LabVIEW™ Core 2 Participant Guide

Course Software Version 2014
November 2014 Edition
Part Number 326293A-01

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.

Table of Contents

Student Guide

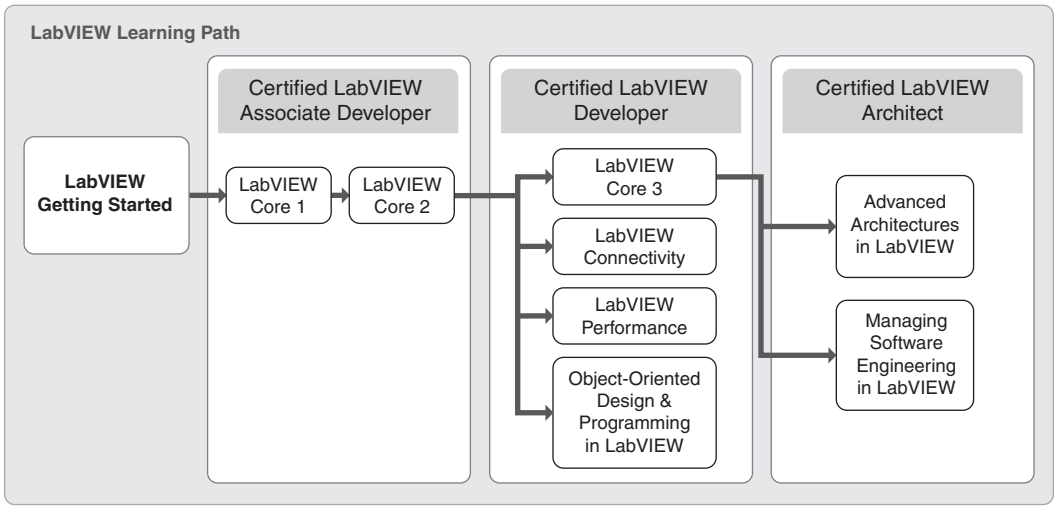
In this section you will learn about the LabVIEW Learning Path, the course description, and the items you need to get started in the LabVIEW Core 2 course.

Topics

+ NI Certification
+ Course Description
+ What You Need to Get Started
+ Installing the Course Software
+ Course Goals

A. NI Certification

The LabVIEW Core 2 course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for the NI Certified LabVIEW Associate Developer exam (CLAD). The following illustration shows the courses that are part of the LabVIEW training series. Refer to `ni.com/training` for more information about NI Certification.



B. Course Description

The LabVIEW Core 2 course teaches you programming concepts, techniques, features, VIs, and functions you can use to create test and measurement, data acquisition, instrument control, datalogging, measurement analysis, and report generation applications. This course assumes that you are familiar with Windows and that you have experience writing algorithms in the form of flowcharts or block diagrams.

The Participant Guide is divided into lessons. Each lesson contains the following:

- An introduction with the lesson objective and a list of topics and exercises.
- Slide images with additional descriptions of topics, activities, demonstrations, and multimedia segments.
- A set of exercises to reinforce topics. Some lessons include optional and challenge exercises.
- A lesson review that tests and reinforces important concepts and skills taught in the lesson.

Note For course and exercise manual updates and corrections, refer to `ni.com/info` and enter the Info Code `core2`.

Several exercises use a plug-in multifunction data acquisition (DAQ) device connected to a DAQ Signal Accessory or BNC-2120 containing a temperature sensor, function generator, and LEDs. If you do not have this hardware, you still can complete the exercises. Alternate instructions are provided for completing the exercises without hardware. You also can substitute other hardware for those previously mentioned. For example, you can use another National Instruments DAQ device connected to a signal source, such as a function generator.

C. What You Need to Get Started

Before you use this course manual, make sure you have all of the following items:

- ☐ Computer running Windows 7/Vista/XP
- ☐ Multifunction DAQ device configured as Dev1 using Measurement & Automation Explorer (MAX)
- ☐ DAQ Signal Accessory or BNC-2120, wires, and cable
- ☐ LabVIEW Professional Development System 2014 or later
- ☐ DAQmx 14.0 or later
- ☐ *LabVIEW Core 2* course CD, from which you install the following folders:

Directory	Description
`Exercises`	Contains VIs used in the course
`Solutions`	Contains completed course exercises

D. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer. The **LabVIEW Core 2 Course Setup** dialog box appears.
2. Click **Install the course materials**.
3. Follow the onscreen instructions to complete installation and setup.

Exercise files are located in the `<Exercises>\LabVIEW Core 2\` folder.

Note Folder names in angle brackets, such as `<Exercises>`, refer to folders on the root directory of your computer.

E. Course Goals

This course prepares you to do the following:

- Apply common design patterns that use queues and events
- Use event programming effectively
- Programmatically control user interface objects
- Evaluate file I/O formats and use them in applications
- Modify existing code for improved usability
- Prepare, build, debug, and deploy stand-alone applications

This course does *not* describe any of the following:

- LabVIEW programming methods covered in the *LabVIEW Core 1* course
- Every built-in VI, function, or object. Refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course.
- Developing a complete application for any student in the class. Refer to the NI Example Finder, available by selecting **Help»Find Examples**, for example VIs you can use and incorporate into VIs you create.

1 Using Variables

In this lesson you will learn to recognize when to use local and global variables and be able to determine the result of block diagrams that use variables.

Topics

+ Variables
+ Using Variables Appropriately
+ Race Conditions

Exercises

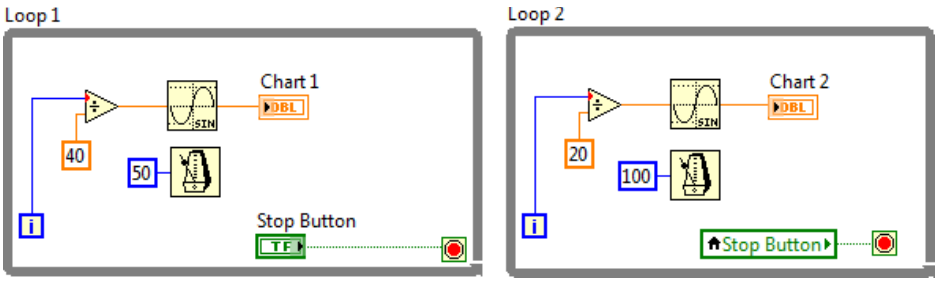
Exercise 1-1 Weather Station UI VI with Local Variables

A. Variables

Objective: Identify the differences between local and global variables.

Definition

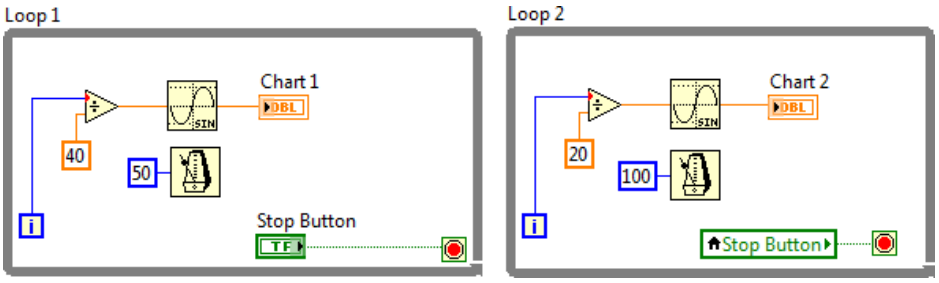
Variables Block diagram elements that access or store data in another location

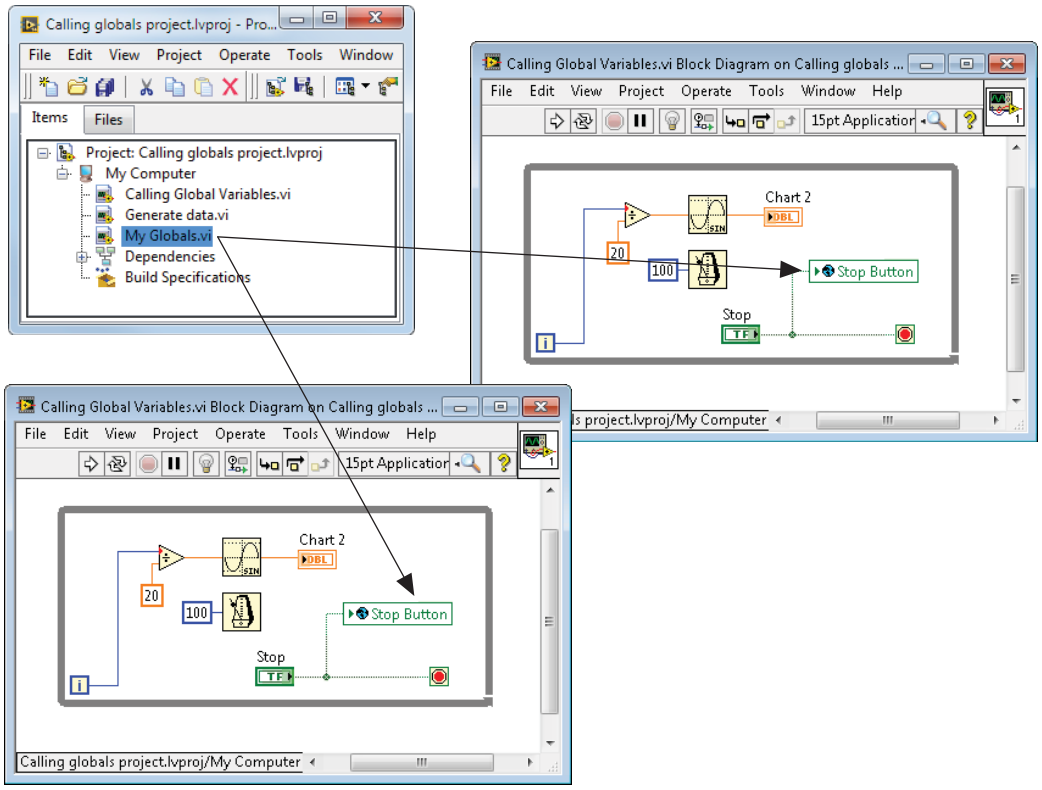


Local versus Global Variables

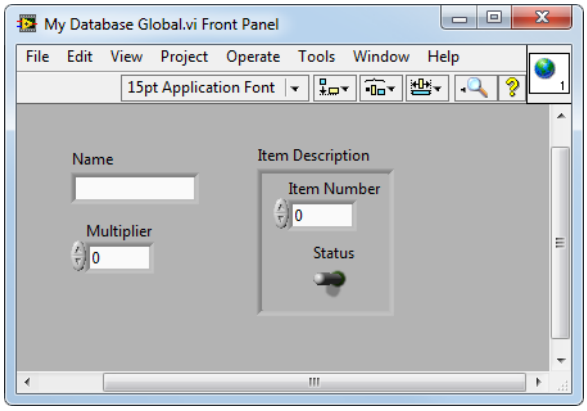
Local and global variables pass information between locations in the application that you cannot connect with a wire. The following table outlines the differences between local and global variables.

