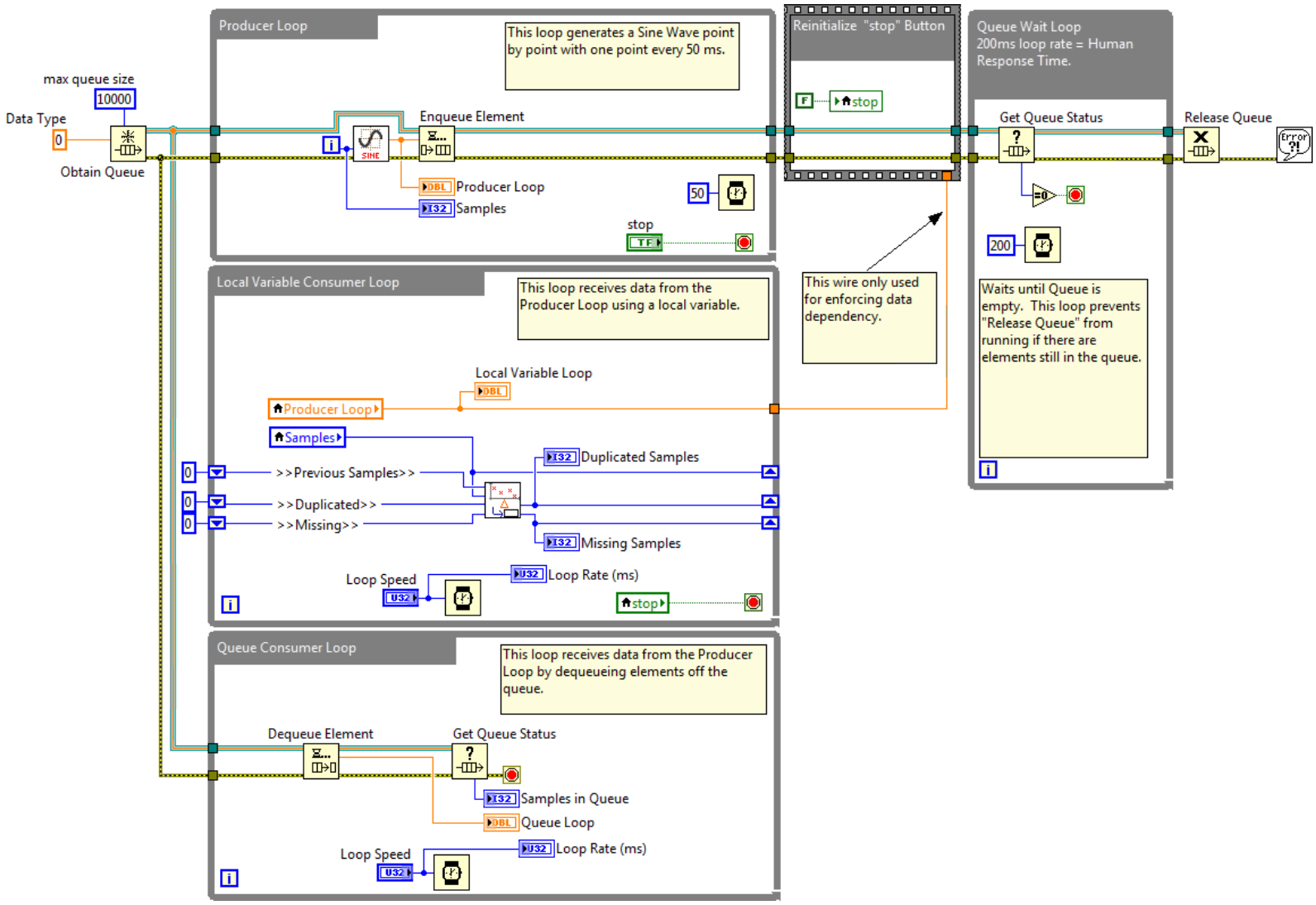
1. Open `Queues vs Local Variables.lvproj` in the `<Exercises>\LabVIEW Core 2\Queues versus Local Variables` directory.
2. Double-click **Queues vs Local Variables.vi** in the **Project Explorer** window to open the VI. The front panel of this VI is shown in Figure 2-1.

Figure 2-1. Front Panel of the Queues vs Local Variables VI

3. Run the VI. The Producer Loop generates data and transfers it to each consumer loop using a local variable and a queue. Observe the behavior of the VI when the consumer loops are set to the same speed as the producer loop.
4. Stop the VI.

5. Display and examine the block diagram for this VI. The following sections describe parts of this block diagram in more detail.

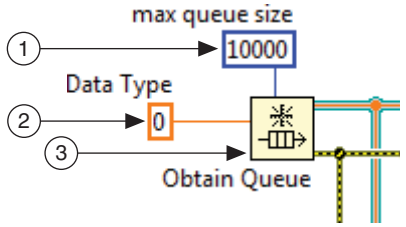
Figure 2-2. Block Diagram of the Queues vs Local Variables VI



Creating a Queue

You create the queue with code shown in Figure 2-3. This code is located to the left of the producer loop.

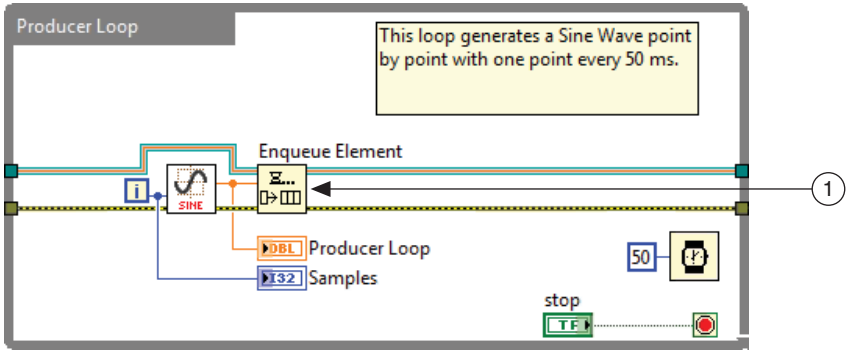
Figure 2-3. Creating the Queue



1 Max queue size—Sets the maximum number of elements that the queue can hold.
2 Double-precision Numeric constant—Wired to the **element data type** input, specifies the type of data you want the queue to contain.
3 Obtain Queue—Creates the queue and defines the data type.

Queueing Data Generated by the Producer Loop

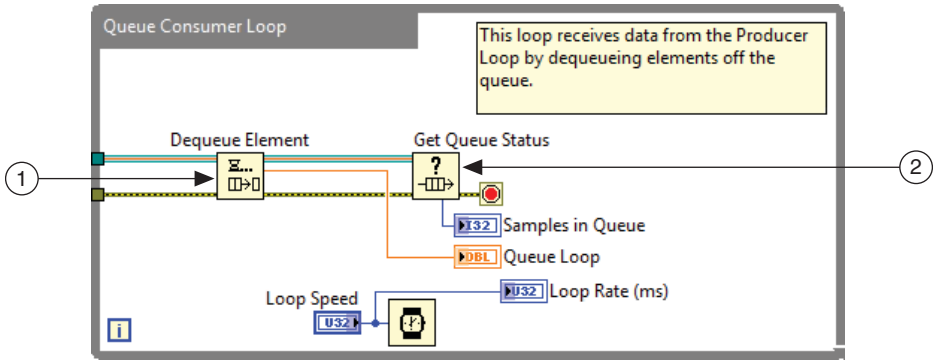
Figure 2-4. Queuing Data the Producer Loop Generates



1 Enqueue Element—Adds each data element the Generate Sine VI generates in the Producer Loop to the back of the queue.

Dequeuing Data from the Producer Loop inside the Queue Consumer Loop

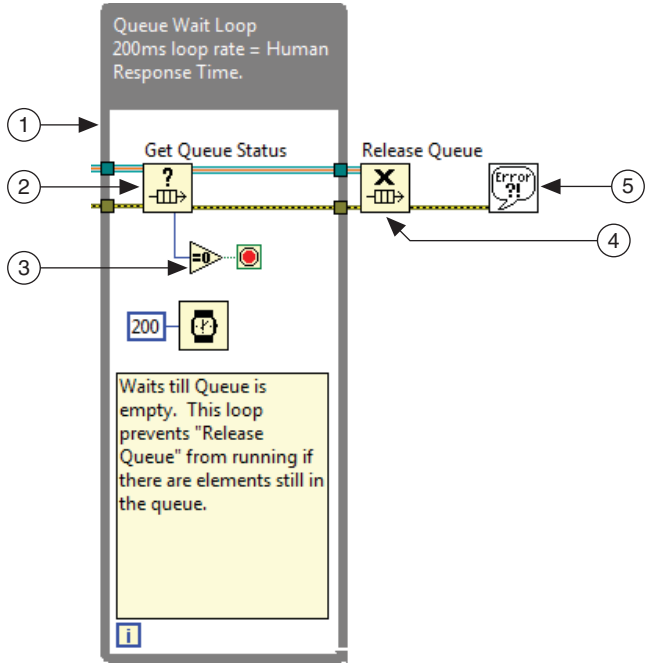
Figure 2-5. Dequeuing Data inside the Consumer Loop



1 Dequeue Element—Removes an element from the front of the queue and sends the data element to the Queue Loop waveform chart.
2 Get Queue Status—Indicates how many elements remain in the queue. In order to process these data elements, you must execute the Queue Consumer Loop faster than the Producer Loop, or continue to process after the Producer Loop stops.

Waiting for the Queue to Empty

Figure 2-6. Waiting for the Queue to Empty

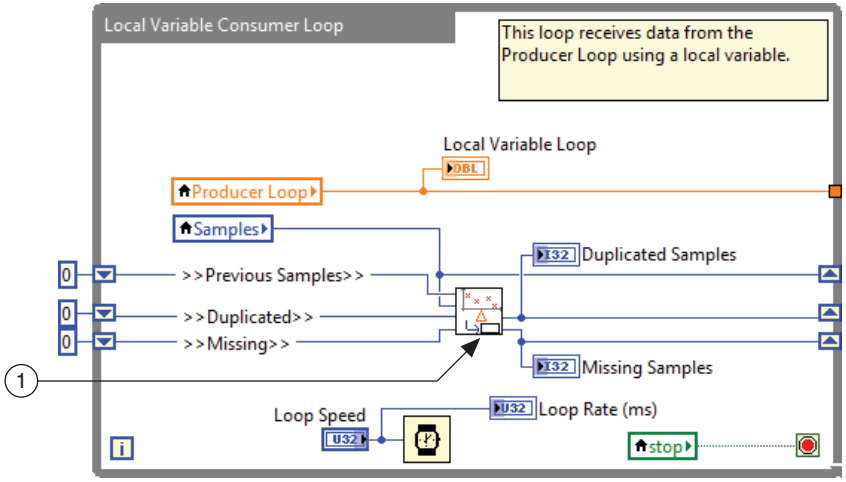


1 While Loop—Waits for the queue to empty before stopping the VI. Refer to this While Loop as the Queue Wait Loop.
2 Get Queue Status—Returns information about the current state of the queue, such as the number of data elements currently in the queue.
3 Equal To 0?—Wired to the stop condition of the Queue Wait Loop, checks if the queue is empty.
4 Release Queue—Releases and clears references to the queue.
5 Simple Error Handler—Reports any error at the end of execution.

Local Variable Consumer Loop

The Producer Loop also writes the generated sine wave data to a local variable while the Local Variable Consumer Loop periodically reads out the sine wave data from the same local variable.

Figure 2-7. Local Variable Consumer Loop



1 Update Counters—Updates the counters for missed or duplicated samples.

Test

Local Variable Consumer Loop

1. Switch to the front panel of the Queues vs Local Variables VI.
2. Run the VI.
3. Select different speeds for the Local Variable Consumer Loop and observe the Local Variable Consumer Loop chart and the results generated on the **Missing Samples** indicator or **Duplicated Samples** indicator.

   ☐ Ensure that the **Loop Speed** selected is **Same as Producer Loop** and observe the Producer Loop chart and the Local Variable Consumer Loop chart. A race condition may occur resulting in missed points or duplicated data.

   ☐ Select **2x as Producer** from the pull-down menu of the **Loop Speed** control and observe the Local Variable Consumer Loop chart. A race condition occurs because data is consumed faster than it is produced, allowing the local variable to read the same value multiple times.

- ☐ Select **1/2 as Producer** from the pull-down menu of the **Loop Speed** control and observe the Local Variable Consumer Loop chart. A race condition occurs because data is produced faster than it is consumed. The data changes before the local variable has a chance to read it.
- ☐ Select the remaining options available from the pull-down menu of the **Loop Speed** control and observe the data retrieval.

4. Stop the VI.

Data transfer between two non-synchronized parallel loops using local variables causes a race condition. This occurs when the Producer Loop is writing a value to a local variable while the Local Variable Consumer Loop is periodically reading out the value from the same local variable. Because the parallel loops are not synchronized, the value can be written before it has actually been read or vice versa resulting in data starvation or data overflow.

Queue Consumer Loop

1. Run the VI.
2. Select the loop time speed of the Queue Consumer Loop and observe the Queue Consumer Loop waveform chart and the results generated on the **Samples in Queue** indicator.
   - ☐ Ensure that the **Loop Speed** selected is **Same as Producer** and observe the value of the **Samples in Queue** indicator. The value should remain zero. Hence with queues, you will not lose data when the producer and consumer loops are executing at the same rate.
   - ☐ Select **2x as Producer** from the pull-down menu of the **Loop Speed** control and observe the value of the **Samples in Queue** indicator. The value should remain zero, because the Dequeue Element function in the Queues Consumer Loop controls the maximum speed of the loop. If the queue is empty, the Dequeue Element function waits for a data element to enter the queue, essentially pausing the loop.
   - ☐ Select **1/2 as Producer** from the pull-down menu of the **Loop Speed** control and observe the value of the **Samples in Queue** indicator. The data points accumulate in the queue. You need to process the accumulated elements in the queue before reaching the maximum size of the queue to avoid data loss.
   - ☐ Select the remaining options available from the pull-down menu of the **Loop Speed** control and observe the synchronization of data transfer between the producer loop and the consumer loop using queues.
3. Stop the VI.

When the Producer Loop and Queue Consumer Loop run at the same speed, the number of elements in the queue remains unchanged. When the Queue Consumer Loop runs slower, the queue quickly backs up and the Producer Loop must wait for the Queue Consumer Loop to remove the elements. When the Queue Consumer Loop runs faster, the queue quickly empties and the consumer loop must wait for the Producer loop to insert elements. Hence queues synchronize the data transfer between the two independent parallel loops and thus avoid loss or duplication of data.

4. Close the VI. Do not save changes.

End of Exercise 2-1

Exercise 3-1 Group Exercise: Producer/Consumer Design Pattern

Goal

As a group, explore the Producer Consumer template.

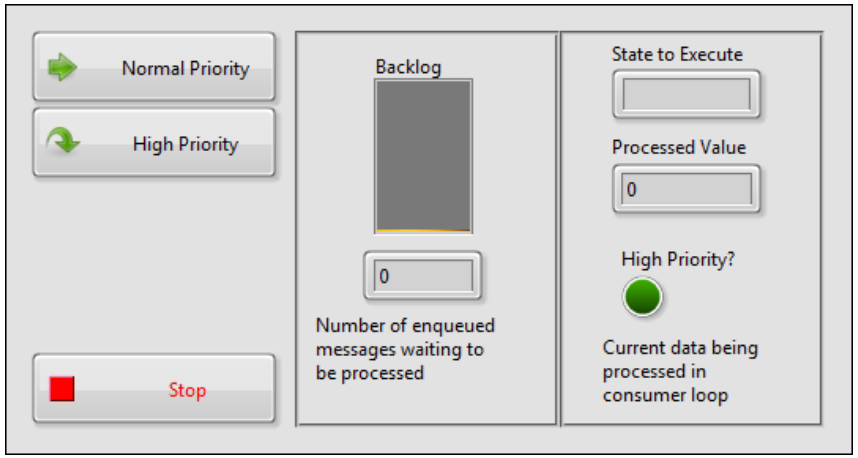
Scenario

You have a VI that uses the Producer/Consumer design pattern to process messages. The consumer rate is slower than the producer and therefore, a backlog is created. The VI clears messages from the backlog in the order the messages are received, until a high priority message is introduced. High priority messages are processed as soon as they are received and then the VI resumes processing normal priority messages.

Implementation

1. Open the Producer Consumer project located in the `<Exercises>\LabVIEW Core 2\Producer Consumer - Event` directory, and then open Main.vi, shown in Figure 3-1, from the project.

Figure 3-1. Producer Consumer - Events Main VI Front Panel



2. Run the VI.
3. Click the **Normal Priority** button several times.
   - ☐ Notice the **State to Execute** indicator says **`Normal Priority`**.
   - ☐ Notice that **Processed Value** increases.
   - ☐ Notice the **High Priority?** Boolean indicator is off.

4. Rapidly click the **Normal Priority** button to create a backlog.

   ☐ Notice that the **Backlog** indicator increases.

   ☐ Notice that the **Backlog** decreases by 1 every second.

5. Click the **High Priority** button.

   ☐ Notice that **State to Execute** says **`High Priority`** and the **Processed Value** indicator says **`1000`**.

   ☐ Notice the **High Priority?** Boolean indicator is on.

6. Watch the flow of data on the block diagram.

   ☐ Select **Window»Tile Left and Right**.

   ☐ Click the **Highlight Execution** button on the block diagram and then run the VI and watch what happens when you click the **Normal Priority** button.

   ☐ Click the **High Priority** button.

   ☐ Notice that the Wait (ms) in the Default state of the Consumer loop is set to 1000. This is what causes the processing of one message per second.

7. Disable Highlight Execution.

8. Click the **Normal Priority** button several times to create a backlog.

9. While the backlog is present click the **Stop** button.

   ☐ Notice that the VI stops even though the backlog has not been processed.

   ☐ The Enqueue Element at Opposite End function caused this to occur. If the Stop should occur after all messages in the backlog are processed, then this function would be replaced with a regular Enqueue Element function.

10. Inject an error and see what happens.

    ☐ Delete the error cluster wire running through the Default state of the Consumer loop.

    ☐ Right-click the error output tunnel of the Case Structure and select **Create»Constant**.

    ☐ In the new error cluster constant, set the value of the **Status** Boolean to True.

- ☐ Run the VI again and click **Normal Priority**.
- ☐ Notice that the VI does not behave properly anymore. This is because an error in the Consumer loop has caused the Consumer loop to shut down. Because the Consumer loop was doing the bulk of the work, the VI does not behave correctly. The Producer loop is still running.