Local Variable	Global Variable
Stop Button	Stop Button
Store data in front panel controls and indicators.	Store data in special repositories that you can access from multiple VIs.
Use to access front panel objects from more than one location in a single VI.	Use to access and pass data among several VIs.
Has a one-to-one relationship with a control or indicator.	Can contain one or more variable data types.



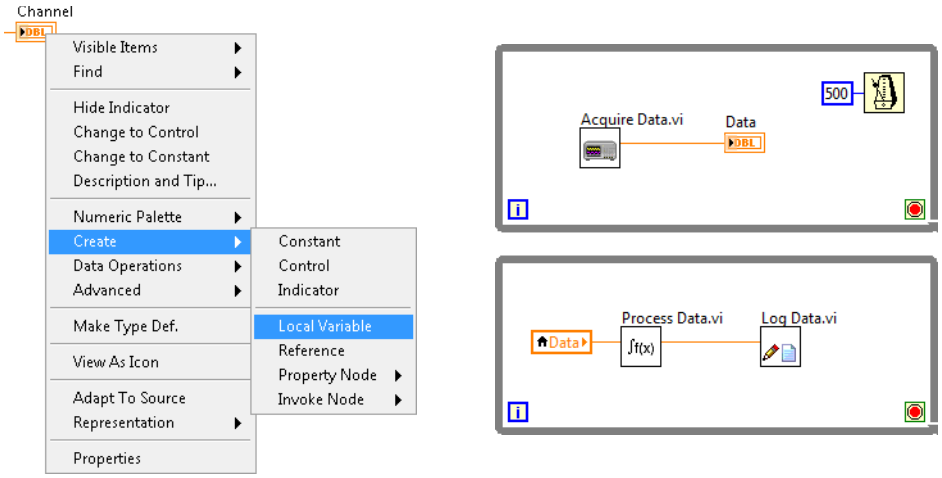


The following figure shows a global VI front panel window with a numeric, a string, and a cluster containing a numeric and a Boolean control. The toolbar does not show the **Run**, **Stop**, or related buttons as it would in a normal front panel window.



Multimedia: Using Variables

Complete the multimedia module, *Using Variables*, available in the `<Exercises>\LabVIEW Core 2\Multimedia\` folder.



B. Using Variables Appropriately

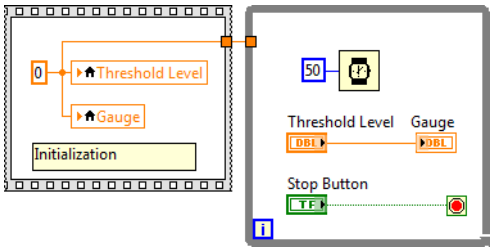
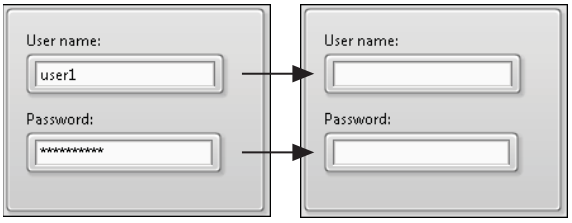
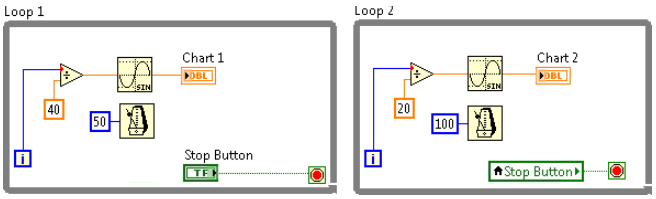
Objective: Recognize issues that variables can solve and drawbacks to using them.

All variables allow you to circumvent normal data flow by passing data from one place to another without connecting the two places with a wire. For this reason, variables are useful in parallel architectures, but also have certain drawbacks, such as race conditions.

When to Use Local Variables

The following table highlights use cases for local variables.

Table 1-1. Use Cases for Local Variables

Initializing front panel controls or indicators. For example, you need to set data in a terminal in one location, but access the terminal from another location on the block diagram. Because each front panel object has only one block diagram terminal you can wire, you need to use variables.	
Writing to controls. For example, to clear a log in screen you need to modify the front panel control while the VI is running. You use variables to programmatically write empty strings to the log in controls after a user enters their information.	
Keeping parallel loops parallel. For example, if you pass the data using a wire, the loops are no longer parallel. Variables let you pass data between multiple loops without creating a data dependency.	

Drawbacks of Variables

Local and global variables are inherently not part of the LabVIEW dataflow execution model. The following table highlights some of the potential programming issues with using variables.

Table 1-2. Potential Programming Issues with Variables

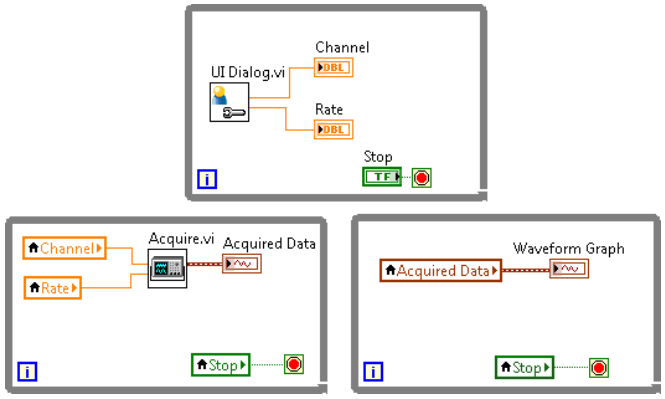
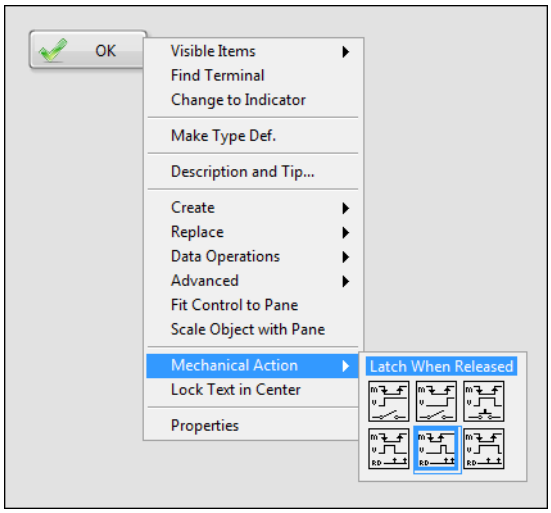
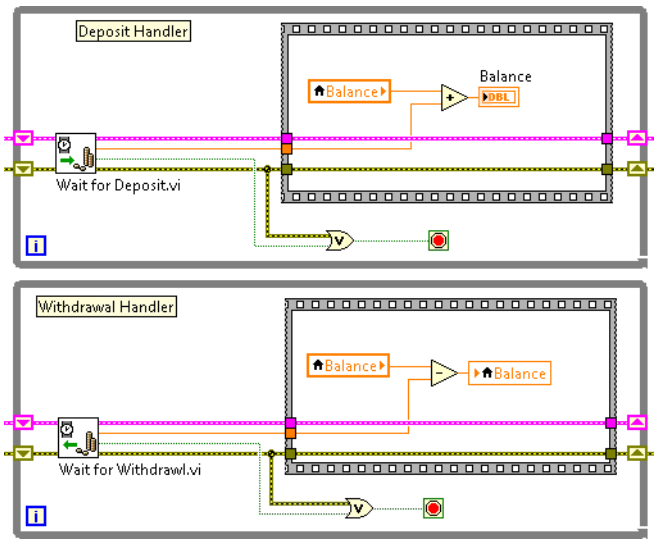
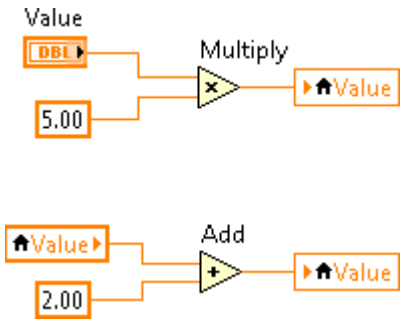
Less readable block diagram code Because variables break the dataflow model, you cannot use wires to follow the flow of data.	
Limited Boolean mechanical actions Boolean controls associated with variables must use a switch mechanical action. If a Boolean control has an associated local variable, it cannot use a latch mechanical action because the first local variable to read the Boolean control with latch action would reset its value to the default, which is not the expected behavior.	

Table 1-2. Potential Programming Issues with Variables (Continued)

Unexpected behavior in VIs Using variables instead of a connector pane or using variables to access values in each frame of a sequence structure is a bad practice and can cause unexpected behavior in VIs. Because variables only contain the latest value, you could have potential data loss if you fail to synchronize data operations properly.	
Slow performance Overusing local and global variables, such as using them to avoid long wires across the block diagram or using them instead of dataflow, can slow performance because each instance of a local or global variable makes a copy of the data in memory.	
Race conditions Refer to the *Race Conditions* section for more information.	