11. Close the VI and the project. Do not save your changes.

This VI does not include any error handling. You modify a version of this VI in another exercise to enable error handling so the VI shuts down if an error occurs.

End of Exercise 3-1

Exercise 3-2 User Access Level

Goal

To create a gating application, using a functional global variables design pattern, which restricts user access to certain features based on different user access levels.

Scenario

You need to create an application in which some features are not available to all users. You create a finite number of user access levels and assign an appropriate user level to various users. You use a functional global variable design pattern to check for different access levels.

Design

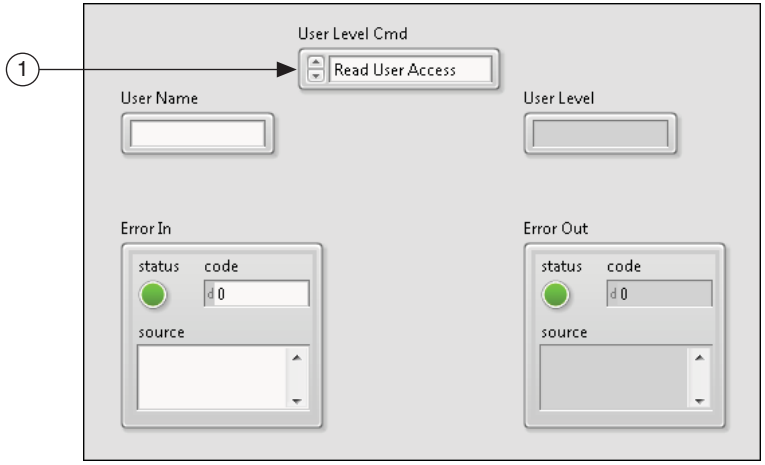
The following table describes the different actions you need to handle so you can implement user access control. In this exercise, you create a custom control to handle these items.

Action	Description
Read User Access Level File	Read information about authorized users and their access levels from a specified file and stores this access information in memory.
Set Current User	Specify the name of the current user. Store the access level for that user in application memory.
Get User Access Level	Retrieve the access level of the current user from memory so that the application can determine if a user has access to a certain feature.

Implementation

1. Open the `User Level FGV.lvproj` project from `<Exercises>\LabVIEW Core 2\FGVs`.
2. Open the **User Level FGV** folder in the **Project Explorer** window and then open **User Access Level FGV.vi**. The User Access Level FGV VI already contains several items on the front panel, an icon, and connector pane.
3. Create a type-defined enum control and modify the front panel as shown in Figure 3-2.

Figure 3-2. User Access Level FGV Front Panel

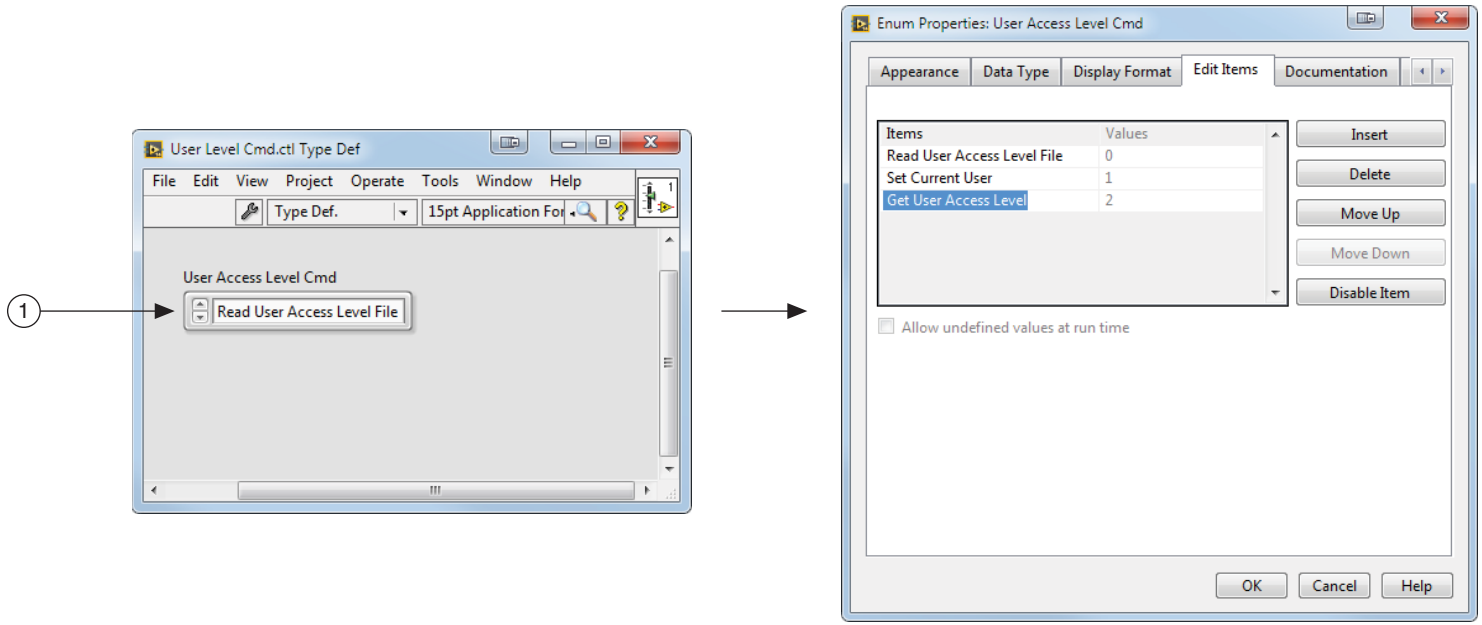


1 Enum (Silver)—Place an Enum (Silver) control on the front panel. Right-click the enum control and select **Make Type Def**.

4. Right-click the **Use Level Cmd** enum control and select **Open Type Def** from the shortcut menu.

5. Add the three actions listed in the Design section of this exercise to the **User Level Cmd** type definition as shown in Figure 3-3.

Figure 3-3. Editing the User Access Level Cmd Enum



1 Enum—Right-click and select **Edit Items**.

6. Save the enum as `User Level Cmd.ctl` in `<Exercises>\LabVIEW Core 2\FGVs\User Level FGV` and close the custom control editor window.

7. From the User Access Level FGV VI front panel, assign a terminal from the top-level of the User Access Level FGV VI connector pane to the **User Level Cmd** control as shown in Figure 3-4.

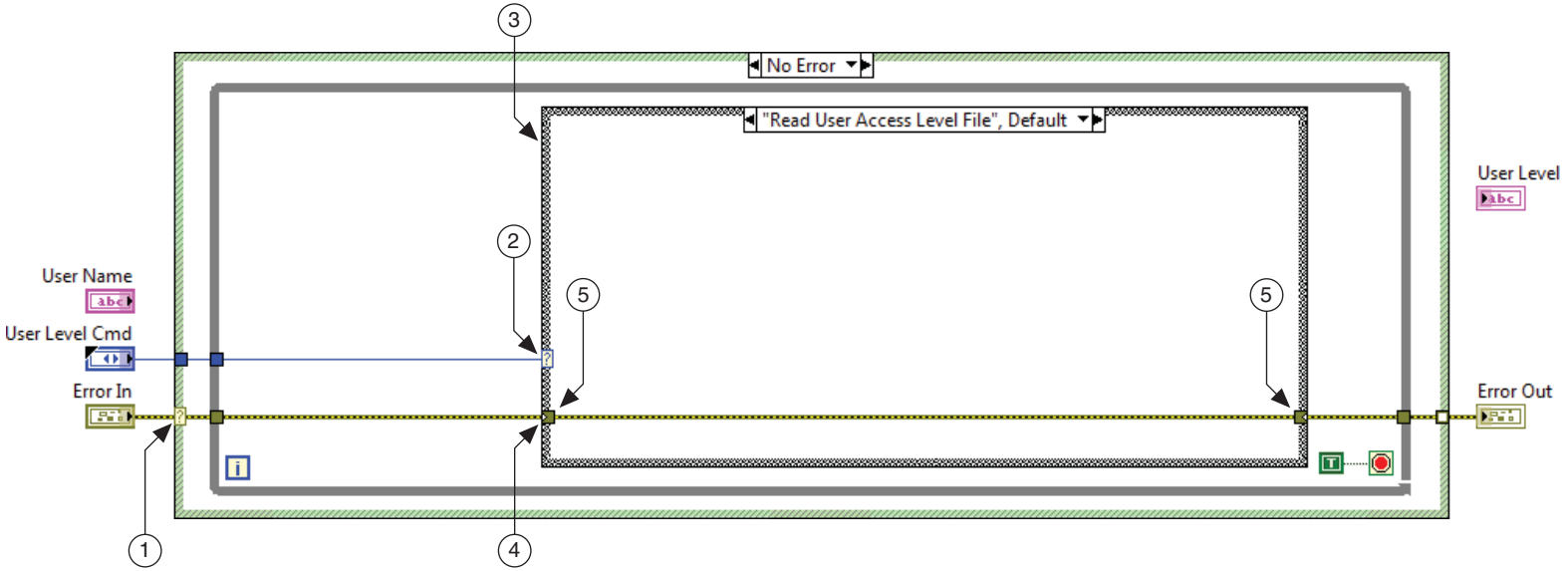
Figure 3-4. Assign the User Level Cmd Control to a Connector Pane Terminal

1 Connector pane terminal—Click the terminal indicated by the arrow, then click the **User Level Cmd** control to assign the control to the connector pane terminal.

Right-click the connector pane terminal and select **This Connection Is»Required**. By making this terminal required, an application must provide a value to the User Level Cmd input when you use the User Access Level VI in another VI.

8. Create the framework for the functional global variable design by completing the block diagram as shown in Figure 3-5.

Figure 3-5. Creating the Functional Global Variable Design Framework

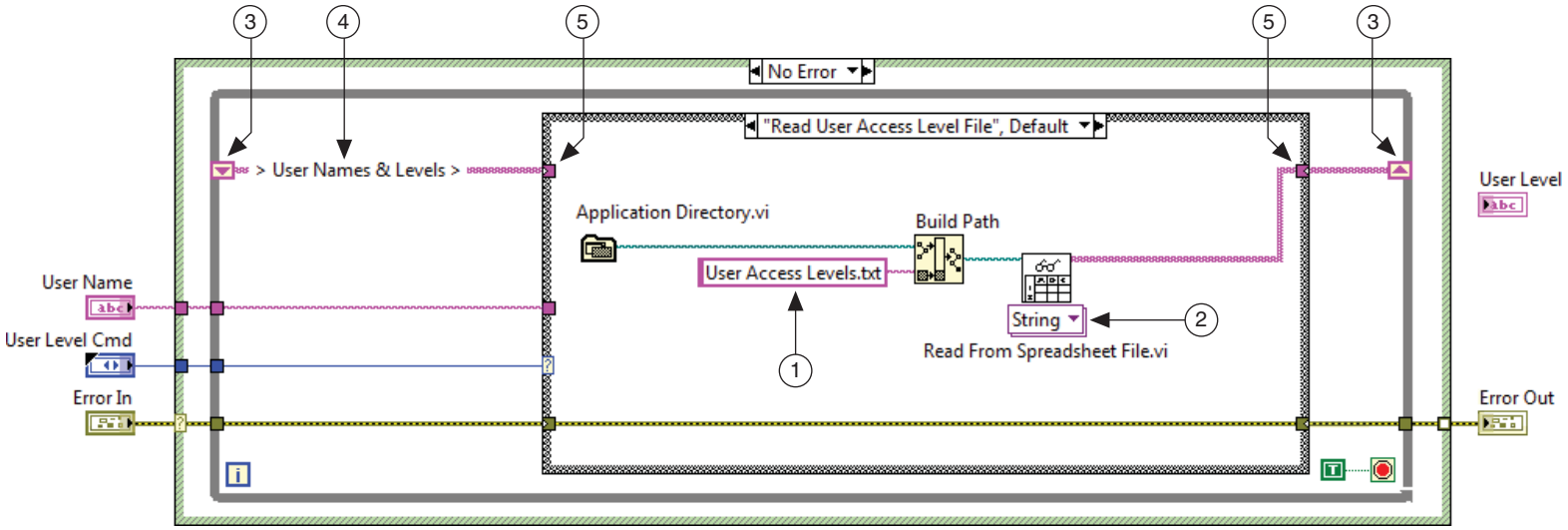


1 Case selector—Wire the **Error In** cluster to the case selector of the outer Case structure to set the error and no error cases.
2 Case selector—Wire **User Level Cmd** to the case selector.
3 Case structure—Right-click the Case structure and select **Add Case for Every Value** from the shortcut menu and then select "Read User Access Level File", Default.
4 Error In/Error Out—Wire **Error In** to **Error Out** through the default case.
5 Wire Error In/Error Out through all cases—Right-click the output tunnel and select **Linked Input Tunnel»Create & Wire Unwired Cases**. When the cursor changes to a wiring tool, click on the left-side input tunnel. Small white triangles inside the input and output tunnels indicate the link.

Tip Use the Add Case for Every Value option when you know that each case diagram is significantly different. If cases contain similar subdiagrams, use the Duplicate Case option instead. After you duplicate a case, you can modify and rename it.

9. Complete the Read User Access Level File case as shown in Figure 3-6.

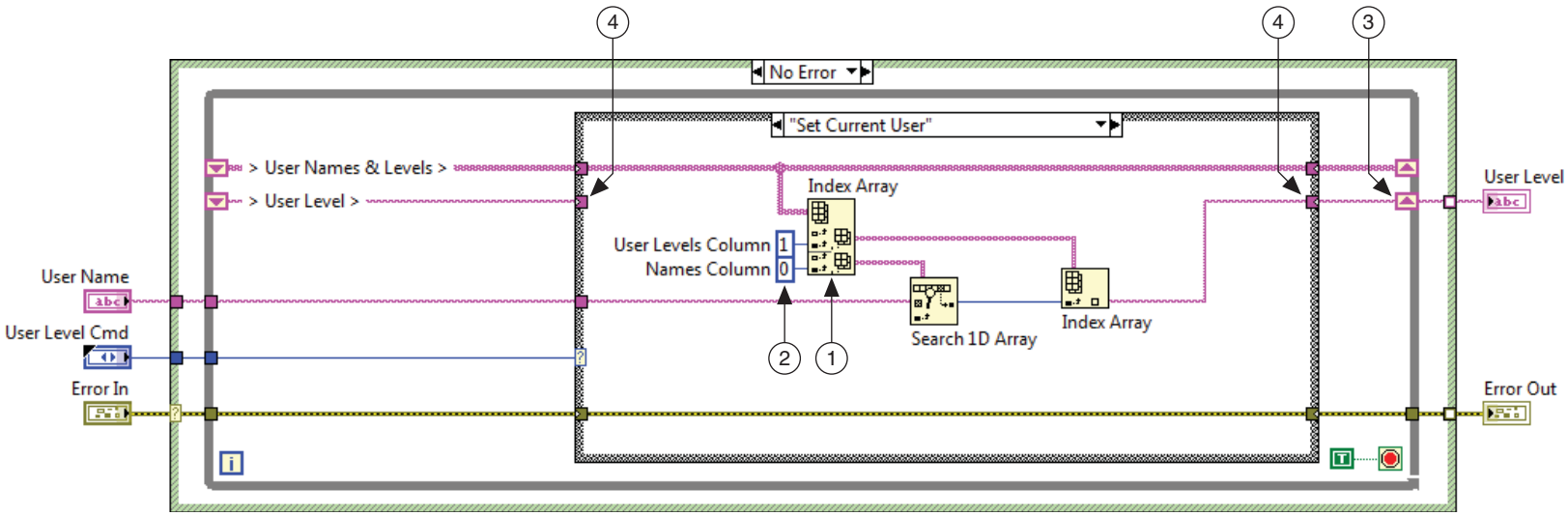
Figure 3-6. Configuring the Read User Access Level File Case



1 The path to `User Access Levels.txt` is relative to where you save the User Access Level FGV VI. In this case, the text file is in a parallel directory.
2 Read From Spreadsheet File VI—Click the polymorphic VI selector and select **String**. Wire the **all rows** output through a tunnel on the Case Structure to the While Loop.
3 Shift register—Right-click the tunnel and select **Replace with Shift Register**. Click the left side of the While Loop to complete the shift register.
4 Label—Right-click the wire and select **Visible Items»Label** to show the label, then type the name `User Names & Levels`.
5 Right-click the output tunnel and select **Linked Input Tunnel»Create & Wire Unwired Cases**, then click the corresponding input tunnel.

10. Complete the Set Current User case as shown in Figure 3-7.

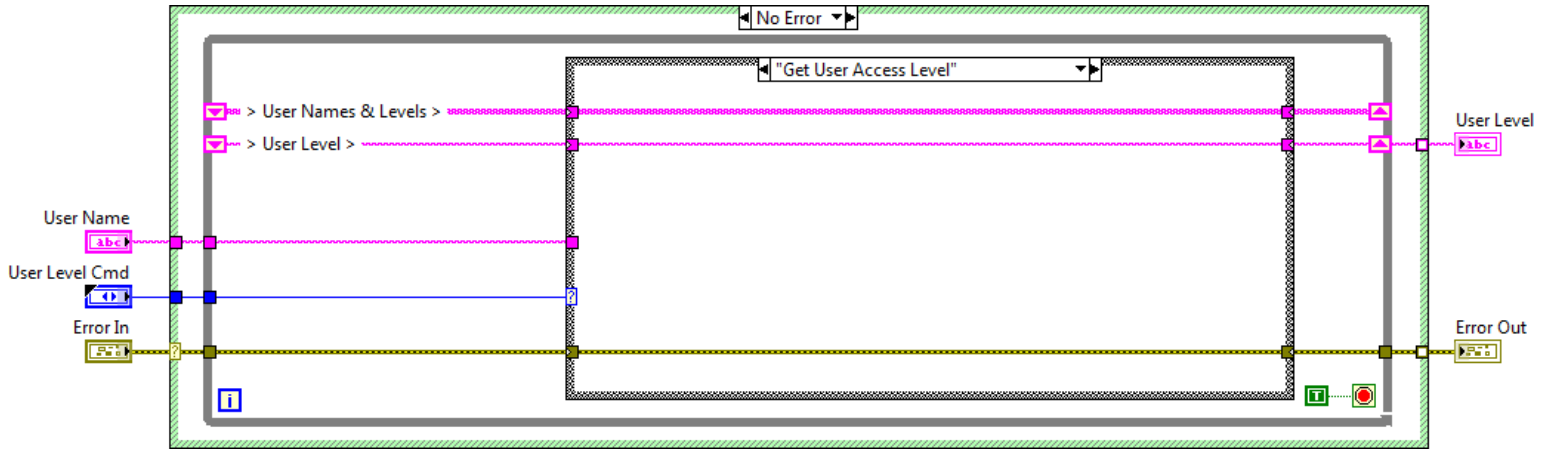
Figure 3-7. Configuring the Set Current User Case



1 Index Array function—Wire the User Names & Levels wire to an Index Array function.
2 User Levels Column and Names Column constants—Create constants for the **index (col)** input of the Index Array function.
3 Shift register—Change the tunnel to shift register.
4 Right-click the output tunnel and select **Linked Input Tunnel»Create & Wire Unwired Cases** then click the corresponding input tunnel.

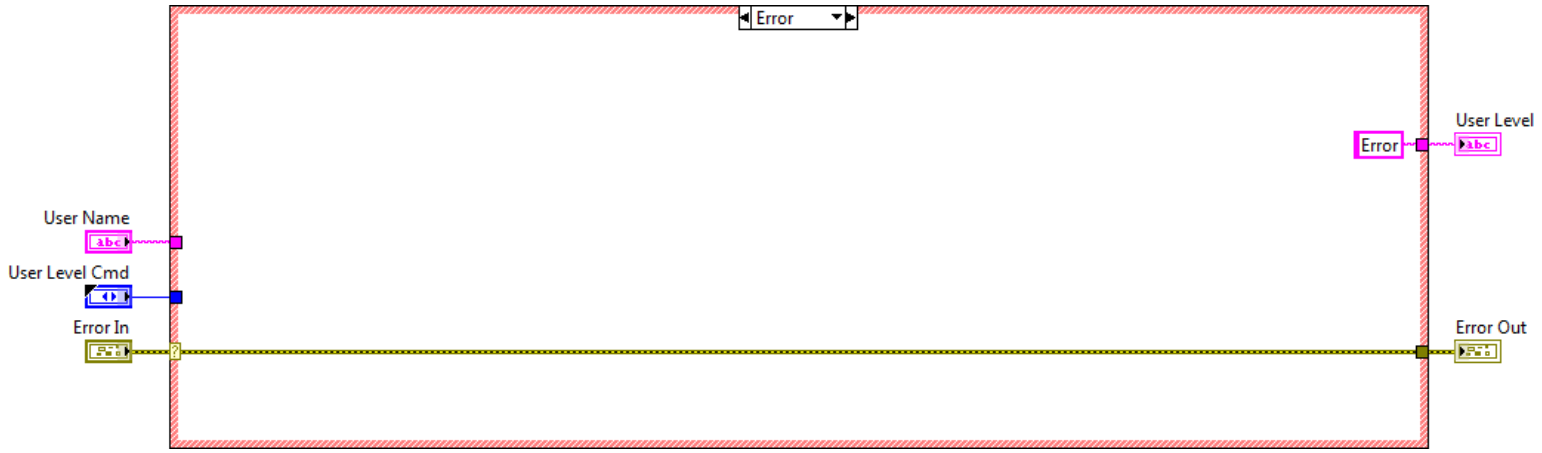
11. Leave the Get User Access Level case as shown in Figure 3-8.

Figure 3-8. Get User Access Level Case



12. Modify the Error Case as shown in Figure 3-9.

Figure 3-9. Error Case



13. Save and close the VI.

Test

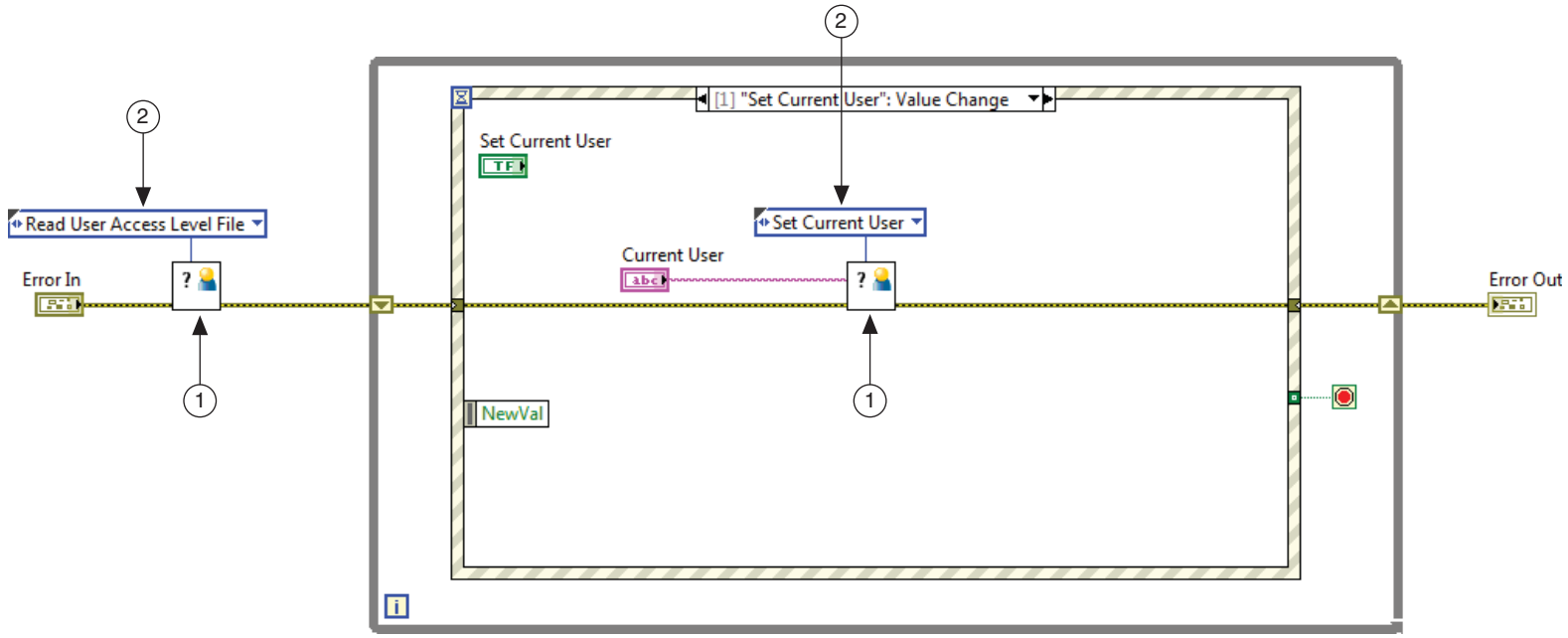
1. From the project, open `User Access Levels.txt` and review the contents of the file. This file contains the names of authorized users and their access levels. You can add additional user names and levels if you want.

2. Double-click **User Access Level FGV Unit Test.vi** in the **Project Explorer** window to open the VI. This VI takes a user name you input, sets the permissions of the VI to the user's access level, and tests whether the proper access level is set.

Note The User Access Level FGV Unit Test VI is broken until you complete step 3 because of the required input you set in step 7 of the implementation.

3. Complete the "Set Current User": Value Change event as shown in Figure 3-10.

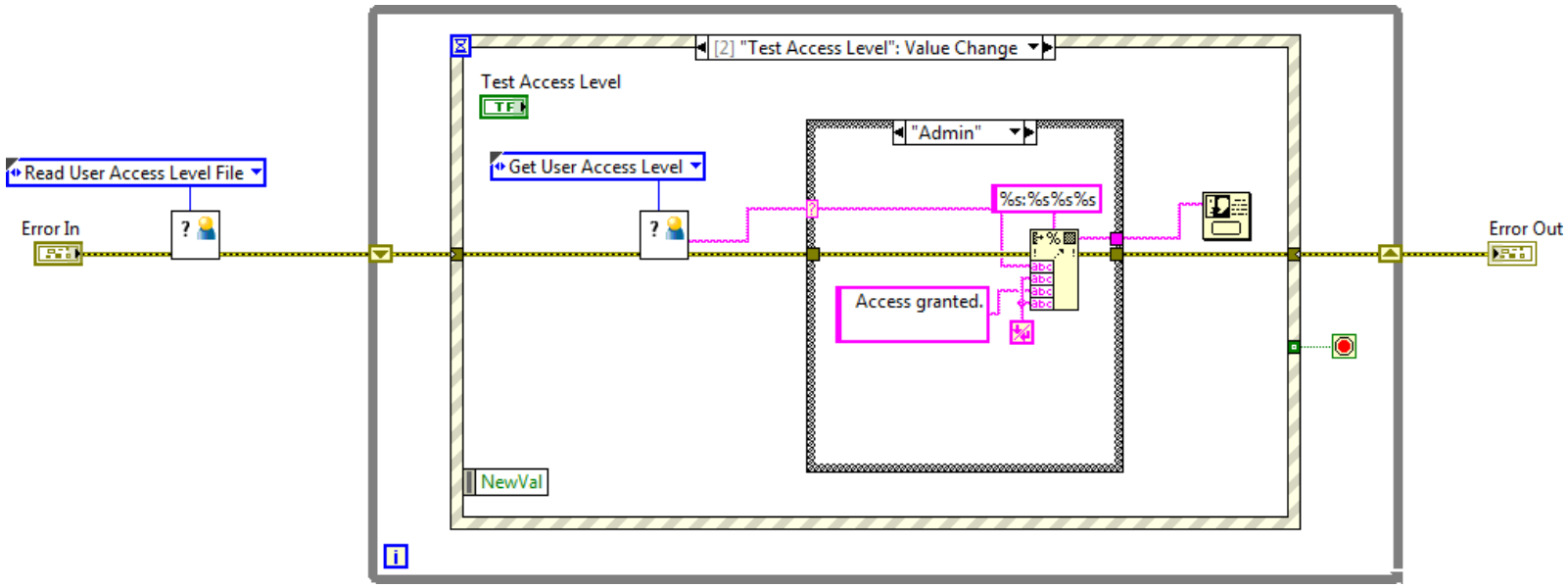
Figure 3-10. Completing the "Set Current User": Value Change Event



1 User Access Level FGV VI—This is the VI you modified in this exercise. It has already been placed on the block diagram.
2 User Level Cmd Constants—Right-click the User Level Cmd input and select **Create»Constant**.

4. Complete the “Test Access Level”: Value Change event case as shown in Figure 3-11.

Figure 3-11. Completing the “Test Access Level”: Value Change Event



5. Run the VI with the following user names and verify the user level is correct by clicking the **Test Access Level** button.

User Name	User Level
John	Operator
Paul	Admin
George	Admin
Ringo	Operator

6. Save and close the project.

End of Exercise 3-2

Exercise 3-3 Producer/Consumer with Error Handling

Goal

Modify the Producer/Consumer template to handle error codes.

Scenario

You need to test error handling in a Producer/Consumer design pattern VI.

Design

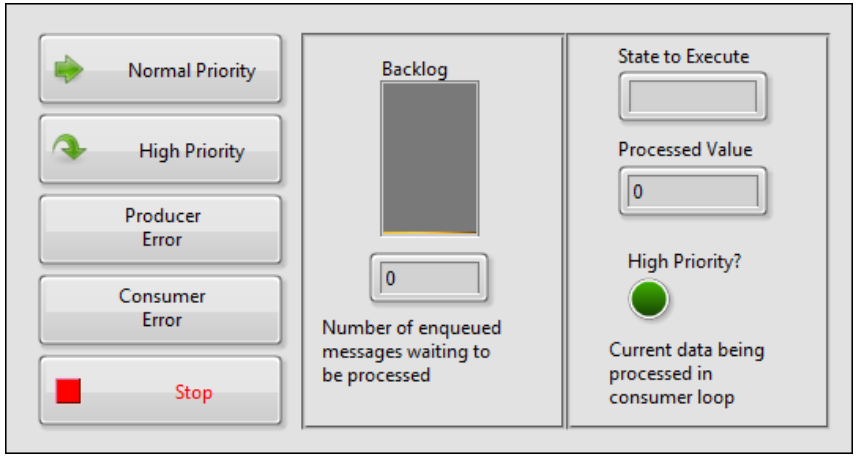
Add buttons to inject simulated errors in the producer loop and the consumer loop. Use Error Ring constants to produce simulated errors. Update the producer loop to handle an error case.

Implementation

1. Open the Producer Consumer project located in the `<Exercises>\LabVIEW Core 2\Producer Consumer - Error` directory, and then open the **Main.vi** from the project.

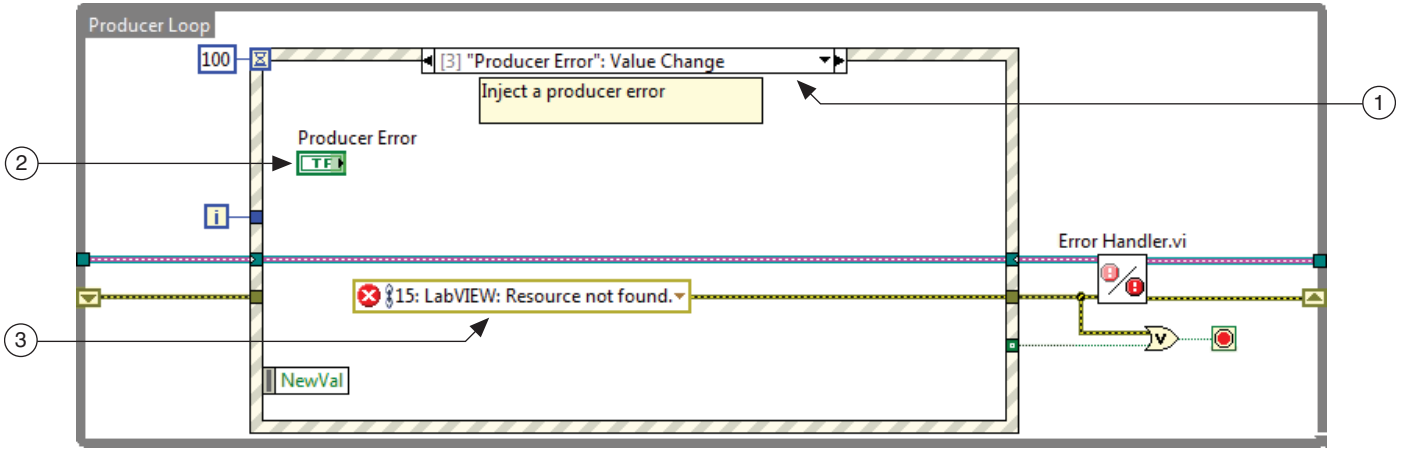
   This VI is similar to the one you used in Exercise 3-1. The Producer Error and Consumer Error buttons are provided on the front panel, as shown in Figure 3-12. You modify the block diagram to enable the buttons and test error handling in this VI.

Figure 3-12. Producer Consumer Main VI Front Panel with Error Buttons



2. Create a new event to inject an error into the Producer Loop as shown in Figure 3-13.

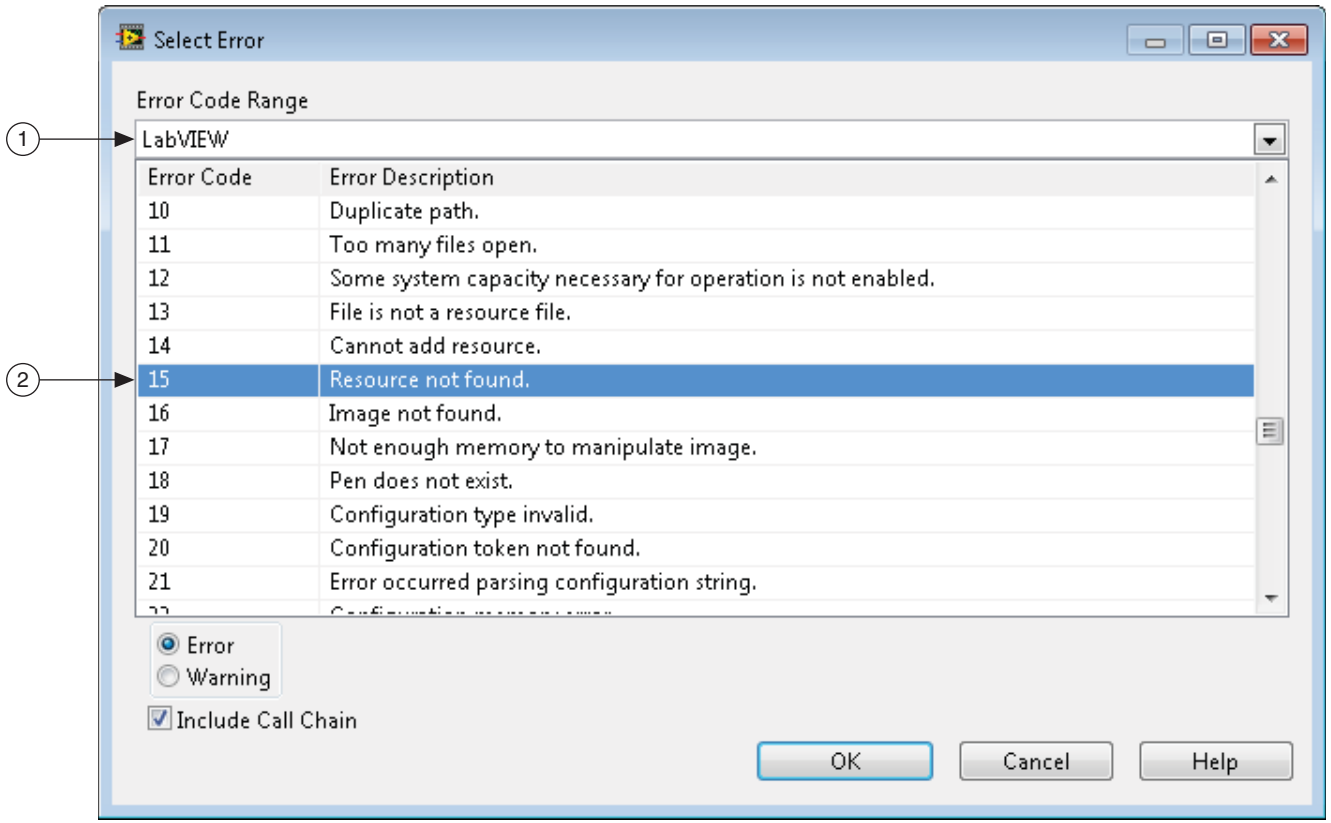
Figure 3-13. Producer Loop "Producer Error": Value Change Event



1 "Producer Error": Value Change Event—Right-click the Event Structure and select **Add Event Case**.
2 Producer Error—Drag the terminal into the new Event Case.
3 Error Ring Constant—When an error occurs, the VI stops running and the error message you select here is displayed in a dialog box. Refer to Figure 3-14 to configure the Error Ring Constant.

3. Click the down arrow in the Error Ring constant and configure the Error Ring to display the message **`15: LabVIEW: Resource not found`** as shown in Figure 3-14.