Tip Refer to the *Using Local and Global Variables Carefully* topic in the *LabVIEW Help* for more information about potential issues with using local variables.

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

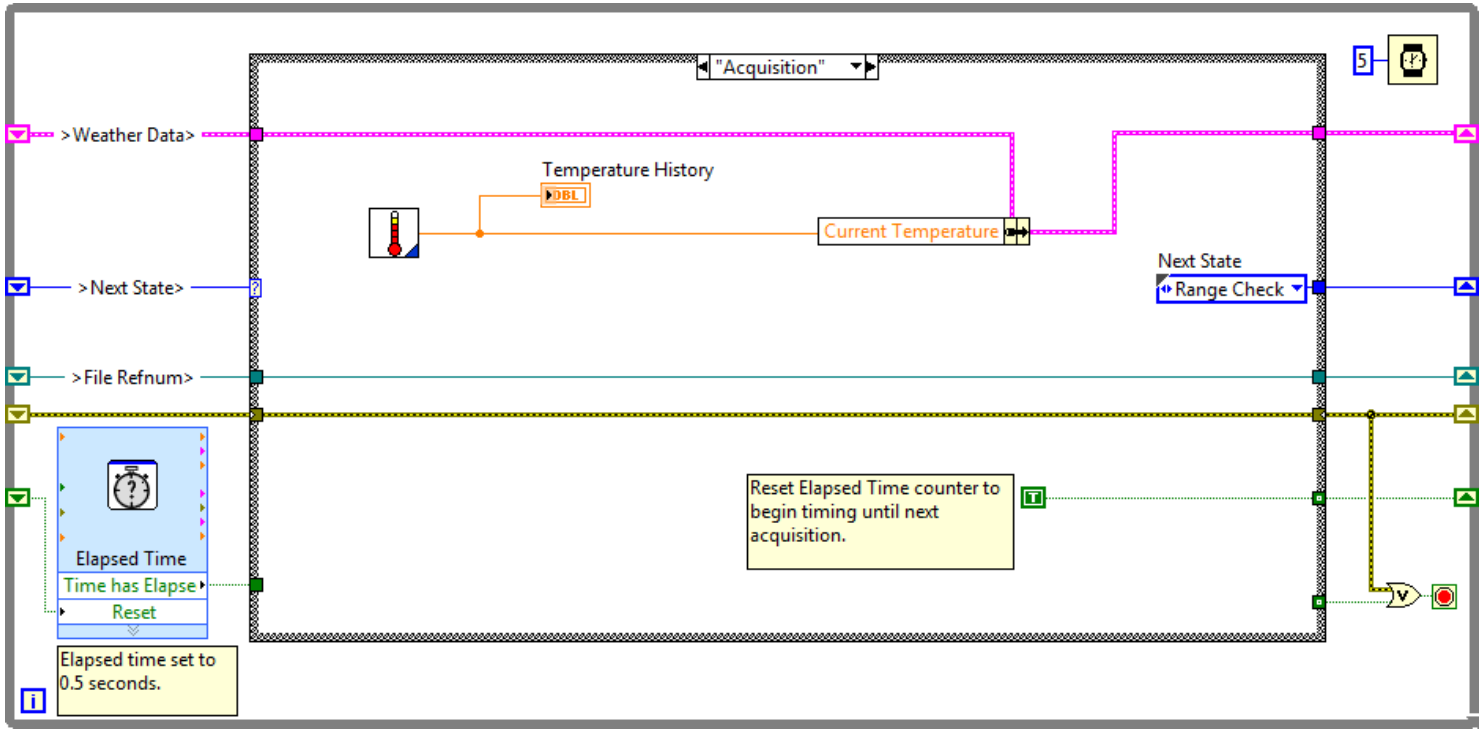
   d. Save and close the control.
   e. Open **Weather Station UI.vi** from the **Project Explorer** window.

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.

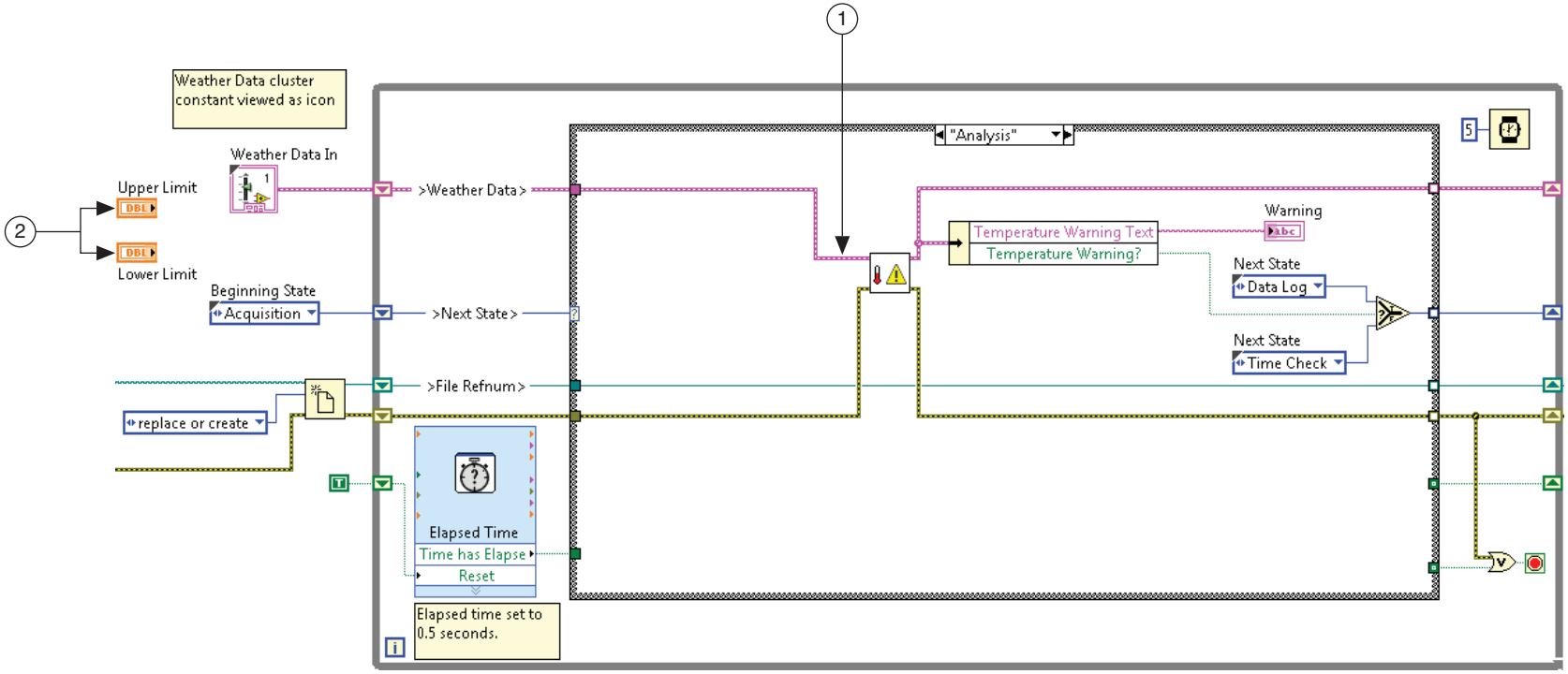
3. Set the **Next State** enum in the Acquisition case to Range Check as shown in Figure 1-1.

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

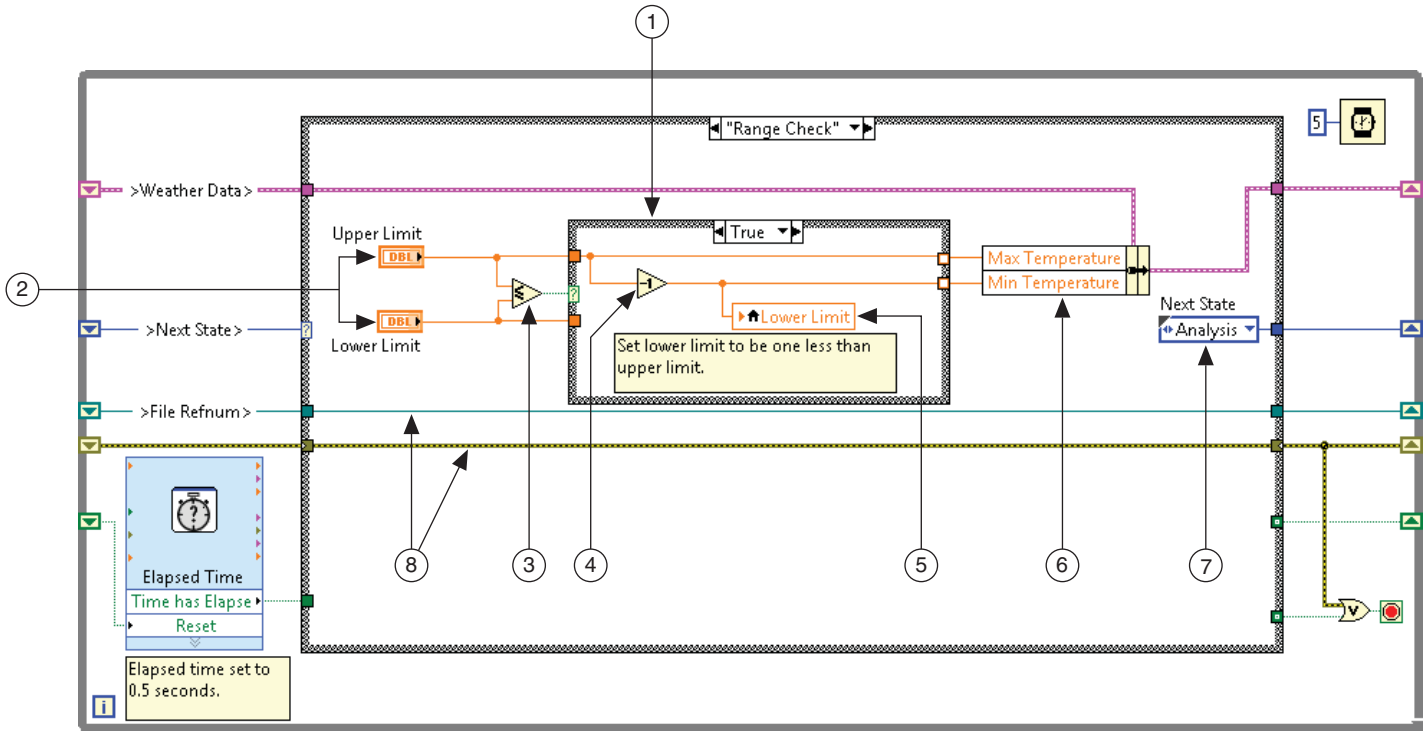


1 Delete the Bundle By Name function and wire the Weather Data wire directly to the Temperature Warnings VI. Press <Ctrl-B> to delete the broken wires from the **Upper Limit** and **Lower Limit** controls.
2 Move the **Upper Limit** and **Lower Limit** controls outside the While Loop.

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

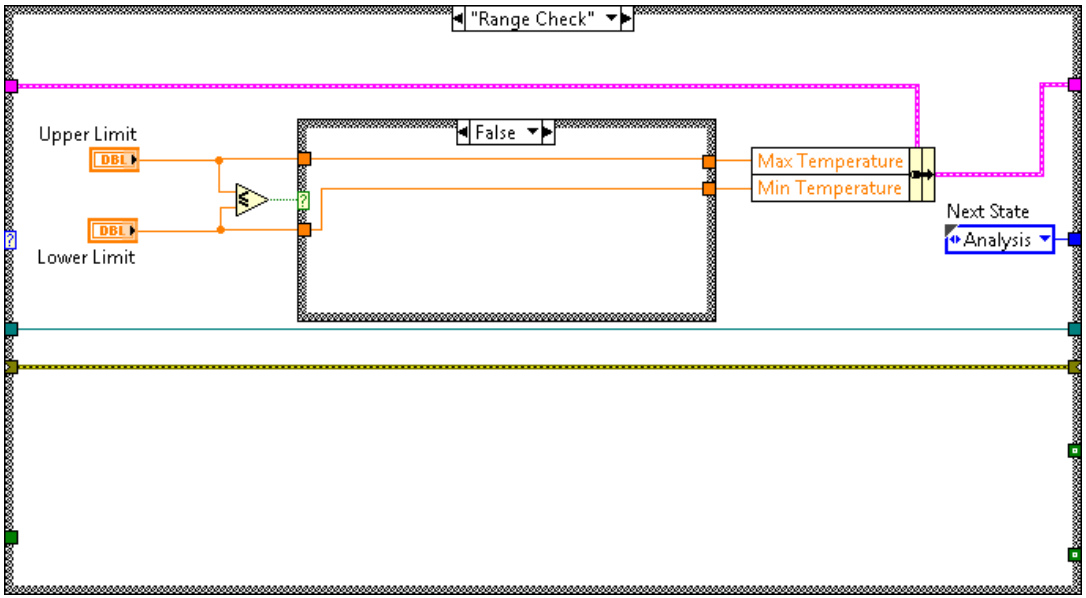


1 **Case Structure**—Place a Case structure inside the Range Check case.
2 Move the **Upper Limit** and **Lower Limit** controls into the Range Check case.
3 **Less or Equal?**—Compares upper limit and lower limit values. Because the Less or Equal? function is wired to the case selector of the inner 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

C. Race Conditions

Objective: Identify race conditions and avoid them.

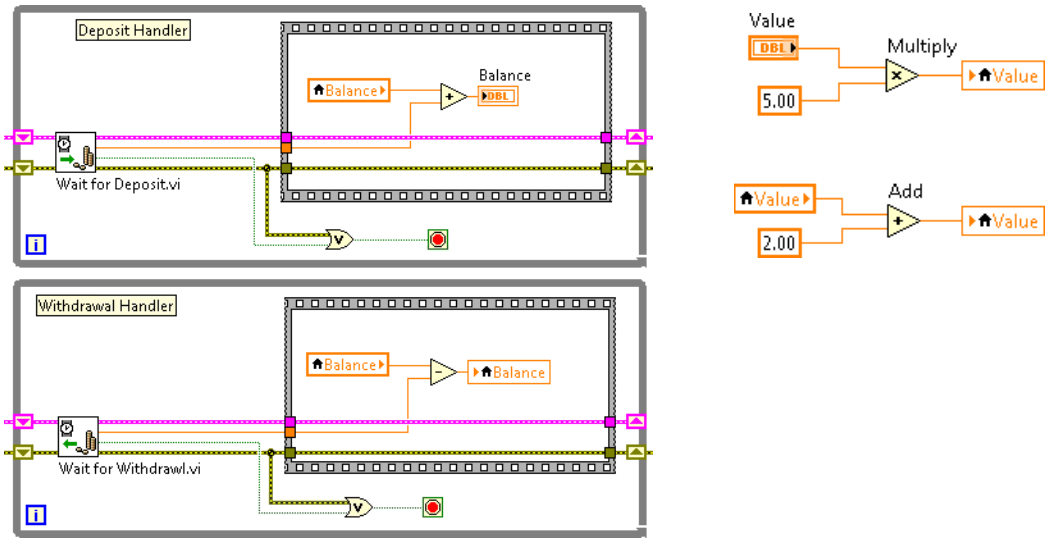
What is a Race Condition?

A race condition occurs when the timing of events or the scheduling of tasks unintentionally affects an output or data value.

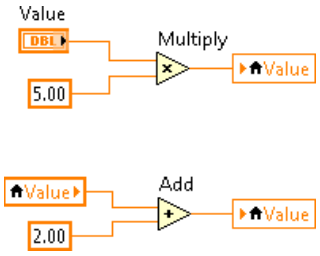
Race conditions commonly occur in programs that:

- Execute multiple tasks in parallel.
- Share data between parallel tasks using variables.
- Have two tasks with both read and write access to the same resource, such as a variable or file.

When a data dependency is not established, LabVIEW can schedule tasks in any order, which creates the possibility for race conditions if the tasks depend upon each other.



The code in this example has four possible outcomes, depending on the order in which the operations execute.



Outcome 1: Value = (Value × 5) + 2

1. Terminal reads Value.
2. Value × 5 is stored in Value.
3. Local variable reads Value × 5.
4. (Value × 5) + 2 is stored in Value.

Outcome 2: Value = (Value + 2) × 5

1. Local variable reads Value.
2. Value + 2 is stored in Value.
3. Terminal reads Value + 2.
4. (Value + 2) × 5 is stored in Value.

Outcome 3: Value = Value × 5

1. Terminal reads Value.
2. Local variable reads Value.
3. Value + 2 is stored in Value.
4. Value × 5 is stored in Value.

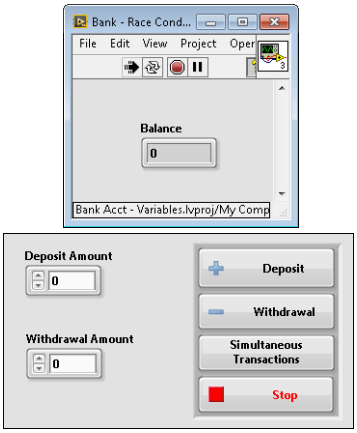
Outcome 4: Value = Value + 2

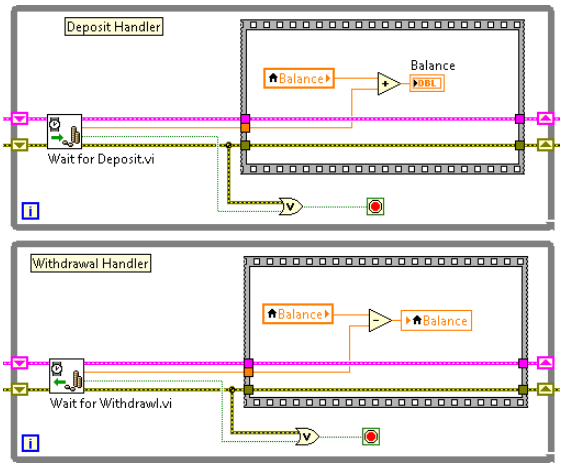
1. Terminal reads Value.
2. Local variable reads Value.
3. Value × 5 is stored in Value.
4. Value + 2 is stored in Value.

Although this code is considered a race condition, LabVIEW usually assigns a consistent order to the operations. However, you should avoid situations such as this one because the order and the behavior of the VI can vary.

Demonstration: Race Conditions - Bank VI

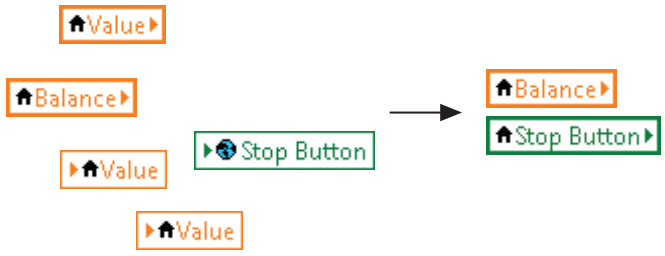
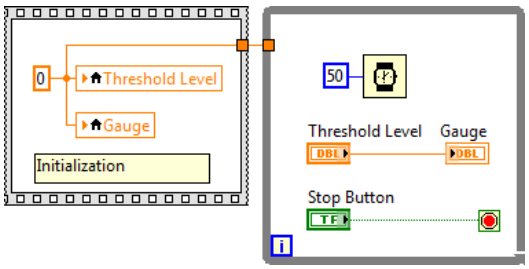
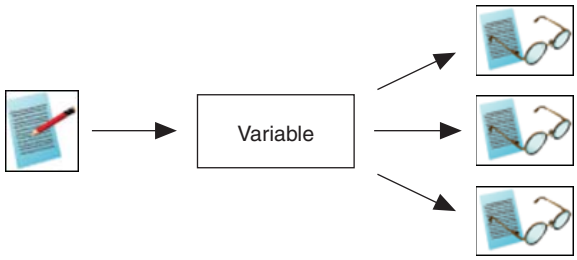
Navigate to `<Exercises>\Demonstrations\Bank VI Race Conditions\` and use Execution Highlighting to explore the race condition that can occur in `Bank.vi`.