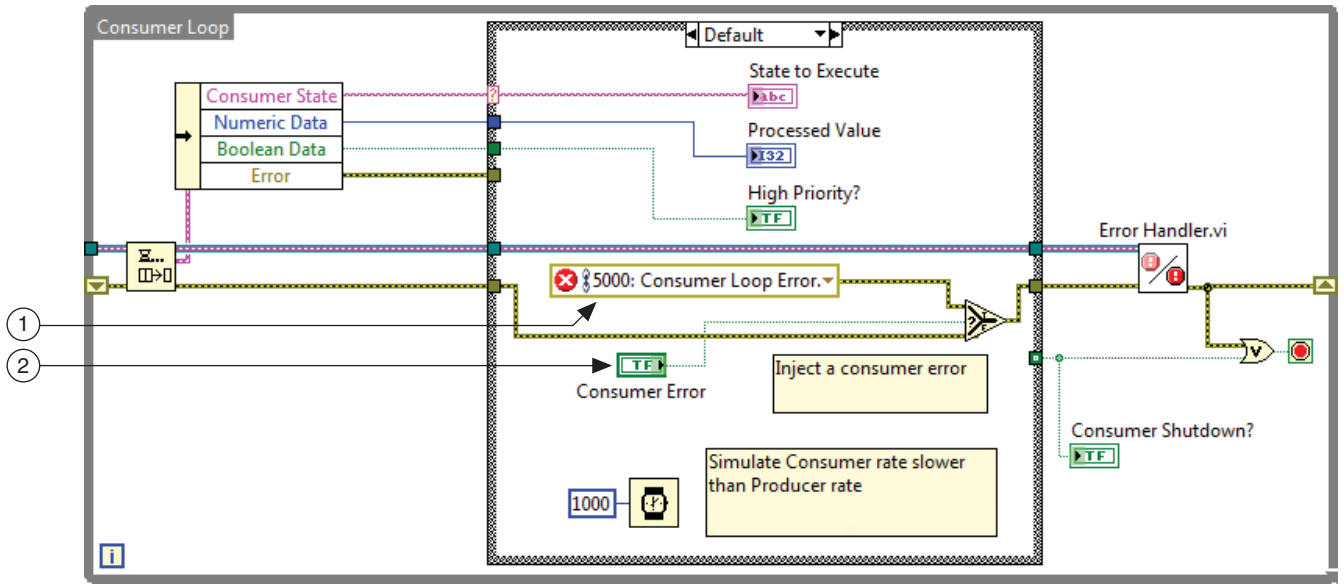
Figure 3-14. Select Error Dialog Box



1 Error Code Range—Select LabVIEW from the drop-down list.
2 Error Code—Select any error in the list and type `15` to find this error quickly.

4. Create the code for the Consumer Error button in the Consumer Loop as shown in Figure 3-15.

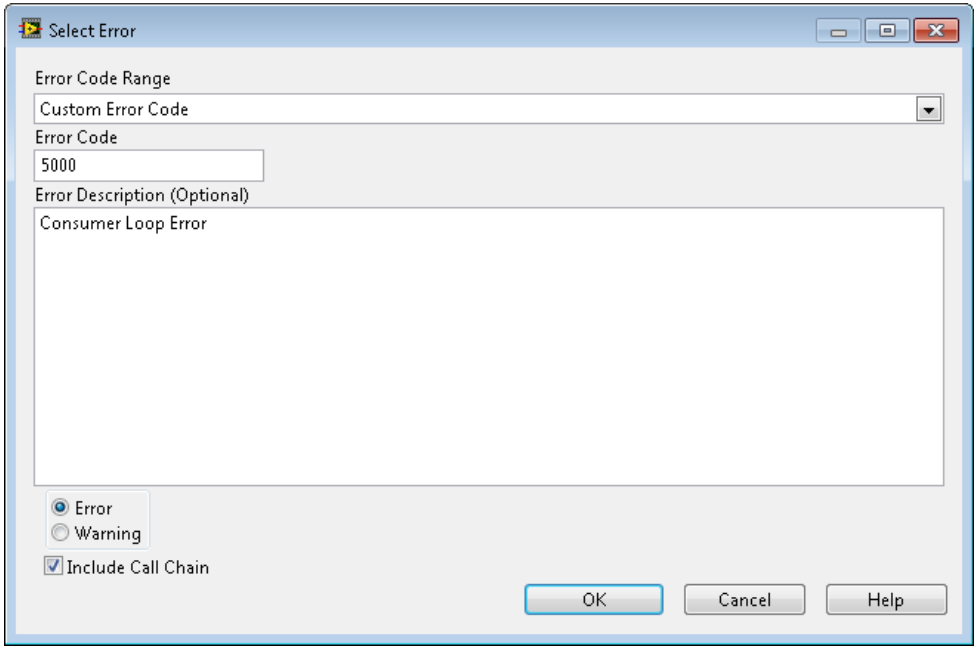
Figure 3-15. Consumer Loop Custom Error Code



1 Custom Error Code—Refer to Figure 3-16 to create a custom error message.
2 Consumer Error—Drag the terminal into the Default case of the Case Structure in the Consumer Loop.

5. Click the down arrow in the Error Ring constant and configure the Error Ring to display a Custom Error Code as shown in Figure 3-16.

Figure 3-16. Custom Error Code



6. Save the VI.

Test

1. Run **Main.vi**.
2. Send several normal and high priority messages to create a backlog.
3. Click the **Producer Error** button.
4. Run the VI again and send multiple messages.
5. Click the **Consumer Error** button.

End of Exercise 3-3

Exercise 3-4 Create a Histogram Application

Goal

Modify the producer/consumer template to create a histogram from acquired data.

Scenario

You want to create an application which does the following:

- Simulates acquisition of a waveform.
- Simulates processing of the waveform which includes generating a histogram.
- Saves a snapshot of a histogram.

You can modify the producer/consumer template to handle those three tasks as well as errors and UI events from the producer/consumer template itself.

Design

After copying the template, you update the producer loop to generate waveform data and you update the consumer loop to display a histogram and take a snapshot of the histogram when the user specifies.

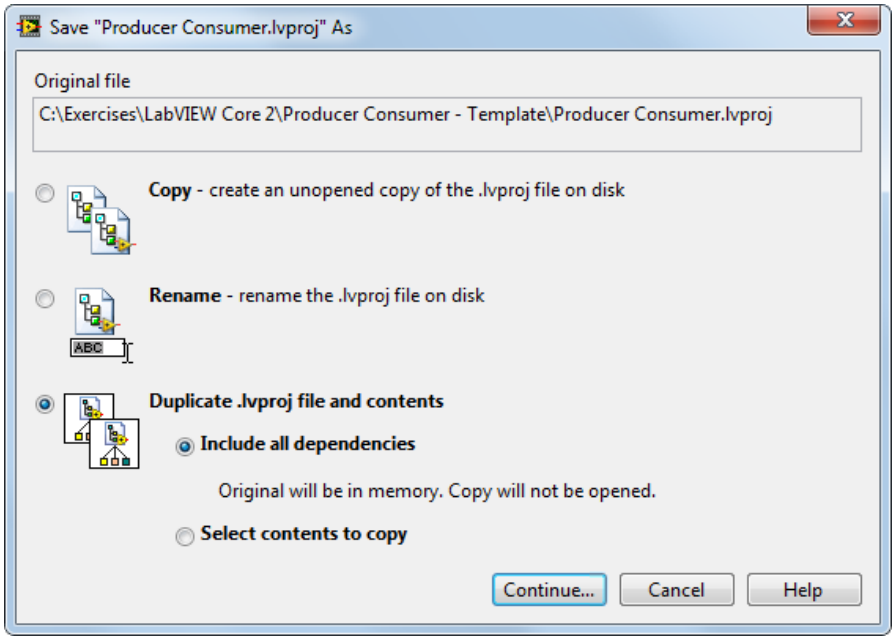
Implementation

1. Move and rename the Producer Consumer project and files.

   ☐ Open the `Producer Consumer.lvproj` located in the `<Exercises>\LabVIEW Core 2\Producer Consumer - Template` directory.

   ☐ Select **File»Save As** and set the save as options as shown in Figure 3-17, and then click the **Continue** button.

Figure 3-17. Save As Options



   ☐ Enter `Histogram` as the name of the project, and save the project to the `<Exercises>\LabVIEW Core 2\Histogram` directory.

2. Close the **Producer Consumer** - **Template** Project Explorer window.

3. Open Histogram.lvproj and rename the project VIs in LabVIEW so that LabVIEW can update all links and instances of the VIs.

   ☐ Right-click **Main.vi** in the **Project Explorer** window and select **Rename**.

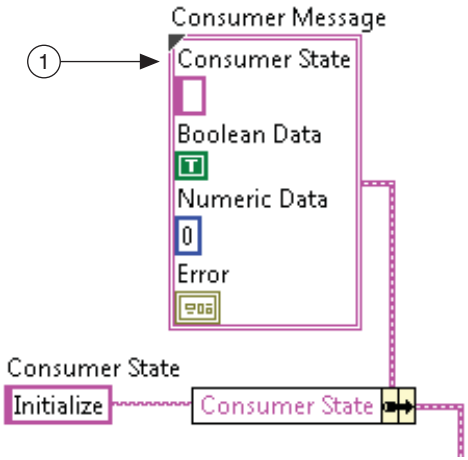
   ☐ Rename the VI as `Histogram Main.vi` and click **OK**.

4. Add the `Shared` folder to the project as an auto-populating folder. The Shared folder contains the Generate Data VI and the Running Histogram VI that you use later.

5. Open the block diagram of the Histogram Main VI.

6. Update the **Consumer Message** type definition, shown in Figure 3-18 to handle waveform data.

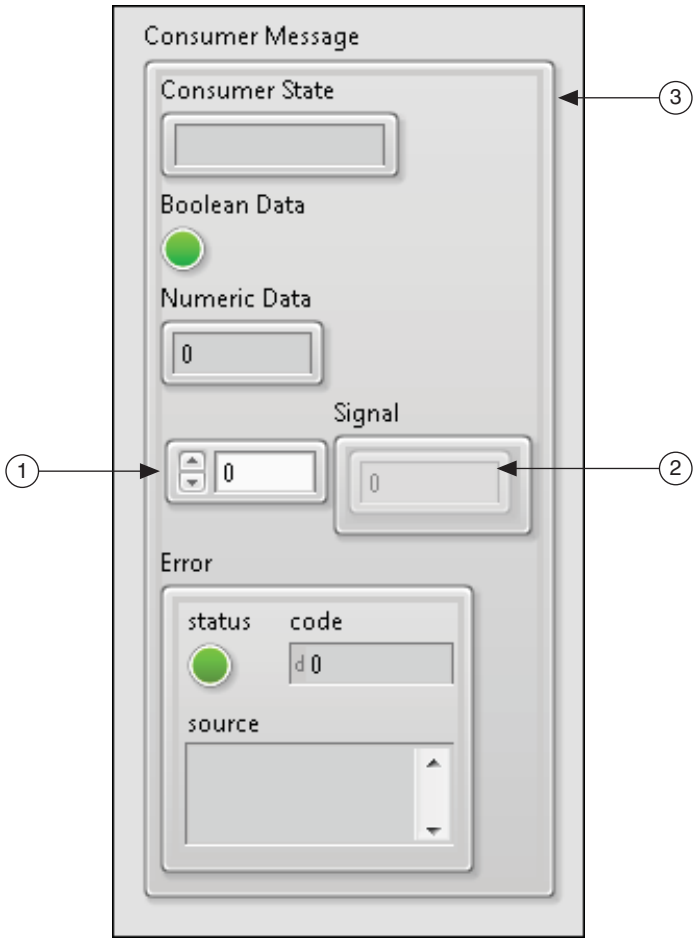
Figure 3-18. Consumer Message Type Definition Terminal on Histogram Main VI Block Diagram



1 Consumer Message type definition—Right-click the Consumer Message type definition located to the left of the producer loop on the Histogram Main VI block diagram and select **Open Type Def**.

a. Modify the Consumer Message type definition as shown in Figure 3-19.

Figure 3-19. Consumer Message Type Definition – Consumer Message.ctl

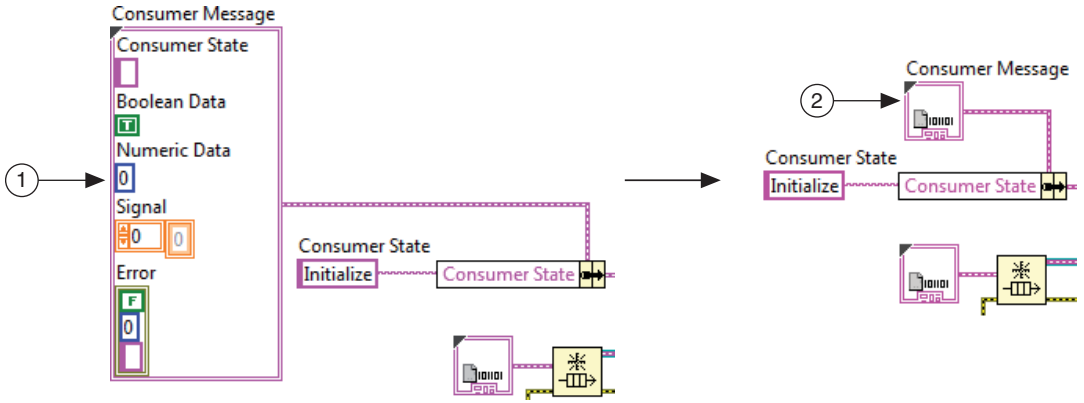


1 Array—Add an array to the type definition so it can handle waveform data. Rename the Array `Signal`.
2 Numeric Indicator—Add a numeric indicator to the array.
3 Right-click the cluster border and select **Reorder Controls In Cluster** and arrange them so that the Signal control is directly below the **Numeric Data** control.

b. Apply changes, save, and close the type definition.

7. Display the type definition as an icon on the block diagram as shown in Figure 3-20.

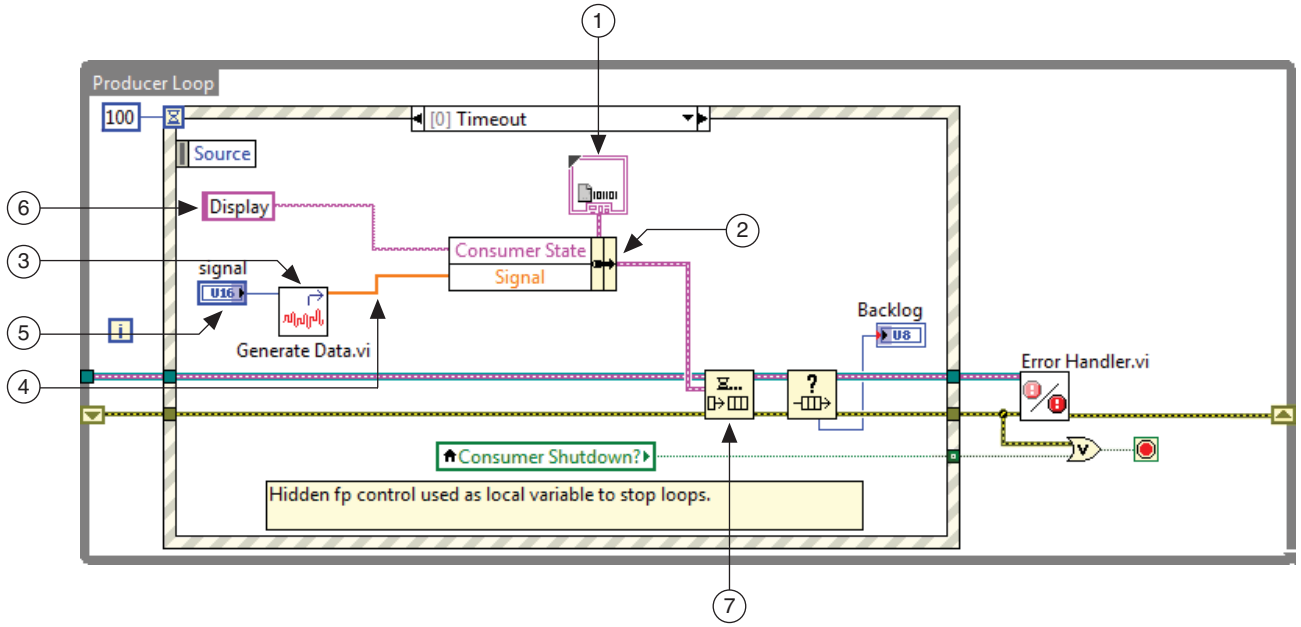
Figure 3-20. Viewing a Type Definition as an Icon



1 Right-click the Consumer Message type definition and select **AutoSizing»Arrange Vertically** from the shortcut menu.
2 Right-click the Consumer Message type definition and select **View Cluster as Icon** to save space on the block diagram.

8. Send signal data through the Consumer Message type definition. Complete the Timeout event in the producer loop as shown in Figure 3-21.