Avoiding Race Conditions

The following table illustrates some ways to avoid race conditions.

Reducing the use of variables. Use subVIs, connector panes, and wires to transfer data whenever possible.	
Properly sequencing instructions by specifying an execution order. Use wires, sequences, or other methods to specify data flow.	
Controlling and limiting shared resources. Minimize shared resources and the number of writers to the remaining shared resources. In general, it is not harmful to have multiple readers for a shared resource. However, try to use only one writer or controller for a shared resource to avoid race conditions.	

Additional Resources

Learn More About	LabVIEW Help Topic
Variables	*Local Variables, Global Variables, and the Feedback Node*
	Read and Write Variables
	Local Variables
Race conditions	*Using Local and Global Variables Carefully*

Activity 1-1: Lesson Review

1. You should use variables in your VI whenever possible.
   a. True
   b. False

2. When controlling resources, which combination(s) of writers and readers reduces the chance of race conditions?
   a. One writer, one reader
   b. One writer, multiple readers
   c. Multiple writers, one reader
   d. Multiple writers, multiple readers

Activity 1-1: Lesson Review - Answers

1. You should use variables in your VI whenever possible.
   a. True
   b. False

2. When controlling resources, which combination(s) of writers and readers reduces the chance of race conditions?
   a. One writer, one reader
   b. One writer, multiple readers
   c. Multiple writers, one reader
   d. Multiple writers, multiple readers

2 Communicating Data Between Parallel Loops

In this lesson you will learn how to develop code that synchronizes data between parallel loops and you learn how to decide which communication method is the most appropriate for different scenarios.

Topics

Exercises

A. Introduction

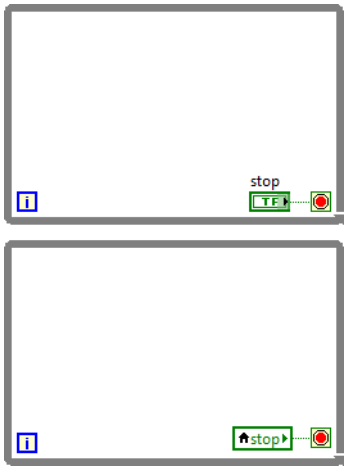
Objective: Review using local variables to communicate between parallel loops.

Communicating Data Between Parallel Loops - Local Variables

You use parallel loops to do the following:

- Execute tasks simultaneously
- Execute tasks at different rates

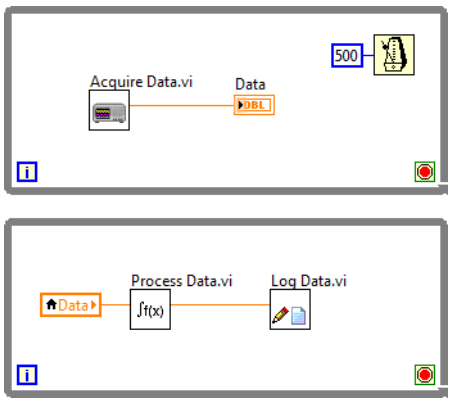
Do you need to transfer only the latest data between loops, such as the value of a Stop button, or do you need to transfer every point of data?



Note Use a variable to communicate data between parallel loops if you need to share only the latest value.

If you need to transfer every point of data between parallel loops, should you use a local variable?

If you need to transfer every point of data between parallel loops, you should not use variables because it is possible to miss or duplicate data.



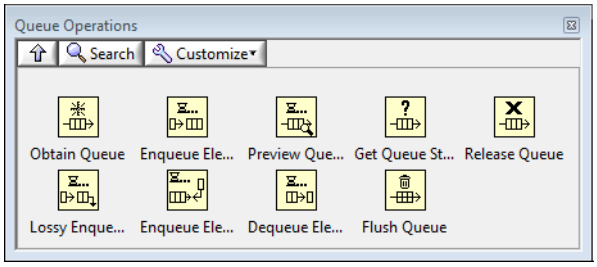
B. Queues

Objective: Demonstrate how to transfer every point of data between parallel loops using queues.

Multimedia: Queues

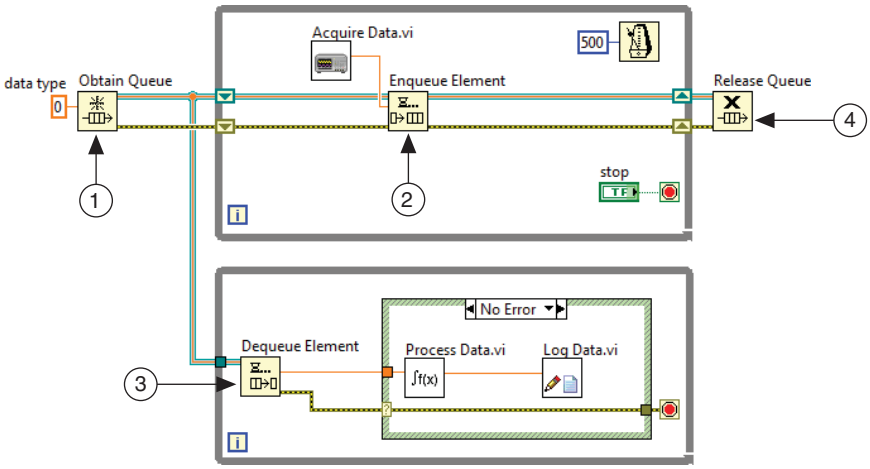
If you need to communicate data between parallel loops and you need to transfer every point of data, you should use queues.

Complete the multimedia module, *Queues*, available in the `<Exercises>\LabVIEW Core 2\Multimedia\` folder.



Communicating Data Between Parallel Loops - Queues

Use queues to transfer every point of data between parallel loops or VIs.



1 Obtain Queue—Creates the queue before the loops begin and defines the data type for the queue.
2 Enqueue Element—Adds data to the queue.
3 Dequeue Element—Removes data from the queue. The loop does not execute until data is available in the queue.
4 Release Queue—Releases the queue.When the queue releases, the Dequeue Element function generates an error, which stops the consumer loop even if more elements are still left in the queue. If you want different functionality for stopping the loops and releasing the queue, you can use different logic and timing than this example.

Benefits of Using Queues

- Loops are synchronized.
- Queues remove the possibility of losing the data.
- Queues do not use polling.

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

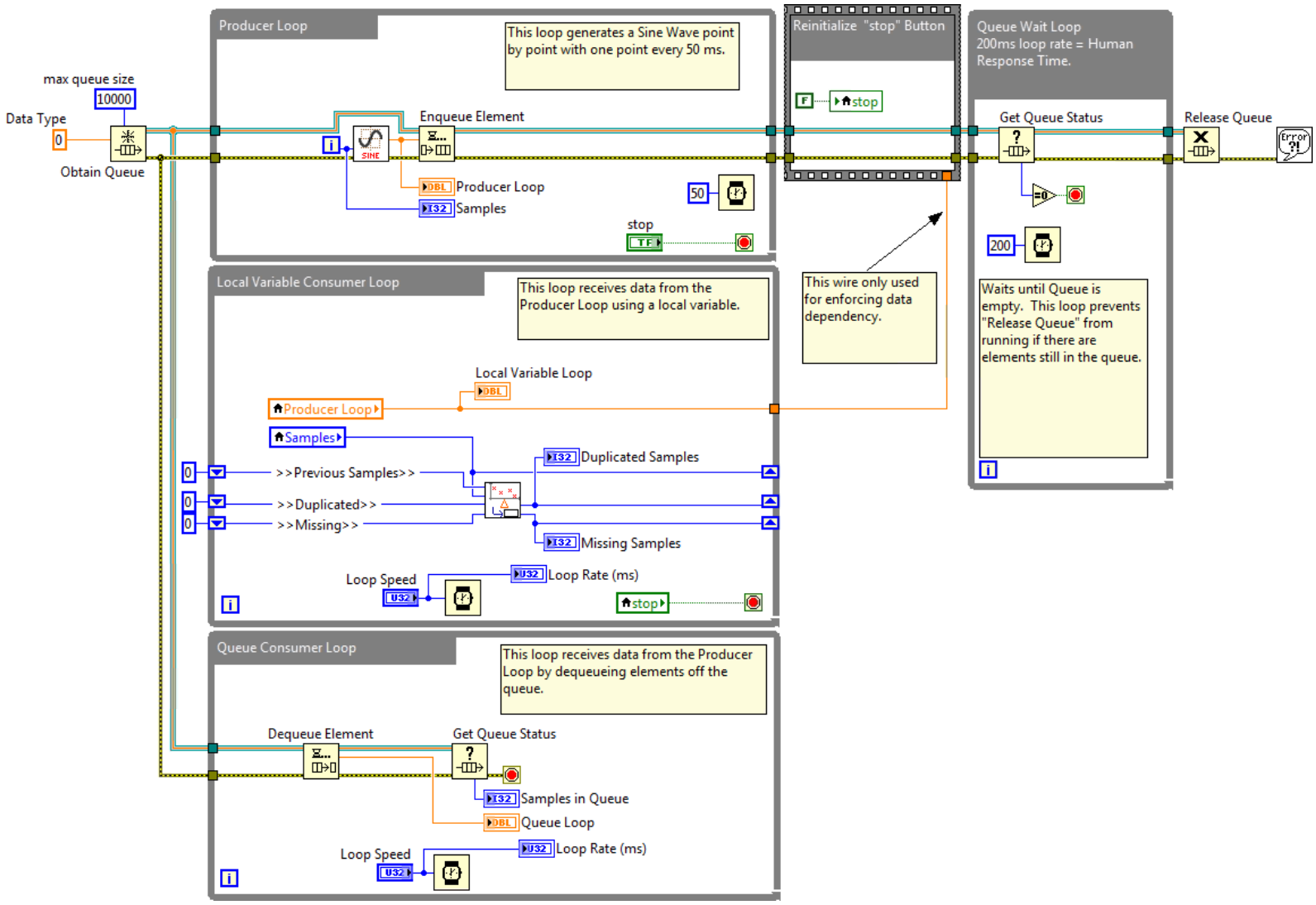
1. Open `Queues vs Local Variables.lvproj` in the `<Exercises>\LabVIEW Core 2\Queues versus Local Variables` directory.
2. Double-click **Queues vs Local Variables.vi** in the **Project Explorer** window to open the VI. The front panel of this VI is shown in Figure 2-1.

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.

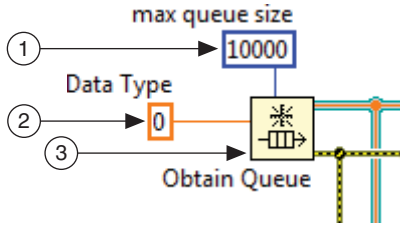
Figure 2-2. Block Diagram of the Queues vs Local Variables VI



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.

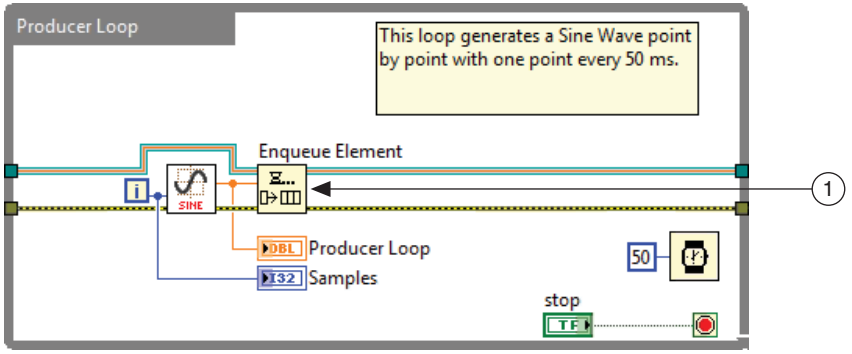
Figure 2-3. Creating the Queue



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

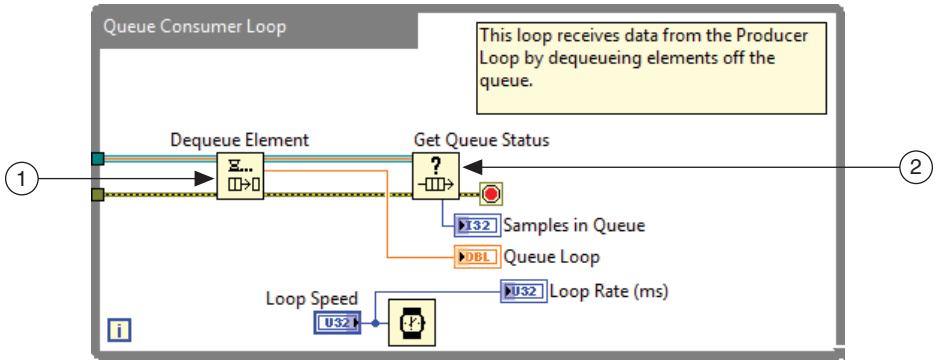
Figure 2-4. Queuing Data the Producer Loop Generates



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

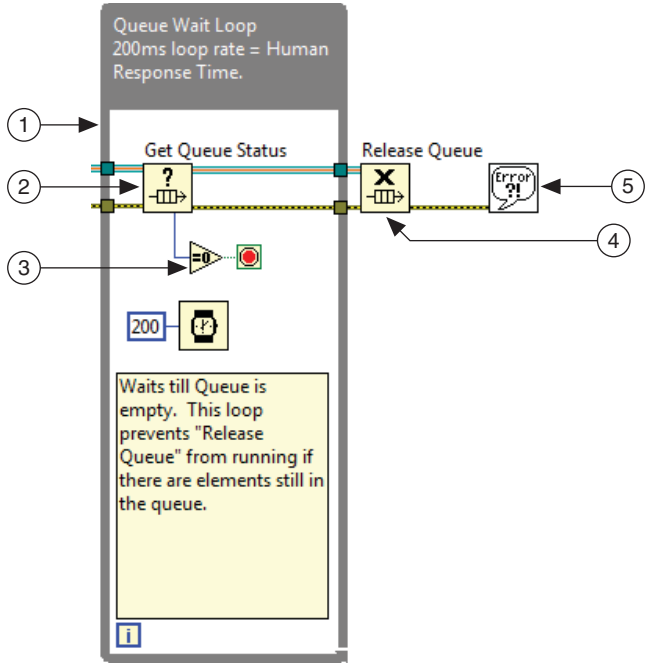
Figure 2-5. Dequeuing Data inside the Consumer Loop



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

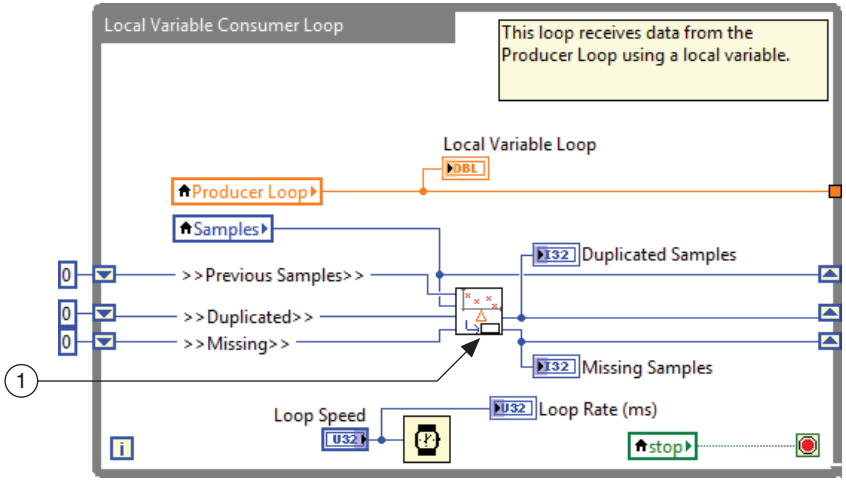


1 While Loop—Waits for the queue to empty before stopping the VI. Refer to this While Loop as the Queue Wait Loop.
2 Get Queue Status—Returns information about the current state of the queue, such as the number of data elements currently in the queue.
3 Equal To 0?—Wired to the stop condition of the 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.

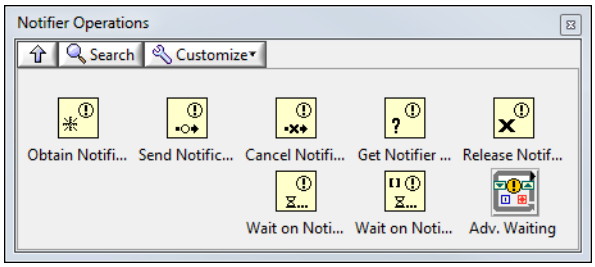
End of Exercise 2-1

C. Notifiers

Objective: Learn how to create code that broadcasts the latest data to waiting parallel loops using notifiers.

What is a Notifier?

Notifier functions Functions that suspend the execution of a block diagram until they receive data from another section of the block diagram or from another VI running in the same application instance.



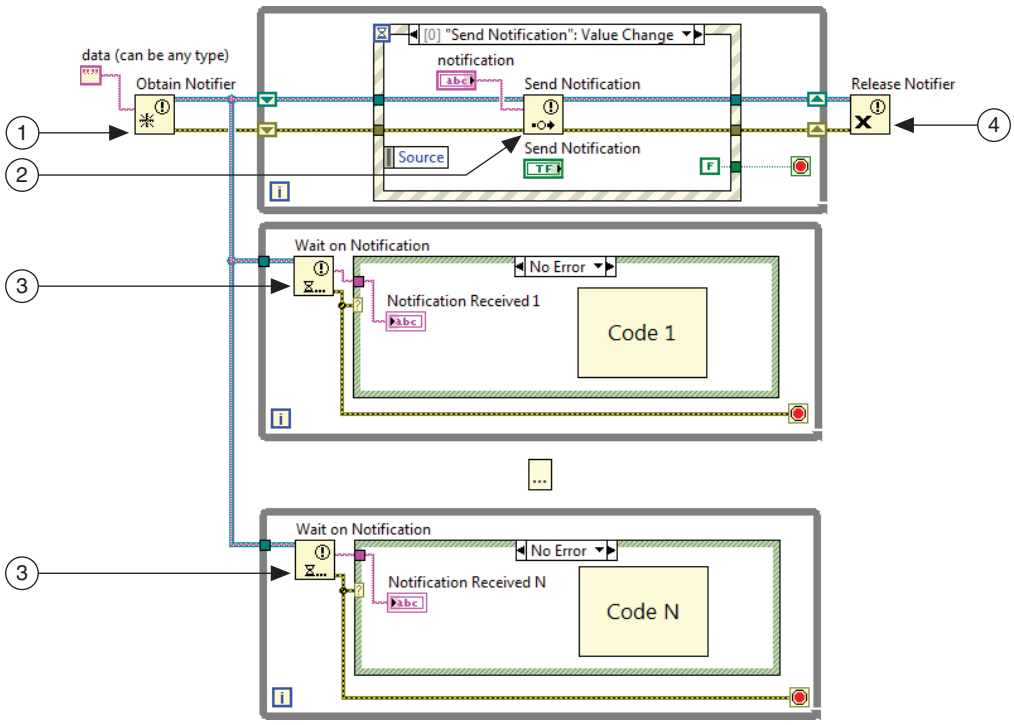
Notifiers vs. Queues

	Notifiers	Queues
Buffer data?	No	Yes
Broadcast data to multiple loops?	Yes	No

Broadcast Data to Parallel Loops

Use Notifiers to communicate between parallel loops when you want one notification to trigger one or more processes and the waiting processes need to receive only the latest notification and data.