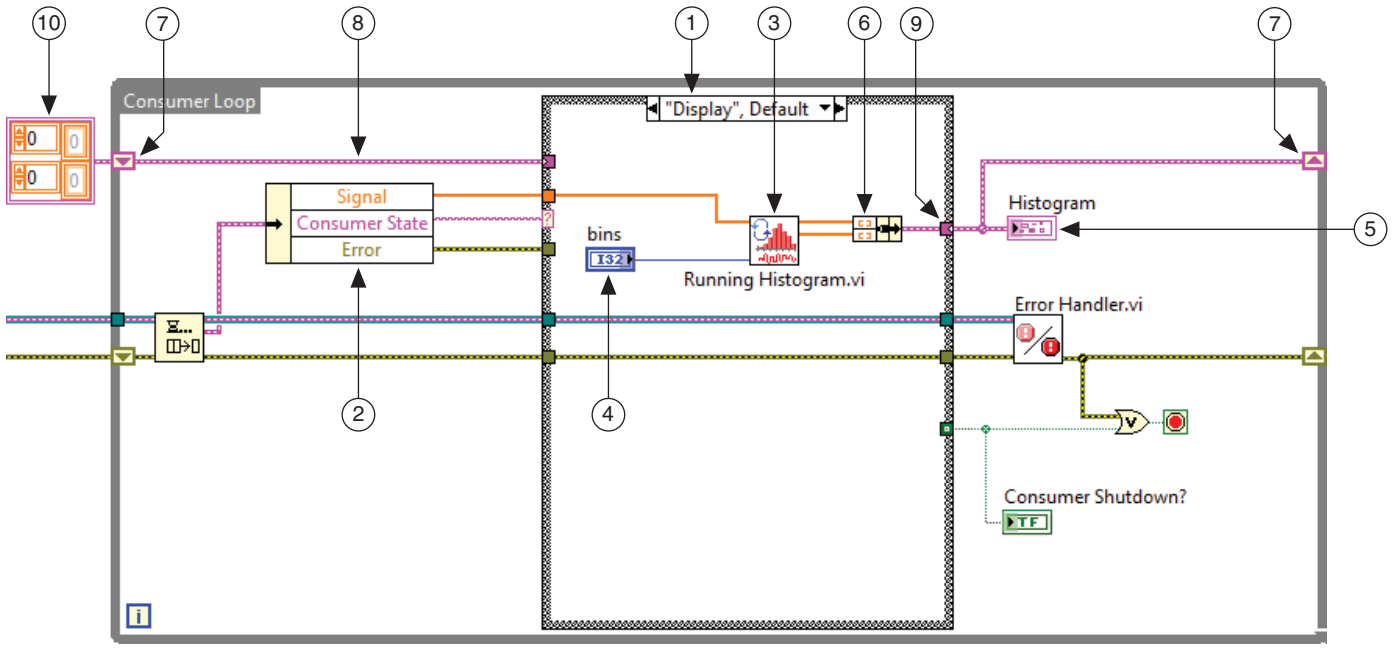
Figure 3-21. Updating the Producer Loop Timeout Event



1 Consumer Message type definition—Copy the Consumer Message type definition and paste it inside the Timeout event case.
2 Bundle By Name function—Wire the Consumer Message typedef to the **input cluster** input.
   ☐ Expand the node to display two elements.
   ☐ Select **Consumer State** and **Signal**.
3 Generate Data VI—Drag the Generate Data VI from the **Shared** folder in the **Project Explorer** window into the Timeout event case.
4 Wire the **Y** output of the Generate Data VI to the **Signal** input of the Bundle By Name function.
5 Create a control for the **signal** input of the Generate Data VI.
6 Create a constant for the **Consumer State** input.
7 Enqueue Element—Right-click the queue wire and select **Insert»Queue Operations Palette»Enqueue Element.**
   ☐ Wire the error wire through the Enqueue Element function to the Get Queue Status function. It will appear wired, but when you insert the node, the error wire is behind the Enqueue Element function.

9. Create the Display case in the consumer loop as shown in Figure 3-22.

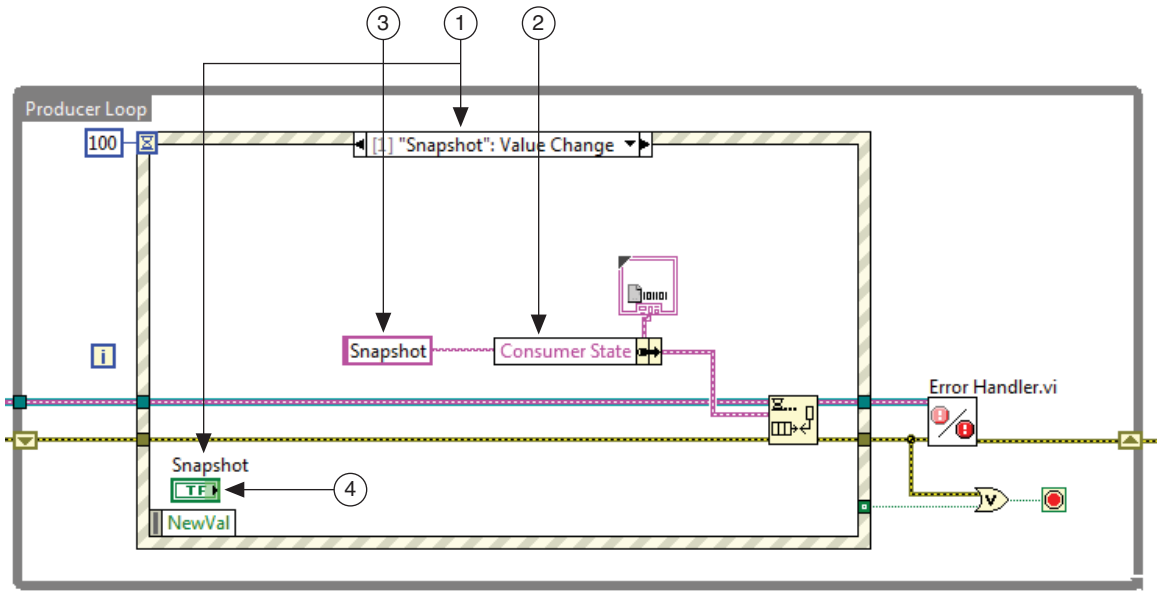
Figure 3-22. Updating the Consumer Loop Display Case



1 Open to the Default case of the Case structure and rename the case to `"Display", Default`.
2 Unbundle By Name function—Change the **Numeric Data** element to **Signal** and remove the **Boolean Data** wire and element.
3 Running Histogram VI—Drag the Running Histogram VI from the **Shared** folder in the **Project Explorer** window.
4 Numeric control—Create a control for the **bins** input and label the control `Bins`.
5 XY Graph (Silver)—On the front panel, place an XY Graph (Silver) and rename it `Histogram`.
6 Bundle function—Wire the **histogram** and **x axis** outputs from the Running Histogram VI to the Bundle function.
7 Replace the right Histogram tunnel with a shift register and complete the shift register.
8 Wire the left shift register to the Case structure.
9 Right-click the Histogram output tunnel and select **Linked Input tunnel»Create & Wire Unwired Cases** and then click the Histogram input tunnel on the left.
10 Right-click the left shift register and create a constant.

10. Create a Snapshot event in the producer loop by changing the “High Priority Message”: Value Change event, as shown in Figure 3-23.

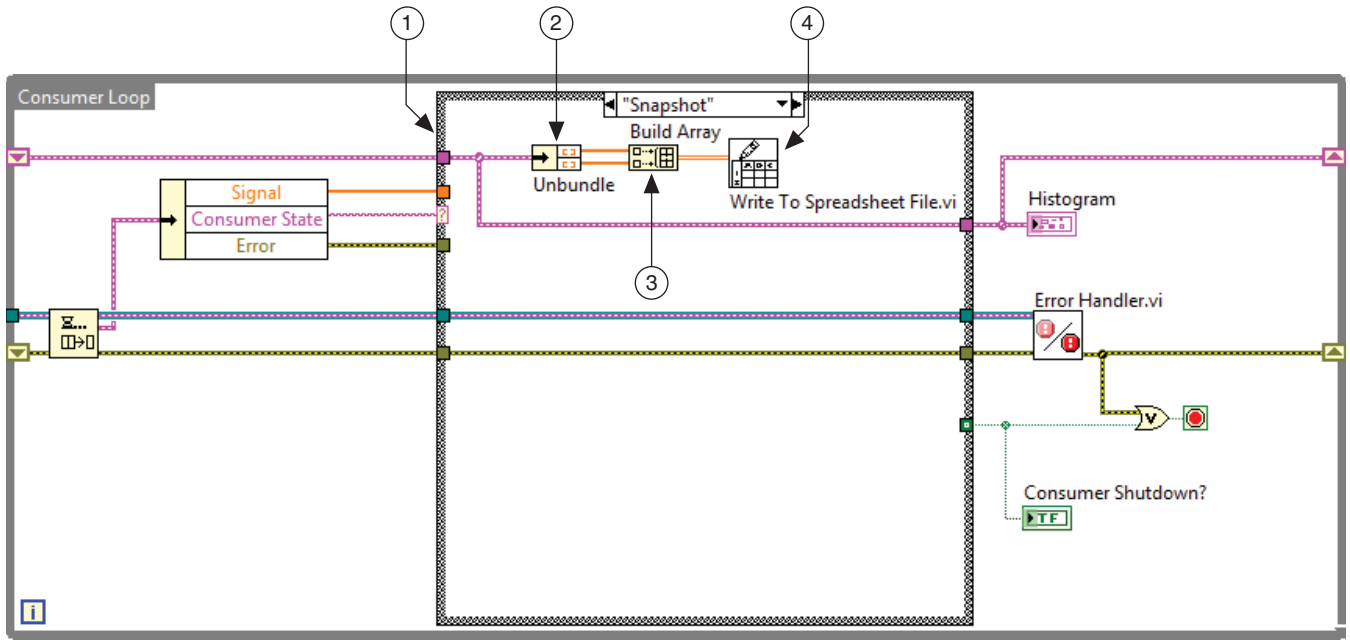
Figure 3-23. Updating the Producer Loop “Snapshot”: Value Change Event



1 Change the event name—Change the label of the **High Priority** button to `Snapshot`. Changing the name of the button changes the event name.
2 Bundle By Name function—Delete the values wired to the **Boolean Data** and **Numeric Data** inputs of the Bundle By Name function and hide the terminals.
3 Change the value of the **Consumer State** string constant to `Snapshot`.
4 Double-click the **Snapshot** control to locate the button on the front panel. Change the Boolean text displayed on the button to `Snapshot`.

11. Add the Snapshot case to the consumer loop as shown in Figure 3-24.

Figure 3-24. Updating the Consumer Loop Snapshot Event

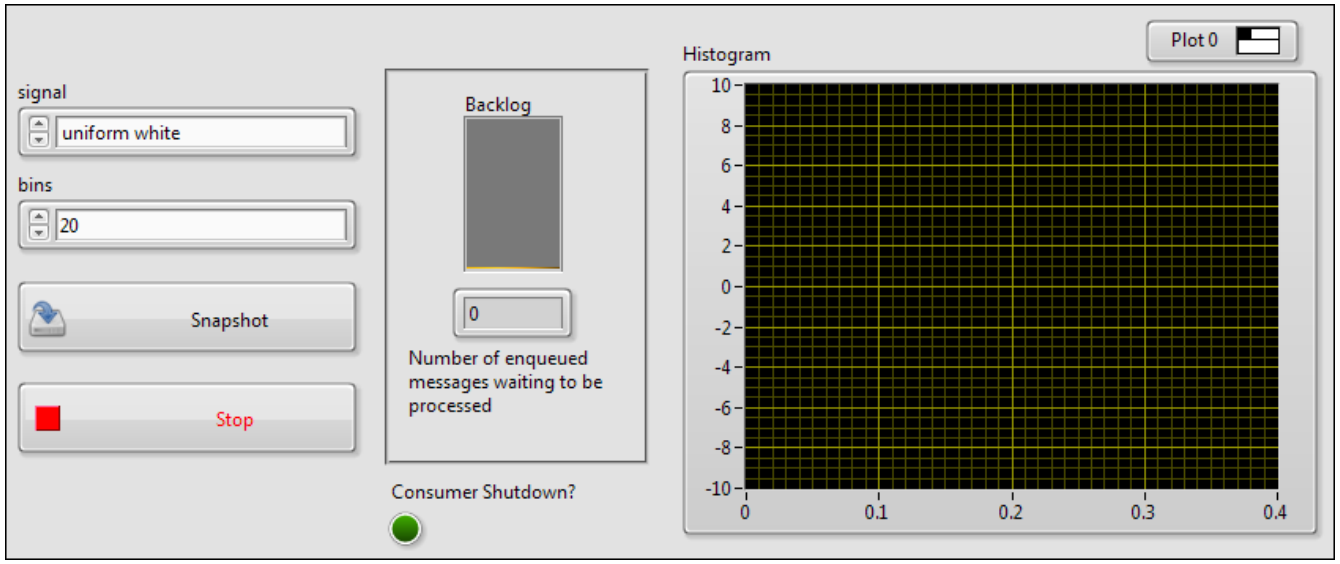


1 Duplicate the Initialize case—Right-click the case structure and select **Duplicate Case**. Rename the duplicate case `Snapshot`.
2 Unbundle function—After you wire the input, the Unbundle function contains two 1D arrays.
3 Build Array function—Wire both **1D Array** outputs to the Build Array function.
4 Write to Spreadsheet File VI—Wire the **appended array** output of the Build Array function to the **2D data** input.

12. Delete the Normal Priority Message event from the Event structure in the Producer Loop. LabVIEW deletes the corresponding button from the front panel.

13. Cleanup the front panel of the VI as shown in Figure 3-25.

Figure 3-25. Cleaning Up the Front Panel of the Histogram Main VI



Test

1. Run the VI.

2. To create the look of a histogram in the chart, click the plot legend and select a horizontal bar plot type from the bottom row. You may also want to remove the line interpolation by clicking the plot legend and selecting **Interpolation** from the shortcut menu.

3. Notice how changing the **Signal** and **Bins** values changes the look of the histogram.

4. Click the **Snapshot** button. A file dialog box displays so you can save the histogram file.

   ☐ Choose a name for the new file, including `.txt`.

   ☐ While the dialog box is open, the **Backlog** indicator rises.

   ☐ Click the **OK** button to save the file.

   ☐ The **Backlog** indicator should quickly decrease.

5. Click the **Stop** button to stop the VI.
6. Open the saved text file and review the contents to see the bins and values of the histogram.
7. Save and close the Histogram project.

End of Exercise 3-4

Exercise 4-1 Display Temperature and Limits

Goal

Use Property Nodes to change the properties of front panel objects programmatically.

Scenario

Complete a VI that records temperature to a waveform chart. During execution, the VI performs the following tasks:

- Disable and enable the controls at the start and completion of execution.
- Set the Δx value of the chart to the user-defined value.
- Clear the waveform chart so it initially contains no data.
- Challenge: Change the color of a plot if the data exceeds a certain value.

Design

You build this VI in four stages, including a challenge.

Part 1—Disable Controls

Part 2—Enable Controls

Part 3—Clear Chart

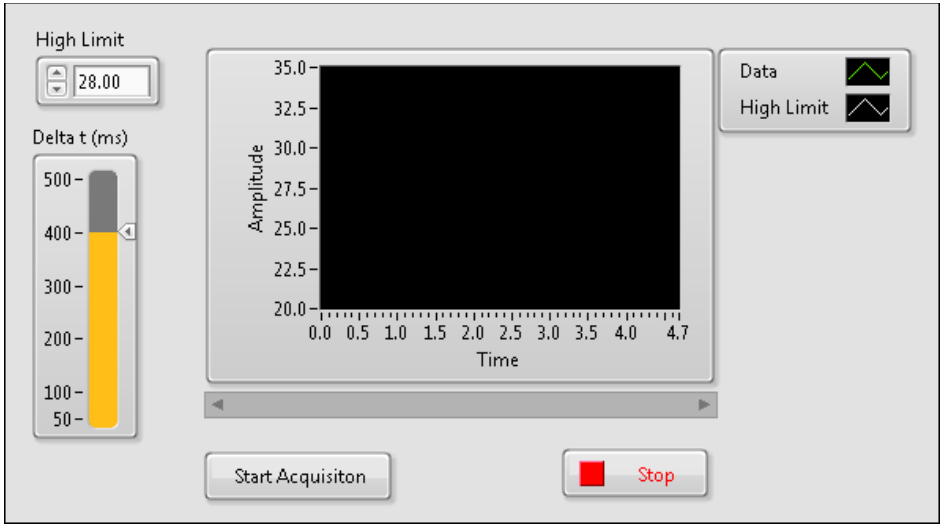
Part 4—Challenge: Change the Plot Color

Implementation

Part 1—Disable Controls

1. Open `Temperature Limit.vi` from the Temperature Limit project located in the `<Exercises>\LabVIEW Core 2\Temp Limit - Ctl Props` directory.

Figure 4-1. Temperature Limit Front Panel



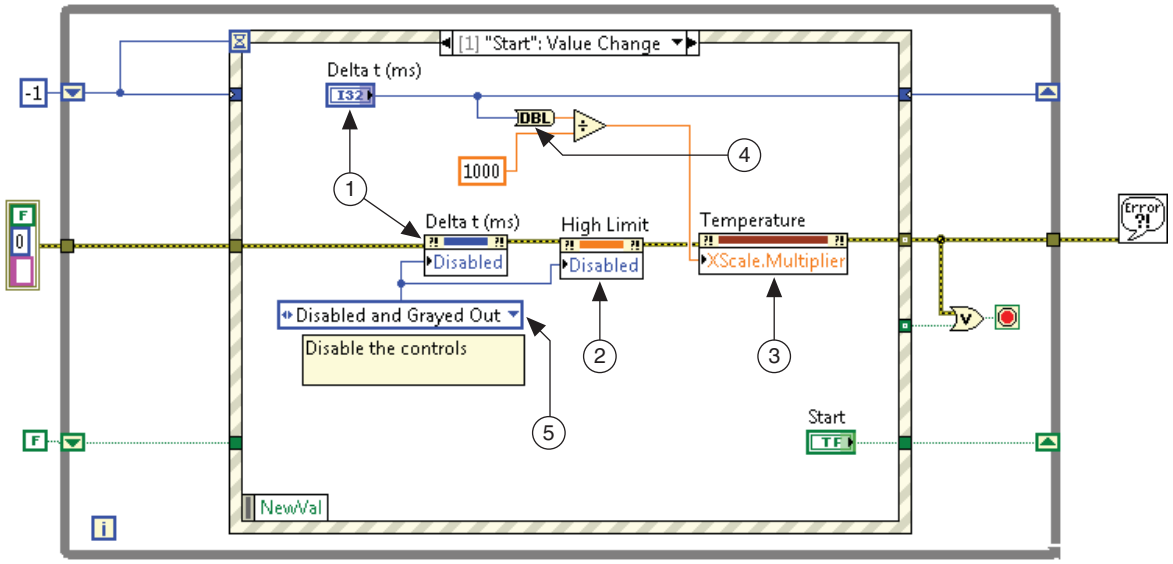
2. Run the VI and then click the **Start Acquisition** button.

   ☐ Notice that while the VI runs, the controls are still enabled.You can change the values on the controls while the VI runs.

   ☐ Click the **Stop** button.

3. Modify the block diagram as shown in Figure 4-2 to disable the controls when the VI is running.

Figure 4-2. Temperature Limit—Disable Controls Block Diagram



1 Delta t (ms) Property Node—Right-click the Delta t (ms) control and select **Create»Property Node»Disabled**. Right click the property node and select **Change All to Write**.

2 High Limit Property Node—In the Timeout Event Case, right-click the High Limit control and select **Create»Property Node»Disabled**.

- ☐ Place the Property Node outside the While Loop, so you can move it into the "Start": Value Change event case.
- ☐ Right-click the property node and select **Change All to Write.**
- ☐ Move the High Limit property node into the "Start": Value Change event case.

3 Temperature Property Node—In the Timeout Event Case, right-click the Temperature indicator and select **Create»Property Node»X Scale»Offset and Multiplier»Multiplier**.

- ☐ Place the Property Node outside the While Loop, so you can move it into the "Start": Value Change event case.
- ☐ Right-click the property node and select **Change All to Write.**
- ☐ Move the Temperature property node into the "Start": Value Change event case.

4 To Double Precision Float—Converts the I32 input from the Delta t (ms) control to a double precision number.

5 Right-click the Delta t (ms) property node and select **Create»Constant** and set it to **Disabled and Grayed Out**.

Test

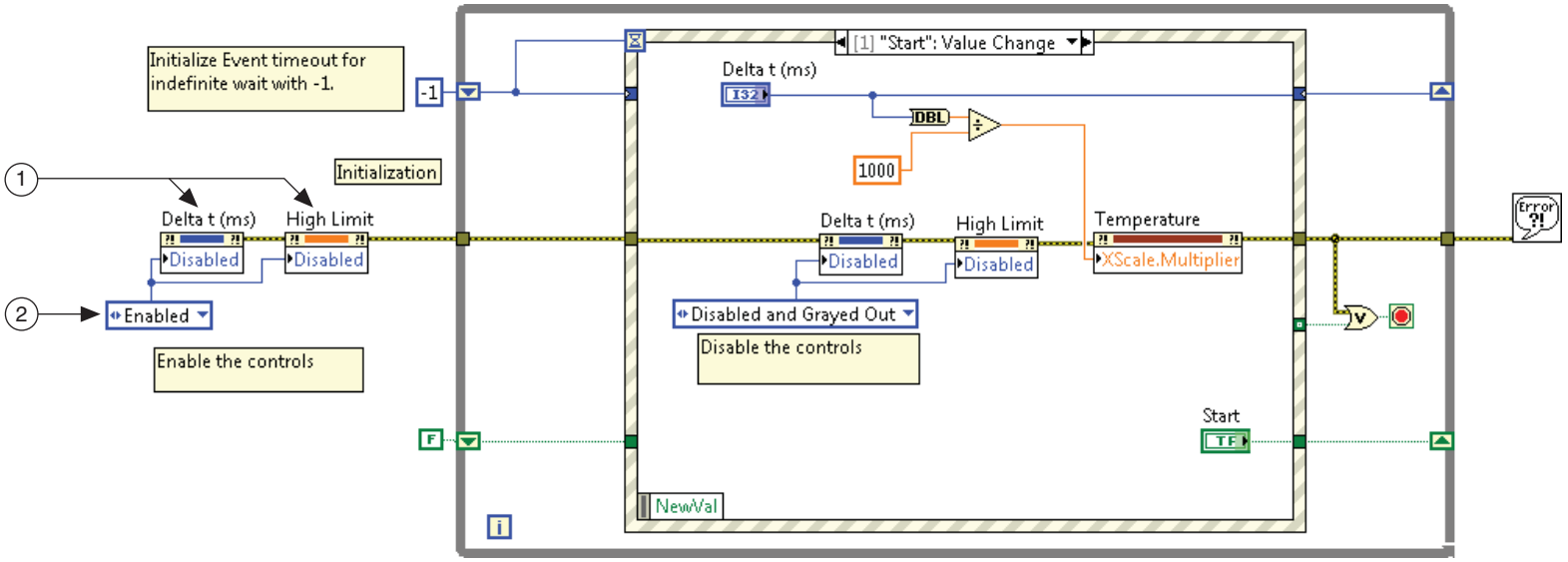
1. Run the VI and click the **Start Acquisition** button. The Delta t (ms) and High Limit controls are disabled and grayed out.
2. Stop the VI.
3. Run the VI a second time, click the **Start Acquisition** button and notice that the controls are still disabled.

Part 2—Enable Controls

You want to disable the controls while the VI is running, however, you want to enable them again the next time you run the VI.

1. Modify the block diagram as shown in Figure 4-3 to enable the controls each time you run the VI.

Figure 4-3. Temperature Limit—Enable Controls Block Diagram



1 Create copies of the Delta t (ms) and High Limit property nodes and drag them to the left of the While Loop.
2 Create a constant to enable the controls.

Test

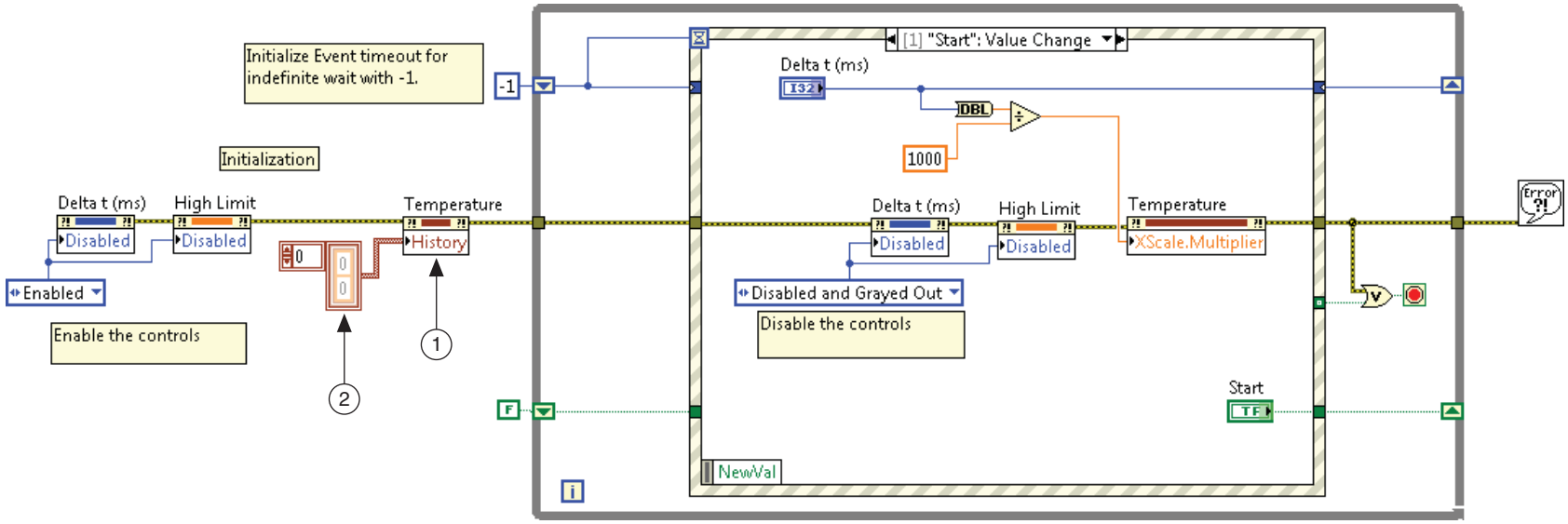
1. Run the VI and notice that the controls are enabled again before you click the **Start Acquisition** button.
2. Set different values for the controls and click the **Start Acquisition** button. Notice that the data displayed on the chart starts from where it stopped the last time you ran the VI.

Part 3—Clear Chart

You want to clear the chart each time you run the VI.

1. Modify the block diagram as shown in Figure 4-4 to clear old data from the chart each time you run the VI.

Figure 4-4. Temperature Limit—Clear Chart Block Diagram



1 Temperature Property Node—Create a copy of the Temperature Property node and drag it to the left of the While Loop.
   ☐ Click the new property node and select **History Data**.
2 Right-click the History input and select **Create»Constant**. This creates an empty array of clusters to initialize the temperature chart to 0 when the VI starts running.

Test

1. Run the VI, click the **Start Acquisition** button and let the VI run for a few seconds, then click the **Stop** button.

2. Run the VI a second time. Notice that the chart clears before you click the **Start Acquisition** button and new data is written to it.

Part 4—Challenge: Change the Plot Color

Modify the VI so that the Data plot changes color when it exceeds the high limit.

End of Exercise 4-1

Exercise 4-2 Customize the VI Window

Goal

Affect the attributes of a VI by using Property Nodes and Invoke Nodes.

Scenario

You can set the appearance properties of a VI statically by using the VI properties page. However, robust user interfaces often must modify the appearance of a front panel while the program runs.

Modify the Temperature Limit VI to have the following appearance and behaviors when the VI is running:

- Hide the tool bar
- Hide the menu bar
- Hide the scroll bars
- Move to the center of the screen
- Write data to an Excel file

Design—Properties

Use the following properties and methods on the VI class:

- **ShowMenuBar**—When this property is false, the menu bar of the VI is not displayed.

Figure 4-5. VI Menu Bar

- **Tool Bar:Visible**—When this property is false, the tool bar of the VI is not displayed.

Figure 4-6. VI Tool Bar

Design—Methods

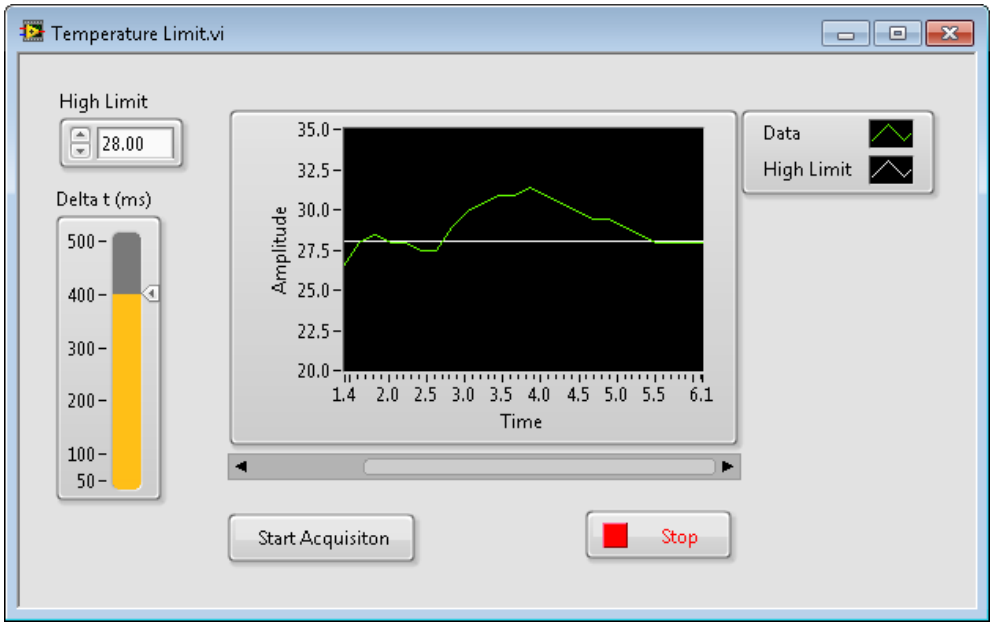
Unlike properties, a method has an effect every time you call it. Therefore, you should call methods only when you want to perform an action. For example, if you call the Fp.Center method during each iteration of a loop, the VI is continually centered, thereby preventing the user from moving it. You can use a Case structure to control calling the method in a given iteration of a loop. Use the following method on the VI class:

- **Center**—Each time this method is called, the VI moves to the center of the screen.

Tip Use the Context Help window to view descriptions of each property and method.

After you implement the changes to the VI, when you run the Temperature Limit VI it should move to the center of the screen and look similar to Figure 4-7.

Figure 4-7. Temperature Limit VI Front Panel with Customized Appearance



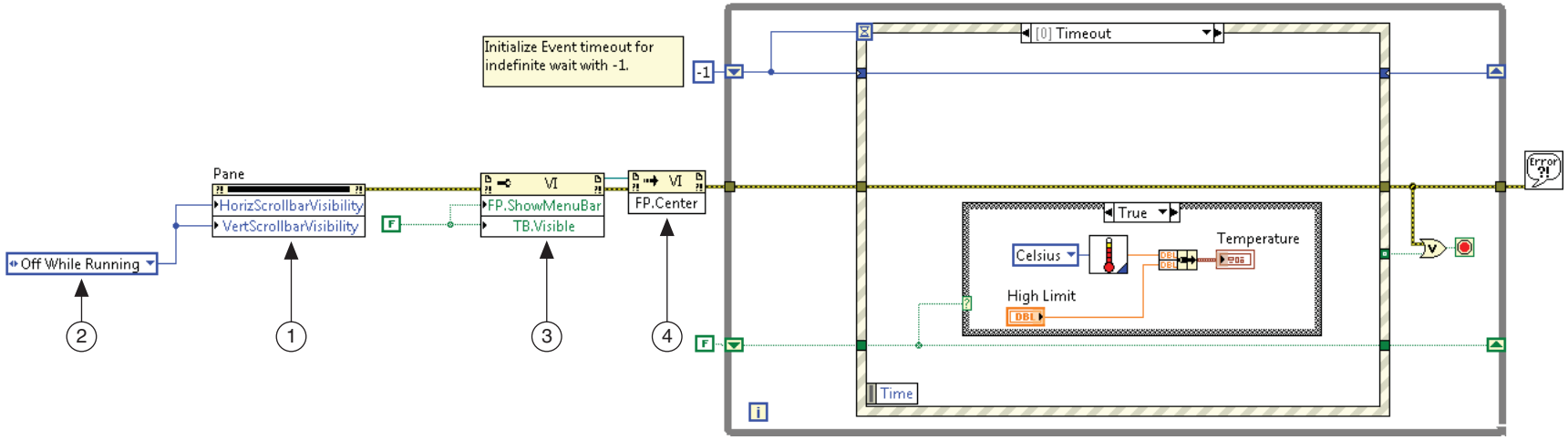
Implementation

Part 1—Set Appearance Properties

1. Open the Temperature Limit VI from the Temperature Limit project located in the `<Exercises>\LabVIEW Core 2\Temp Limit - Methods` directory.

2. Modify the block diagram as shown in Figure 4-8 to hide the scrollbars, menu bar, and tool bar, and center the front panel on the screen while the VI is running.

Figure 4-8. Temperature Limit VI—Methods Block Diagram



1 Property Node—Right-click the Property Node and select **Link to»Pane»Pane**.
   - ☐ Right-click and select **Change All to Write**.
   - ☐ Expand the node to display two properties and set them to **Horizontal Scroll Bar Visibility** and **Vertical Scrollbar Visibility**.

2 Off While Running constant—Right-click one of the inputs to the Pane property node and select **Create»Constant**.

3 Property Node—Right-click the Property Node and choose **Select Class»VI Server»VI»VI**.
   - ☐ Right-click and select **Change All to Write**.
   - ☐ Expand the node to display two properties.
   - ☐ Click the top property and select **Select Property»Front Panel Window»Show Menu Bar**.
   - ☐ Click the lower property and select **Select Property»Tool Bar»Visible**.
   - ☐ When you wire a False constant to each of the properties, the menu bar and tool bar will be hidden when the VI runs.

4 Invoke Node—You must wire the reference from the VI Property Node before setting this method. Click Method and select **Select Method»Front Panel»Center**.

Note Notice that the scrollbar visibility properties apply to the Pane class, not the VI class. The front panel can be split into multiple panes using the horizontal splitter bar or vertical splitter bar. Each pane can have its own scrollbars.

3. Save the VI.

Test

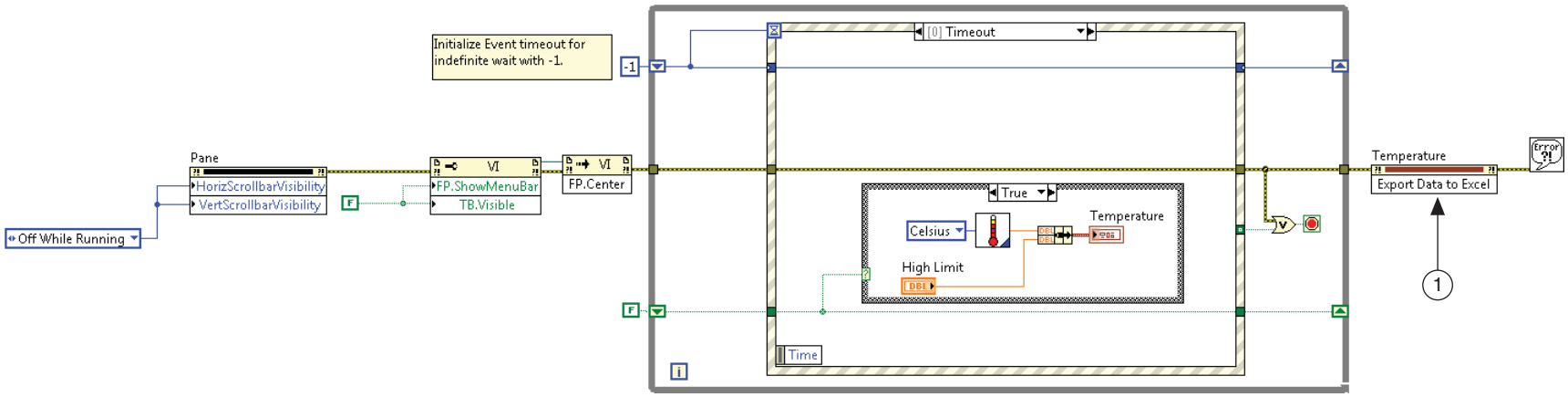
1. Run the VI.
2. Verify that the scroll bars, tool bar, and menu bar are not displayed, and that the front panel window is centered on the screen while the VI runs.
3. Stop the VI.

Part 2—Export Data

Add a method to export Temperature chart data to Excel.

1. Modify the block diagram as shown in Figure 4-9 to export the data displayed on the Temperature chart to Excel.

Figure 4-9. Temperature Limit VI—Export to Excel Block Diagram



1 Temperature Invoke Node—Right-click the Temperature indicator and select **Create»Invoke Node»Export Data to Excel**.

2. Save the VI.

Test

1. Run the VI.
2. Click **Stop**. The Export Data to Excel method creates a temporary Excel file with the data from the Waveform chart. View the data in the Excel file.
3. Save and close the VI.

End of Exercise 4-2

Exercise 4-3 Create SubVIs for Common Operations

Goal

Use control references to create subVIs that modify VI properties.