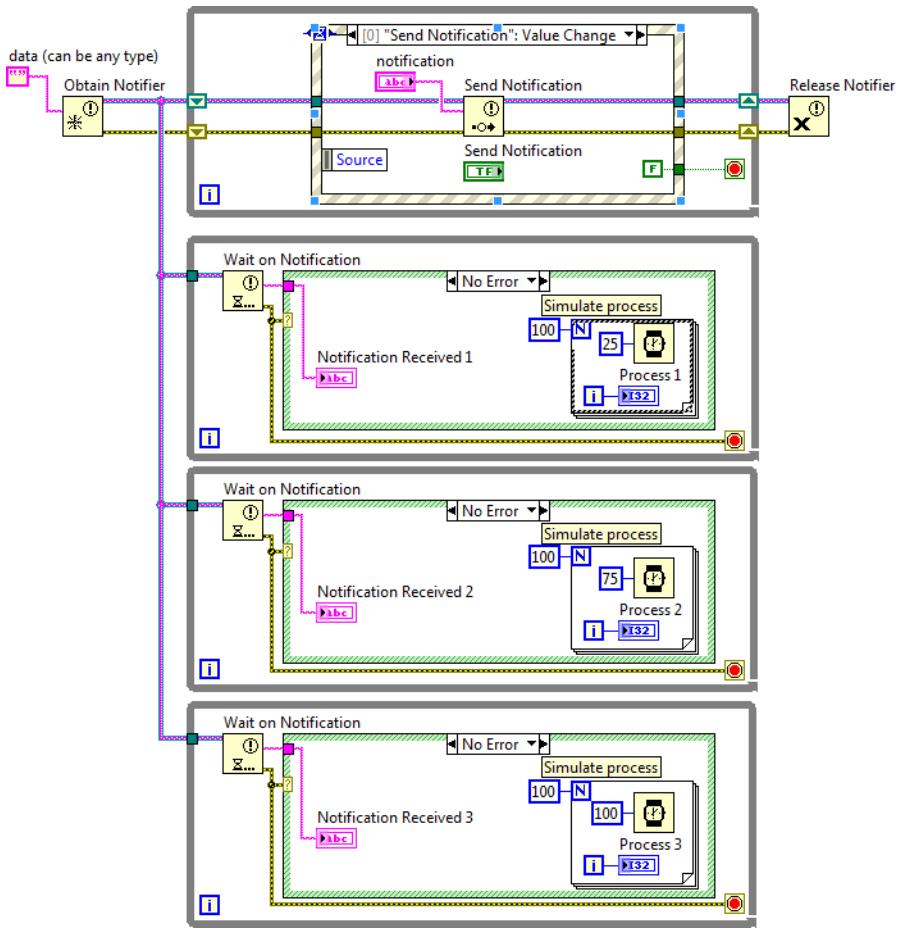
The following figure shows how notifiers can be used to broadcast the same data to multiple waiting loops.



1 Obtain Notifier—Creates the notifier before the loops begin and defines the data type for the notifier.
2 Send Notification—Sends a message to all functions waiting on the notifier.
3 Wait on Notification—Waits until the notifier receives a message. The Wait on Notification function will only receive the latest message.
4 Release Notifier—Releases the notifier. When the notifier releases, the Wait on Notification functions in the bottom loops generates an error, which stops those loops.

Demonstration: Notifiers

Open and run `Notifier scenario.vi` in the `<Exercises>\Demonstrations\Notifiers` directory to explore the functionality of notifiers.



D. Summary

Objective: Compare variables, queues, and notifiers.

	Suspend execution in reader loop?	Buffer data?	Data can be read by multiple loops?	Use case
Local/global variables			Yes	Transfer latest data
Queues	Yes	Yes		Transfer every point of data
Notifiers	Yes		Yes	Transfer latest data to multiple loops that are waiting on notification

Additional Resources

Learn More About	LabVIEW Help Topic
Functions to implement queues	*Queue Operations Functions*
Functions to implement notifiers	*Notifier Operations Functions*
Comparing different communication tools	*Data Communication Methods in LabVIEW*

Activity 2-1: Lesson Review

1. Which of the following buffer data?
   a. Queues
   b. Notifiers
   c. Local Variables

2. Match the following:

Obtain Queue	a. Destroys the queue reference
Get Queue Status	b. Assigns the data type of the queue
Release Queue	c. Adds an element to the back of a queue
Enqueue Element	d. Determines the number of elements currently in the queue

3. Which of the following are valid data types for queues and notifiers?
   a. String
   b. Numeric
   c. Enum
   d. Array of Booleans
   e. Cluster of a string and a numeric

Activity 2-1: Lesson Review - Answers

1. Which of the following buffer data?
   - **a. Queues**
   - b. Notifiers
   - c. Local Variables

2. Match the following:

Obtain Queue	**b. Assigns the data type of the queue**
Get Queue Status	**d. Determines the number of elements currently in the queue**
Release Queue	**a. Destroys the queue reference**
Enqueue Element	**c. Adds an element to the back of a queue**

3. Which of the following are valid data types for queues and notifiers?
   - **a. String**
   - **b. Numeric**
   - **c. Enum**
   - **d. Array of Booleans**
   - **e. Cluster of a string and a numeric**

3 Implementing Design Patterns

In this lesson you will learn how to implement common design patterns for single and parallel loop applications.

Topics

+ Why Use Design Patterns
+ Simple Design Patterns
+ Multiple Loop Design Patterns
+ Functional Global Variable Design Pattern
+ Error Handlers
+ Generating Error Codes and Messages
+ Timing a Design Pattern

Exercises

Exercise 3-1 Group Exercise: Producer/Consumer Design Pattern
Exercise 3-2 User Access Level
Exercise 3-3 Producer/Consumer with Error Handling
Exercise 3-4 Create a Histogram Application

A. Why Use Design Patterns

Objective: Describe the benefits of design patterns.

Design Patterns Code implementations and techniques that are solutions to specific problems in software design.

Benefits of Design Patterns

- Evolve through the efforts of multiple developers.
- Are fine-tuned for simplicity, maintainability, and readability.

B. Simple Design Patterns

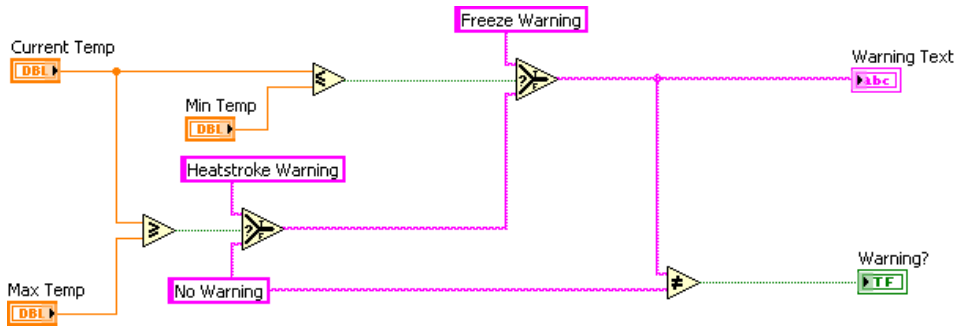
Objective: Describe and use simple design patterns.

There are four types of simple design patterns:

- Simple VI Pattern
- General VI Pattern
- State Machine
- Event-Based State Machine

Simple VI Pattern

The simple VI pattern consists of a single VI that takes a measurement, performs calculations, and either displays the results or records them to disk. Use this architecture for simple applications or for functional components within larger applications.



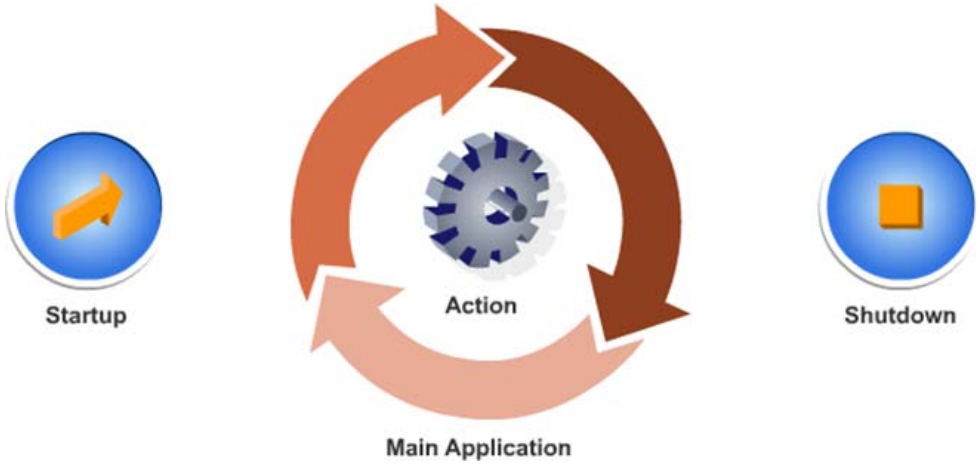
Benefits of the Simple VI Pattern

- Can perform calculations and make quick lab measurements without a complicated architecture.
- Does not require a specific start or stop action from the user. The user just clicks the Run button.
- Can convert these simple VIs into subVIs that you use as building blocks for larger applications.

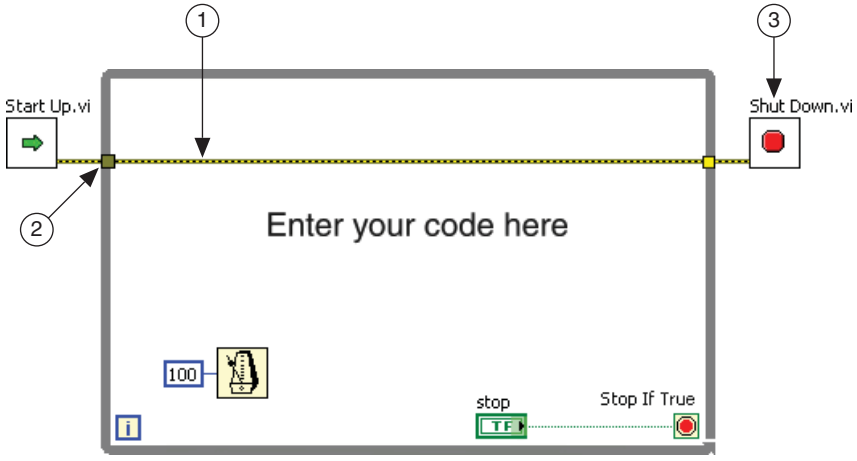
General VI Pattern

The General VI pattern has three main phases:

- **Startup**—Initializes hardware, reads configuration information from files, or prompts the user for data file locations.
- **Main Application**—Consists of at least one loop that repeats until the user decides to exit the program or the program terminates for other reasons such as I/O completion.
- **Shutdown**—Closes files, writes configuration information to disk, or resets I/O to the default state.