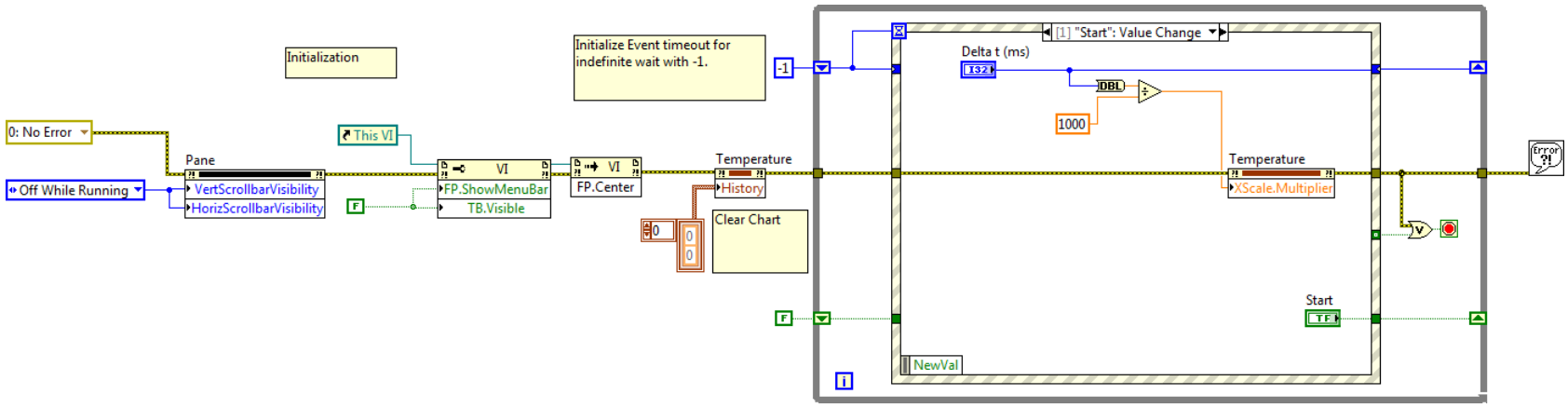
Scenario

Create subVIs for the Temperature Limit VI that allow you to handle some of the functionality that you enabled in previous exercises.

Implementation

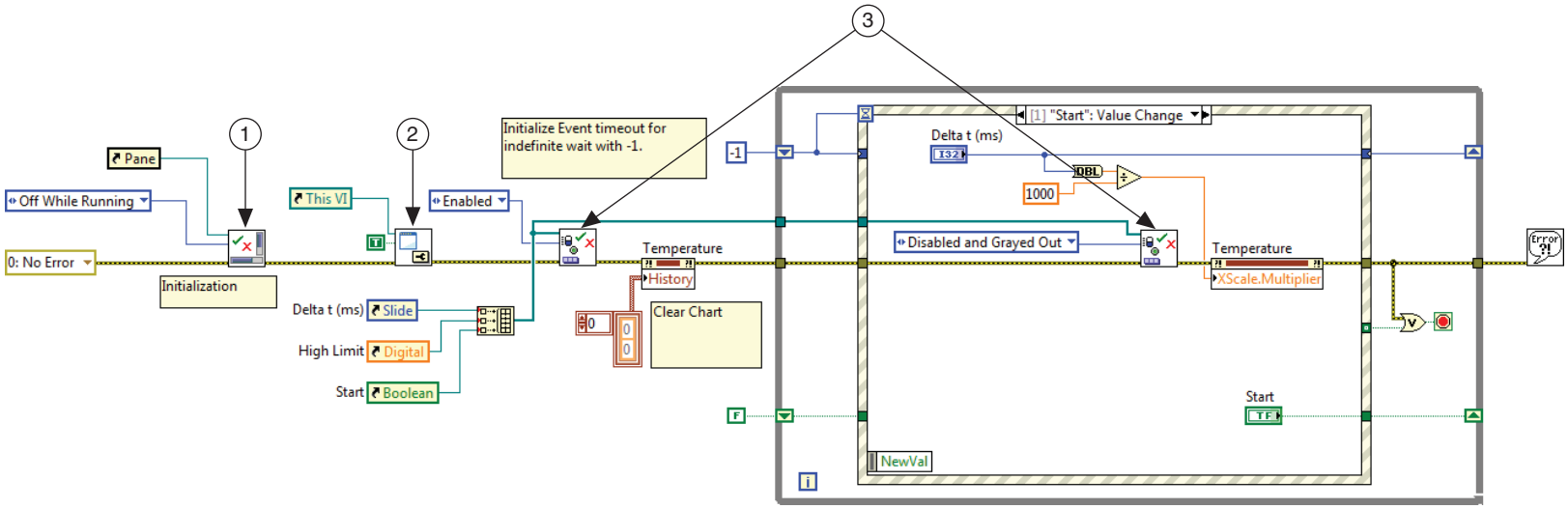
1. Open the Temperature Limit VI from the Temperature Limit Project located in the `<Exercises>\LabVIEW Core 2\Temp Limit - SubVIs` directory.
2. The block diagram, as shown in Figure 4-10 includes code to:
   - hide scroll bars while running
   - set dialog properties
   - reenable the controls when you stop the VI

Figure 4-10. Temperature Limit VI—Before Using SubVIs



In this exercise, you modify the block diagram to use subVIs and use control references. Figure 4-11 describes the subVIs you create and use to make the Temperature Limit VI more modular and scalable.

Figure 4-11. Temperature Limit VI—After Using SubVIs



1 Set Scroll Bar State SubVI—Hides the scroll bars when the VI runs.
2 Set Dialog Properties SubVI—Hides the tool bar and menu bar when the VI runs.
3 Set Enable State on Multiple Controls VI—Sets all the controls in the input array to the Enable state value.

3. Create the Set Scroll Bar State subVI.

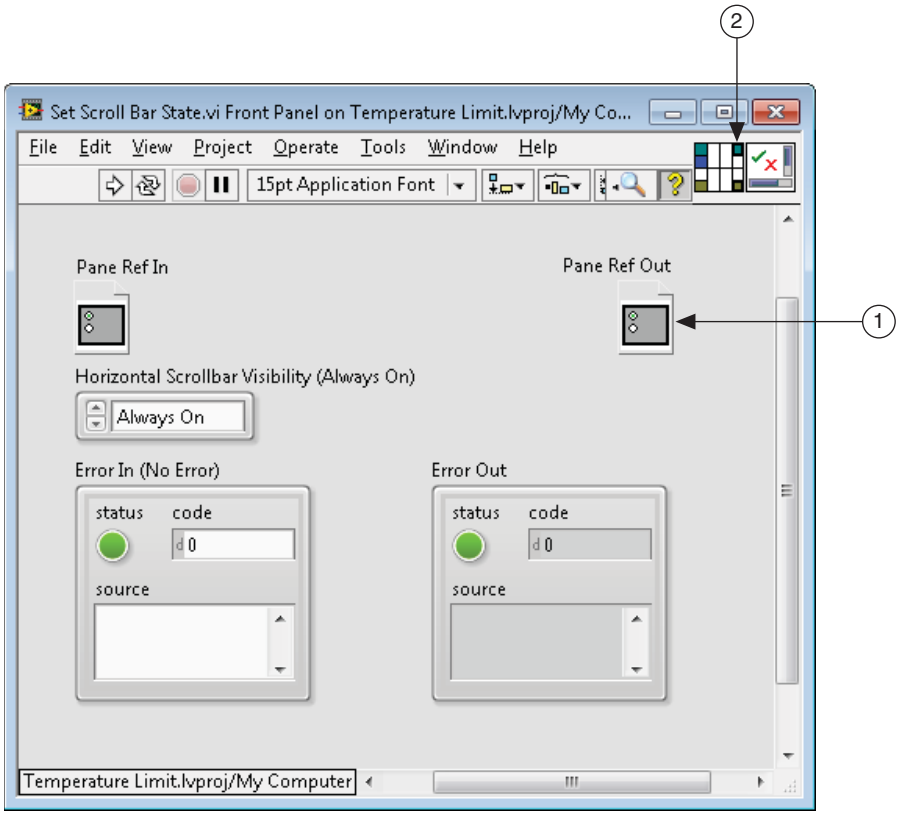
☐ On the Temperature Limit block diagram, highlight the code shown in Figure 4-12 and select **Edit»Create SubVI**.

Figure 4-12. Set Scroll Bar State Code to Highlight

4. Open the subVI.

   ☐ Double-click the subVI icon on the block diagram to open and modify the front panel of the subVI you just created as shown in Figure 4-13.

Figure 4-13. Set Scroll Bar State SubVI Front Panel

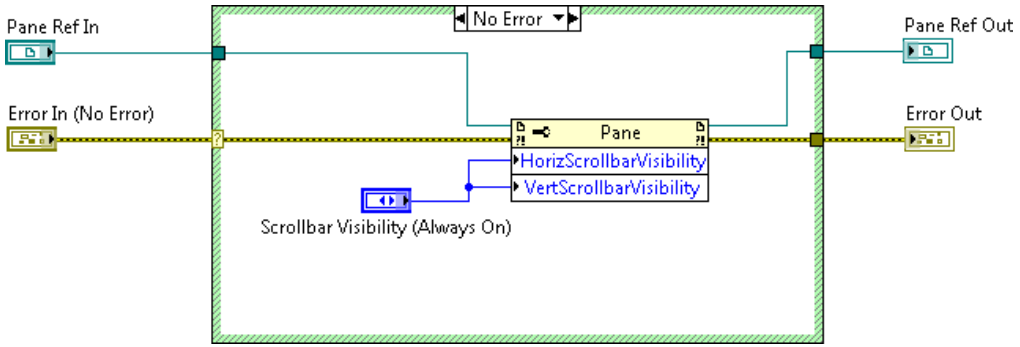


1 Pane Ref Out Indicator—Create a copy of the Pane Ref In control. Right-click the copy and select **Change to Indicator** and change the label.
2 Assign the Pane Ref Out indicator to the top right terminal of the connector pane. Connections for the other controls and indicators should already be created.

5. Create a meaningful icon for the subVI and save it as `Set Scroll Bar State.vi` in the `<Exercises>\LabVIEW Core 2\Temp Limit - SubVI` directory.

6. Modify the block diagram of the Set Scroll Bar State subVI as shown in Figure 4-14.

Figure 4-14. Set Scroll Bar State SubVI Block Diagram



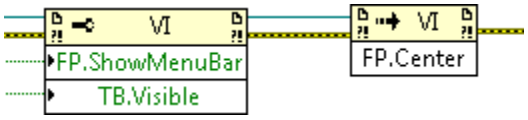
7. Wire the reference and error wires through the Error case.

8. Save and close the subVI.

9. Create the Set Dialog Properties subVI.

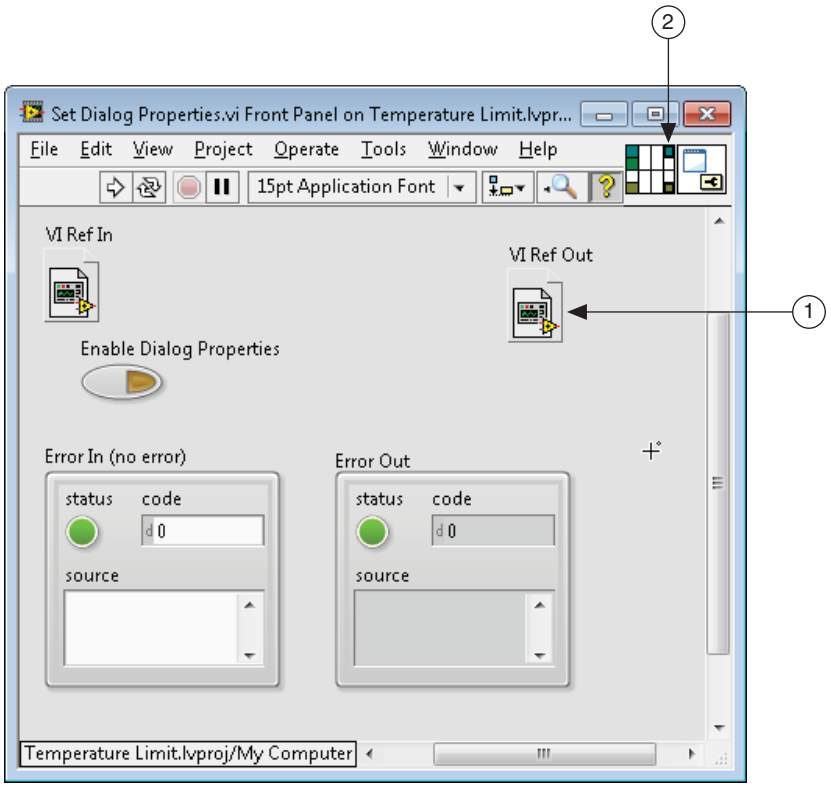
   ☐ On the Temperature Limit VI, highlight the code shown in Figure 4-15 and select **Edit»Create SubVI**.

Figure 4-15. Set Dialog Properties Code to Highlight



10. Open the subVI you just created and modify the front panel as shown in Figure 4-16.

Figure 4-16. Set Dialog Properties SubVI Front Panel

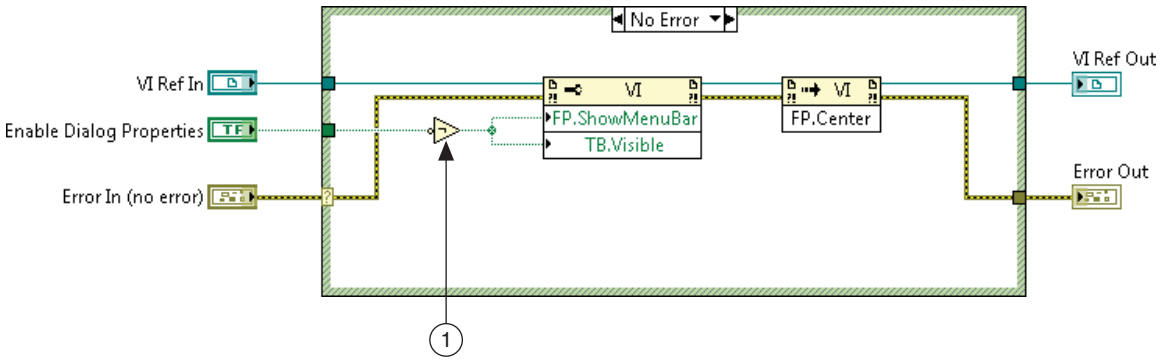


1 VI Ref Out Indicator—Create a copy of the VI Ref In Control. Right-click the copy and select **Change to Indicator** and change the label.
2 Assign the VI Ref Out indicator to the top right terminal of the connector pane. Connections for the other controls and indicators should already be created.

11. Create a meaningful icon for the subVI and save it as `Set Dialog Properties.vi` in the `<Exercises>\LabVIEW Core 2\Temp Limit - SubVI` directory.

12. Modify the block diagram of the Set Dialog Properties subVI as shown in Figure 4-17.

Figure 4-17. Set Dialog Properties SubVI Block Diagram



1	Not function—Invert the logic for the Enable Dialog Properties button when wired to the property node to show the menu bar and tool bar

13. Wire the reference and error wires through the Error case.

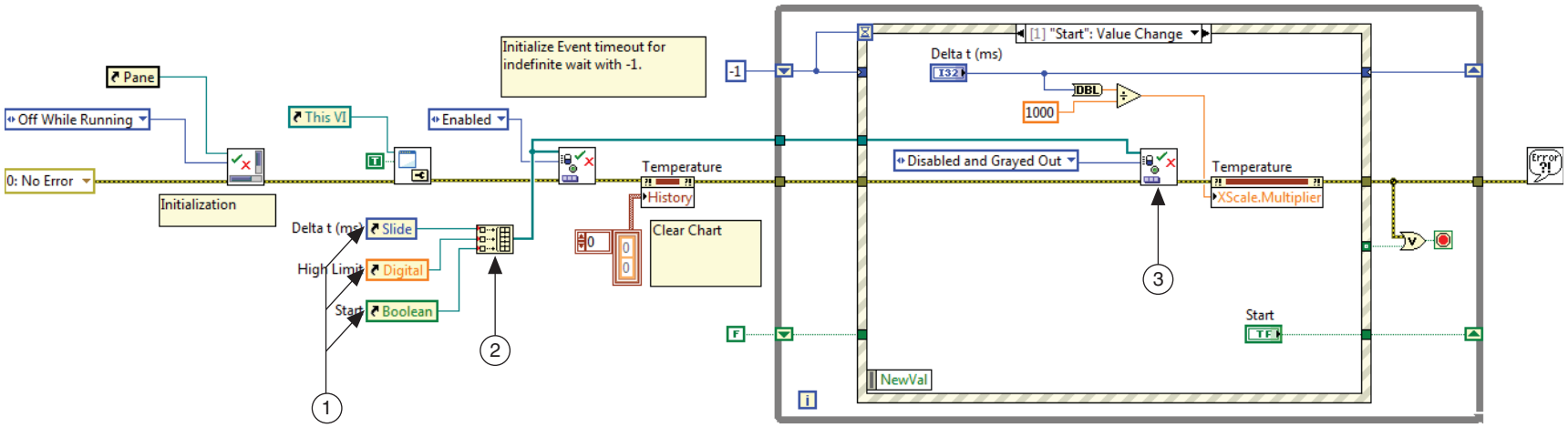
14. Save and close the subVI.

15. Add the Set Enable State on Multiple Controls VI to the block diagram of the Temperature Limit VI.

    ☐ The Set Enable State on Multiple Controls VI is provided for you in the Temperature Limit project.

    ☐ Drag two copies of the VI from the Project Explorer window onto the Temperature Limit block diagram and complete the block diagram as shown in Figure 4-18.

Figure 4-18. Temperature Limit Block Diagram Complete



1 VI Server References—Create references for the Delta t (ms) control, the High Limit control, and the Start button control.
   ☐ Right-click each of the controls and select **Create»Reference**.
   ☐ The High Limit control is in the Timeout event case.
2 Build Array—Expand the node to accept three inputs.
3 Set Enable State on Multiple Controls VI—This subVI disables the specified controls when the user clicks on the Start button.

Test

1. Run the Temperature Limit VI and verify that it behaves as expected.

2. Save and close all open VIs and the Temperature Limit project.

End of Exercise 4-3

Exercise 6-1 Refactoring a VI

This exercise consists of five VIs that you can evaluate for ways to improve. Look over each option and choose one or two to complete during the time allotted in class. The options are listed from easiest to hardest in terms of refactoring.

Select from the following options to practice refactoring LabVIEW code:

- SubVIs to For Loops
- Array Manipulation
- Polling to Events
- Format Into String
- String Formatting

SubVIs to For Loops

Goal

To take an existing VI and make it more readable, scalable, and maintainable.

Description

In the course of the development of a LabVIEW application, there are times when VIs or sections of VIs end up being written “badly”.

Scenario

Your customer is a research facility that is doing experiments on superconducting material. The researchers must perform experiments at very low temperatures. The materials are tested in a chamber that contains four temperature sensors spread throughout the chamber. The sensors return temperatures in °C. Due to the low temperatures involved, the temperatures in °C are less readable than K. For this reason the customer’s application already includes a VI that converts the temperatures from °C to K.

The customer has recently decided to monitor more than four temperatures. He is worried that every time he increases the number of temperatures he will have to update the VI that does the conversion. In this exercise you will refactor the conversion VI to make it more scalable. You also will make the VI more readable and maintainable.

Note The Kelvin scale defines Absolute Zero as the lowest temperature possible. No temperature below Absolute Zero is allowed. Absolute Zero is approximately equal to -273 °C. You should build your refactored application to generate errors if the user tries to convert invalid temperatures, for example, temperatures less than -273 °C.

Implementation

Open the Convert Temperatures VI located in the `<Exercises>\LabVIEW Core 2\Refactoring\Use subVIs_ForLoop` and refactor it.

Hints

- Find repeated code and replace it with subVIs.
- Find code that works on a limited number of elements of an array and scale it to work on an unlimited number of elements.
- Clean up the block diagram of the VI to make it readable.
- Organize subVIs and related files in a project.

Test

Test your refactored code to ensure that it works as the original application did. Also ensure that the refactored application generates errors if the user tries to convert invalid temperatures.

Array Manipulation

Goal

Refactor a VI that uses an outdated technique for conditionally separating an array into multiple arrays.

Description

Each release of LabVIEW introduces new features that improve coding efficiencies. Therefore, you might refactor code you inherited from someone who developed the code in an earlier version of LabVIEW.

Scenario

You inherit an old LabVIEW application which uses outdated programming techniques.

Implementation

Open `Separate Array Values.vi` from the Array Manipulation project located in the `<Exercises>\LabVIEW Core 2\Refactoring\Array Manipulation` directory and refactor it.

Hints

Conditional auto-indexing allows you to conditionally build an array within a For Loop.

Test

Test your refactored code to ensure that it works as the original application did. Notice that the input array contains a mix of positive and negative values. After running, the Positive Array contains positive values while the Negative Array contains negative values.

Polling to Events

Goal

To take an existing VI that uses outdated techniques and refactor it to be more readable, scalable, and maintainable.

Description

A lot of existing LabVIEW code was written using practices which were standard and accepted in the past but which were discovered to be less than ideal in terms of readability, scalability, and maintainability.

Scenario

You inherit an old LabVIEW application which performs the following functions:

- Acquire a waveform as a Time Series.
- Calculate the FFT of the waveform (that is, generate the Spectrum).
- Calculate the Max and Min values of the Waveform.

The waveform and spectrum are displayed in separate Waveform Graph indicators as are the Max and Min values.

You are asked to add a feature to calculate the Standard Deviation of the Time Series. You notice that the block diagram of the VI is built in such a way that adding more features makes the block diagram larger and more difficult to read.

Implementation

Open the Waveform Analysis (Polling) VI located in the `<Exercises>\LabVIEW Core 2\Refactoring\Polling to Events` directory and refactor it.

Hints

- Use Events instead of Polling.
- Use Shift Registers instead of Local Variables.
- Use a Project to organize the files.

Test

Test your refactored code to ensure that it works as the original application did.

Format Into String

Goal

Refactor a VI that uses the Format Into String function to make the VI more scalable.

Description

The Format Into String function is very versatile; it converts multiple pieces of data into a string according to a format string. However, if new parameters are introduced, both the Format Into String function and the format string must be modified.

You can add parameters without changing the VI if all the parameters are of the same data type.

Scenario

You inherit an old LabVIEW application which is difficult to scale and maintain. You notice that it uses individual controls to input information into a Format Into String function.

Implementation

Open `Format Gas Params.vi` from the Format Gas Parameters project located in the `<Exercises>\LabVIEW Core 2\Refactoring\Format Into String` directory and refactor it.

☐ Assume you need to add a new DBL parameter (for example, Explosiveness).

☐ Notice that the Format Into String node needs to be expanded.

☐ Also notice that the format string needs to have `\r\nExplosiveness:\s%f` added.

Hints

- Arrays of parameter values are easily expandable to include future values.
- Some string functions can handle arrays.

Test

Test your refactored code to ensure that it works as the original application did.

String Formatting

Goal

Refactor a VI that uses the Format Into String function to make it more scalable.

Description

The Format Into String function is very versatile; it converts multiple pieces of data into a string according to a format string. However, if new parameters are introduced, both the Format Into String function and the format string must be modified.

Scenario

You inherited some code that creates a file header and includes a series of name-value pairs for your test data. Because the file is expected to be loaded into Excel, each name and value is separated by a tab and terminated with an End of Line character. In addition to time and date information, the file header also includes information contained in a cluster. The cluster element names and values are used in the name-value pairs.

Your manager wants to re-order the name-value pairs so that Date and Time appear first. In the future, you may want to expand the number of elements in the File Header Data cluster from 3 elements to 10 elements. You must update the code to change the order and prepare for future scalability of the cluster elements.

Implementation

Open the Generate File Header VI in the Format File Header project located in the `<Exercises>\LabVIEW Core 2\Refactoring\String Formatting` directory and refactor it.

Hints

- Create a subVI which formats each name-value pair. Separate the name and value using a Tab constant and terminate each pair with an End of Line constant.
- Process the list of name-value pairs. The challenge is to create two parallel arrays, one for names and one for values.
- If all cluster elements are of the same data type, you can convert a cluster to an array using the Cluster to Array function. You can then use a For Loop to process each cluster element.
- Use a control property node to get a list of control references to all the cluster elements. You can then get access to the Label names of the cluster elements. Use that to build an array of names.

Test

Test your refactored code to ensure that it works as the original application did.

End of Exercise 6-1

Exercise 7-1 Preparing Files for Distribution

Goal

Review the Building Applications Checklist and prepare VIs to build a stand-alone application.

Scenario

Review the Building Applications Checklist to assist you in the build process before creating a stand-alone application or installer.

Stand-alone applications in LabVIEW have the **Window Appearance** set to **Top-level application** to enable the front panel to open when the VI runs.

A VI that runs as a stand-alone executable remains in memory when the application finishes running. Therefore, it is necessary to call the Quit LabVIEW function to close the application when the application finishes executing. Placing the Quit LabVIEW function on the block diagram can make editing the application more difficult in the future because LabVIEW exits each time the application finishes.

Design

- Modify the VI Properties to prepare to build a stand-alone application.
- Modify the application to call the Quit LabVIEW function when the code is executed in the run-time system.
- Modify the application to specify a log path relative to the stand-alone executable.

Implementation

Before you build an application, you first prepare the code so that it executes reliably when compiled into an application.

Review the Building Applications Checklist

1. Select **Help»LabVIEW Help** to open the *LabVIEW Help*.
2. Select **Fundamentals»Building and Distributing Applications»Developing and Distributing an Application**.
3. Review the **Preparing to Build the Application** section.

Set Top-Level Application Window

1. Open the Histogram Main VI.

   ☐ If you have hardware connected, open `Histogram.lvproj` in the `<Exercises>\LabVIEW Core 2\Deployment\HW` directory.

   Note If you have hardware installed and have not already completed hardware setup on your system as part of a *LabVIEW Core 1* course exercise, refer to Appendix A, *Setting Up Your Hardware*.

   ☐ If you do not have hardware connected, open `Histogram.lvproj` in the `<Exercises>\LabVIEW Core 2\Deployment\No HW` directory.

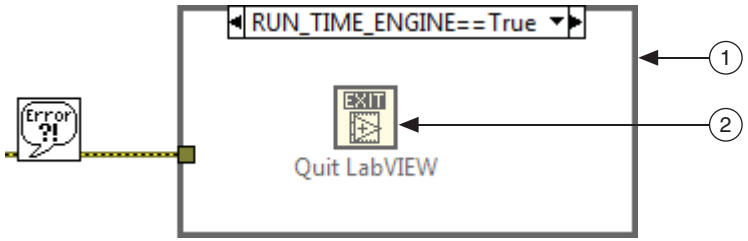
   ☐ In the Project Explorer window, double-click **Histogram Main.vi** to open the VI.

2. Select **File»VI Properties** to display the VI Properties dialog box.
3. Select **Window Appearance** from the **Category** pull-down menu.
4. Enter a name, such as `Histogram Application`, in the **Window Title** text box.
5. Select **Top-level application window** to give the front panel a professional appearance when the VI opens as an executable.
6. Click the **Customize** button to view the various window settings that LabVIEW configures for top-level application windows.
7. Click **OK** to close the **Customize Window Appearance** dialog box and click **OK** to close the **VI Properties** dialog box.
8. Save the VI.

Call the Quit LabVIEW Function

1. Open and modify the block diagram to call the Quit LabVIEW function when the application finishes. The Quit LabVIEW function quits LabVIEW and quits the application after it has executed.

Figure 7-1. Adding the Quit LabVIEW Function to the Block Diagram



1 Conditional Disable Structure—Right-click the border of the structure and select **Add Subdiagram After...** In the Configure Condition dialog, set the value of **Symbol(s)** to `Run_Time_Engine`. Set **Value(s)** to `True`
2 Quit LabVIEW Function—Place this function in the "RUN_TIME_ENGINE==True" subdiagram. You can leave the Default subdiagram empty.

2. In the **Project Explorer** window, select **File»Save All** to save all the VIs.

Specify a File Path Relative to the Executable

The Histogram Main VI already contains code to specify a relative path to the executable application.

Open the Create Data File VI in the Initialize case of the consumer loop. The Application Directory VI creates a path relative to the stand-alone application when you call the VI from a stand-alone application. Otherwise, the Application Directory VI returns the path to the folder containing the project file.

Test

1. Run the Histogram Main VI to ensure that it is working.

Note If you have hardware installed and receive an error, check the configuration of the DAQ Assistant Express VI.

2. Save the VI and the project.

End of Exercise 7-1

Exercise 7-2 Create and Debug a Stand-Alone Application

Goal

Create a build specification, build a stand-alone application (EXE) in LabVIEW, and debug the application running on the local computer.

Scenario

Create a stand-alone application to run the Histogram Main VI. After you prepare your files, you create an Application (`.exe`) Build Specification, and run the application. You then use LabVIEW to debug the running application.

Design

Use the Application (EXE) Build Specifications to create a stand-alone application for the histogram application.

Connect with the running application by creating a debugging session in LabVIEW.

Implementation

Review the Building Applications Checklist

1. Select **Help»LabVIEW Help** to open the *LabVIEW Help*.
2. Select **Fundamentals»Building and Distributing Applications»Developing and Distributing an Application**.
3. Review the **Configuring Specifications for a Built Application** list of steps.

Creating an Application (EXE) Build Specification

1. Open `Histogram.lvproj` in the `<Exercises>\LabVIEW Core 2\Deployment` directory.
2. Right-click **Build Specifications** in the **Project Explorer** window and select **New»Application (EXE)** from the shortcut menu.
3. (Optional) Place a checkmark in the **Do not prompt again for this operation** checkbox and click the **OK** button if you receive a prompt about SSE2 optimization.
4. Modify the filename of the target and destination directory for the application in the **Information** category.
   - ☐ Select the **Information** category.
   - ☐ Change the **Target filename** to `HistogramData.exe`.
   - ☐ Enter `<Exercises>\LabVIEW Core 2\Deployment\Executable` in the **Destination directory** text box.

Tip You do not need to create the directory. LabVIEW creates any directories that you specify.

5. Specify the top-level VI for the application.
   - ☐ Select the **Source Files** category.
   - ☐ Select the **Histogram Main.vi** in the **Project Files** tree.
   - ☐ Click the right arrow next to the **Startup VIs** listbox to add the selected VI to the **Startup VIs** listbox.
6. Include code to allow debugging of the executable.
   - ☐ Select the **Advanced** category.
   - ☐ Place a checkmark in the **Enable debugging** checkbox.
   - ☐ Click **OK**.
7. In the **Project Explorer** window, select **File»Save All**.
8. In the **Project Explorer** window, right-click the **My Application** build specification and select **Build** from the shortcut menu.
9. Click **Done** in the Build status window.

Running the Application Executable

1. Close the Histogram Project Explorer window and close LabVIEW.
2. Navigate to `<Exercises>\LabVIEW Core 2\Deployment\Executable` in Windows Explorer.
3. Run `HistogramData.exe`.
   - ☐ Click the **Snapshot** button.
   - ☐ Click the **Stop** button when done.
4. Verify that the application closed when you stopped the application and the application created a text file in the Logged Data folder within the Executable folder.

Debugging the Executable on the Same Computer

1. Launch LabVIEW.
2. Run `HistogramData.exe`.
3. Select **Operate»Debug Application or Shared Library** from the LabVIEW window.
4. Enter `localhost` in the **Machine name or IP address** text box.

5. Select **HistogramData.exe** from the **Application or shared library** drop-down menu.
   - ☐ Click the **Refresh** button if HistogramData.exe does not appear in the list.
6. Click the **Connect** button to create the debugging connection.
7. Start debugging the running application.
   - ☐ Open the block diagram.
   - ☐ Turn on Execution Highlighting.
   - ☐ Try using probes, breakpoints, and single-stepping.
8. Stop the application by clicking the **Stop** button in the debugging window.

End of Exercise 7-2

Exercise 7-3 Create an Installer

Goal

Create an installer build specification and build the installer. As a challenge, remotely debug the application created by the installer.

Scenario

Creating an installer simplifies deploying an application on multiple machines. After you have prepared your files, you create an Application (`.exe`) Build Specification and then create an Installer Build Specification.

Design

Use an Installer Build Specification to create an installer for the Application (`.exe`) Build Specification you created in Exercise 7-2.

Implementation

Review the Building Applications Checklist

1. Open the LabVIEW Help by selecting **Help»LabVIEW Help**.
2. Select **Fundamentals»Building and Distributing Applications»Building Applications Checklist**.
3. Review the **Distributing Builds** checklist items.

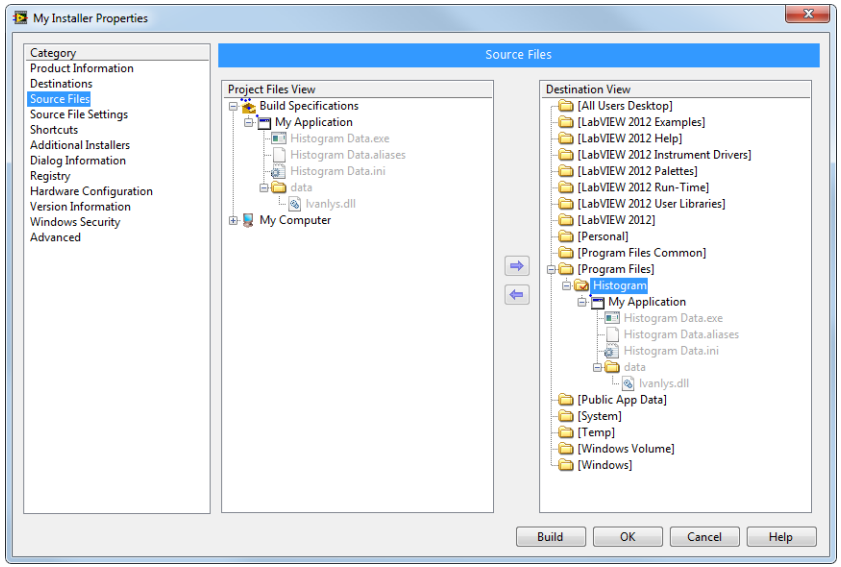
Creating an Installer Build Specification

1. Right-click **Build Specifications** in the **Project Explorer** window and select **New»Installer** from the shortcut menu.
2. Modify the installer destination in the **Product Information** category.
   - ☐ Select the **Product Information** category.
   - ☐ Type `<Exercises>\LabVIEW Core 2\Deployment\Installer` as the Installer destination.
3. Specify the Executable Build Specification.
   - ☐ Click the **Source Files** category.
   - ☐ Select the **My Application** build specification.

☐ Select **Program Files»Histogram** in the **Destination View** tree.

☐ Click the right arrow next to the **Project Files View** tree to place the histogram executable and all executable support files under **Program Files» Histogram** in the **Destination View** tree, as shown in Figure 7-2.

Figure 7-2. Installer Source Files Category



4. Add the NI LabVIEW Run-Time Engine to the installer by modifying the **Additional Installers** category.

   ☐ Select the **Additional Installers** category.

   ☐ Select the **NI LabVIEW Run-Time Engine 2014** installer.

   ☐ Click **OK**.

Note Some additional installers you might select in the **Additional Installers** category, such as DAQmx, can require you to have the product DVD (such as the NI Device Driver DVD) or a downloaded installation package (such as the NI-DAQmx Run-Time Engine) before you can include the product installer in your application installer. To download an installation package, go to `ni.com/info` and enter the code `downloads`.

5. In the **Project Explorer** window, right-click the **My Installer** build specification and select **Build** from the shortcut menu.

6. Click **Done** when LabVIEW finishes building the installer.

Test

1. Run the `setup.exe` file in the `<Exercises>\LabVIEW Core 2\Deployment\Installer\Volume` directory.
2. Follow the instructions on-screen to install the application. By default, the executable is created inside the `<Program Files>\Histogram` directory.
3. From the Start menu, navigate to **Start»Programs»Histogram**.
4. Right-click **Histogram Data** and select **Run as administrator**.

Note You must run the application as an administrator because the executable creates a file in the `<Program Files>` folder. If you do not run the program as an administrator, Error 8 occurs.

5. Click **Yes** in the dialog box asking for permission to make changes to the computer.

Challenge

If you have internet access during class, try to debug the executable on a remote computer.

1. Verify that classroom has internet access.
2. Decide whether to debug a classmate's application or install your application on your classmate's computer.
3. If you decide to debug your own application on a remote computer you must distinguish your application from the applications already on your classmate's computer.
   - ☐ In the installer build specification, rename your application with a unique name.
   - ☐ Transfer your installer to the remote computer using a USB flash drive or the network.
   - ☐ Install your application.
4. To use LabVIEW on your computer to debug a running application on a remote computer, you must determine the IP address of the remote computer, also known as the Destination computer.

Note Consider your computer to be the **Development** computer and your classmate's computer to be the **Destination** computer.

   - ☐ Open the Windows Start menu on the Destination computer.
   - ☐ Enter `cmd` in the search box and press the <Enter> key.
   - ☐ Type `ipconfig` at the prompt in the Command window and press the <Enter> key.
   - ☐ Note the IP address.
5. Run the application on the Destination computer.

6. On the Development computer, launch LabVIEW, if necessary.
7. Select **Operate»Debug Application or Shared Library** from the LabVIEW menu.
8. Enter the IP address of the Destination computer in the **Machine name or IP address** text box.
9. Select the executable from the **Application or shared library** drop-down menu.
   - ☐ Click the **Refresh** button if the executable you want does not appear in the list.
10. Click the **Connect** button to create the debugging connection.
11. Start debugging the running application.
    - ☐ Open the block diagram.
    - ☐ Turn on Execution Highlighting.
    - ☐ Try using probes, breakpoints, and single-stepping.
12. Stop the application by clicking the **Stop** button in the debugging window.

End of Exercise 7-3