General VI Framework

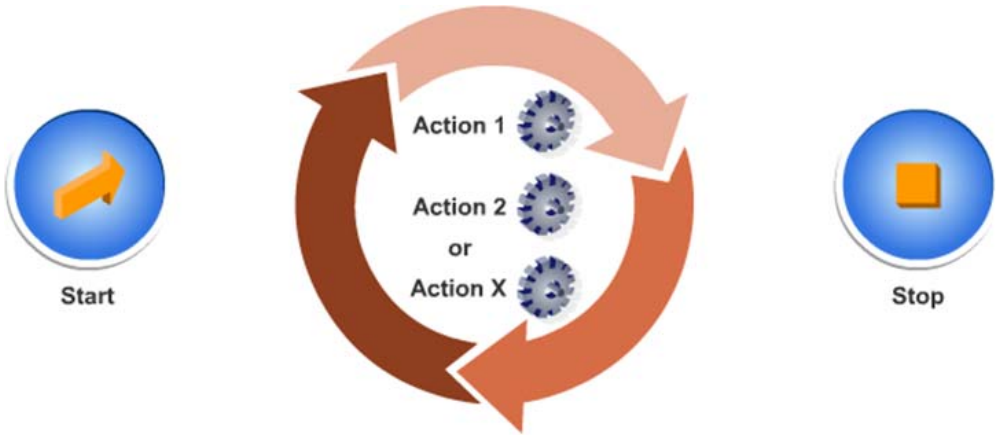


1 The error cluster wires control the execution order.
2 The While Loop does not execute until the Start Up VI finishes running and returns the error cluster data.
3 the Shut Down VI cannot run until the main application in the While Loop finishes and the error cluster data leaves the loop.

State Machine Pattern

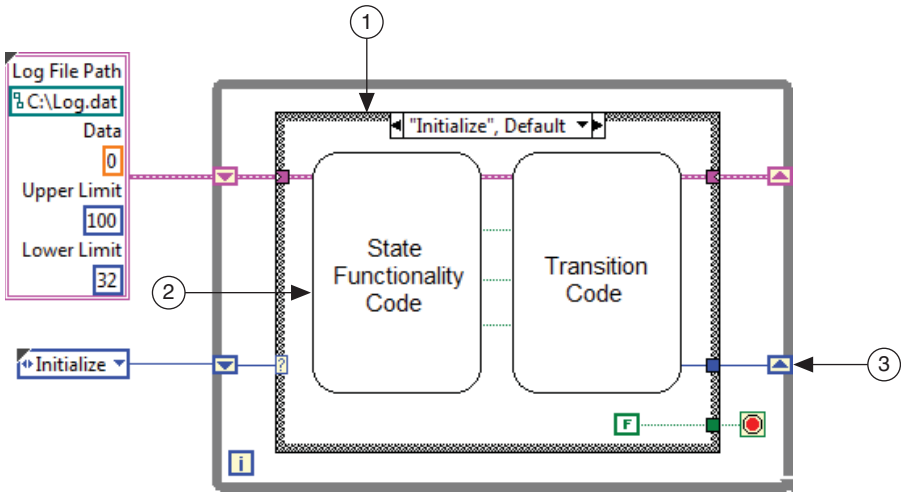
Use this design pattern for VIs that are easily divided into several simpler tasks, such as VIs that act as a user interface.

The state machine pattern has a start up and shut down phase. The main application runs different code each time the loop executes, depending upon some condition.



State Machine Framework

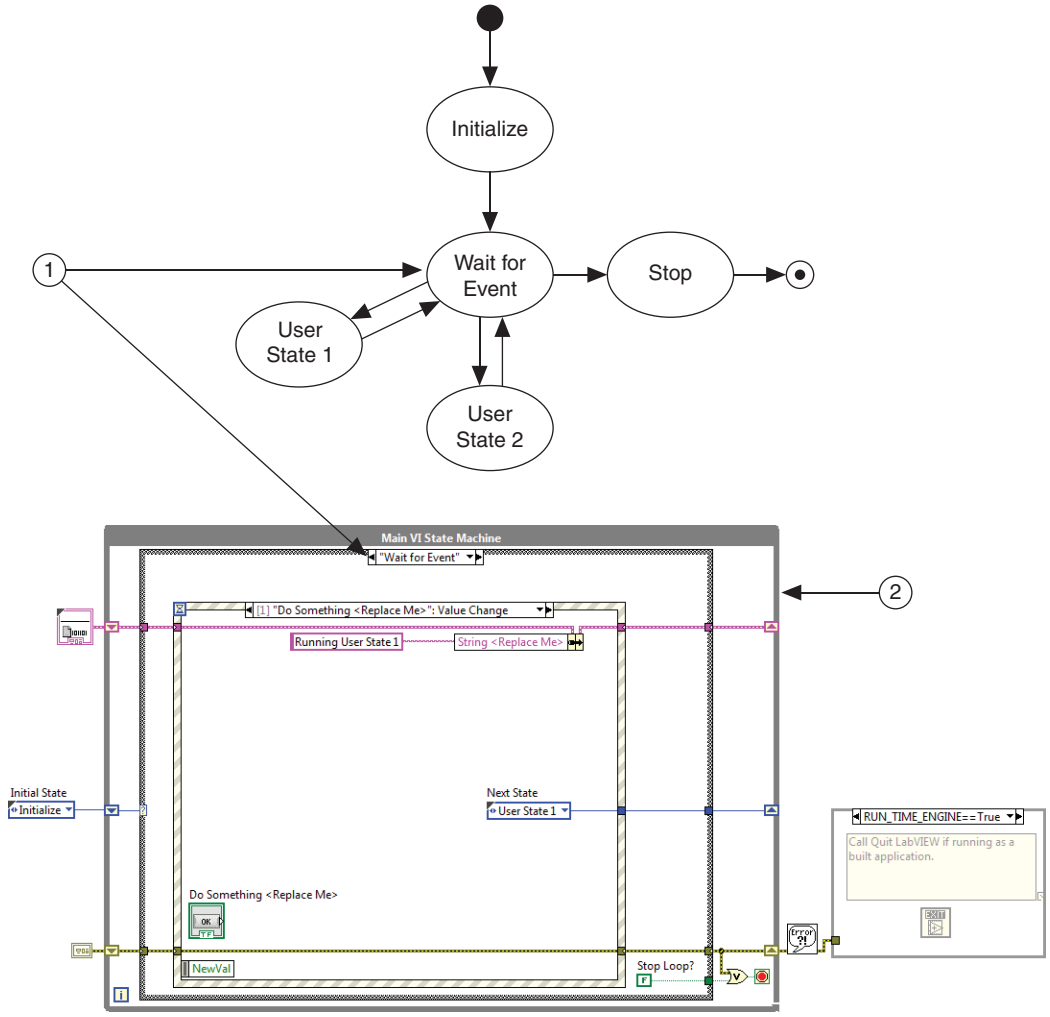
The state machine framework consists of a While Loop, a Case structure, and a shift register.



1 Each state of the state machine is a separate case in the Case structure.
2 Place VIs and other code that the state should execute within the appropriate case.
3 A shift register stores the state that should execute upon the next iteration of the loop.

Event-Based State Machine

The event-based state machine combines the powerful user interaction of the user interface event handler with the transition flexibility of the state machine.



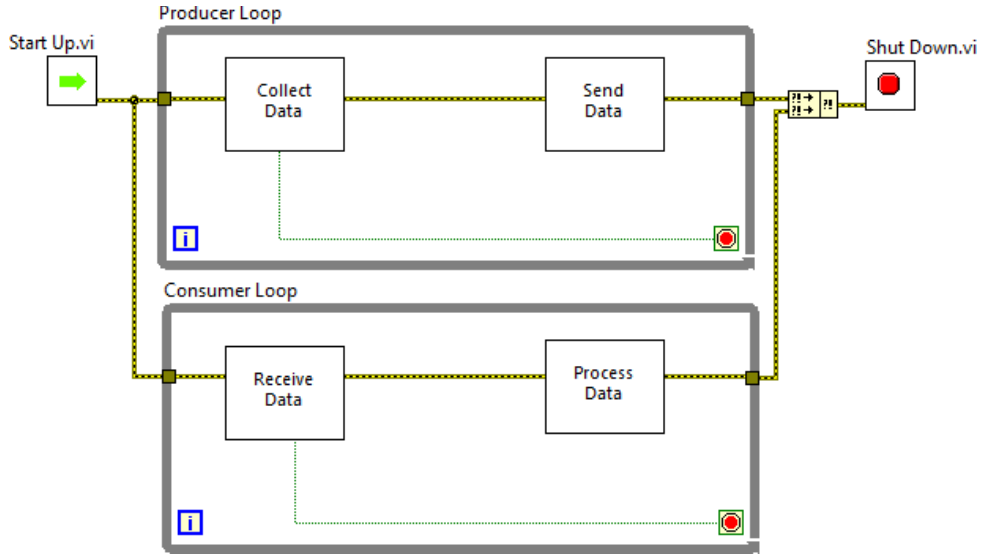
1 The Wait for Event state contains an Event structure that waits for front panel changes. The Wait for Event state is the only one that recognizes user input.
2 Only one state executes at a time, and the single While Loop means all tasks execute at a single rate.

C. Multiple Loop Design Patterns

Objective: Use the producer/consumer (events) design pattern to create multiple loop applications.

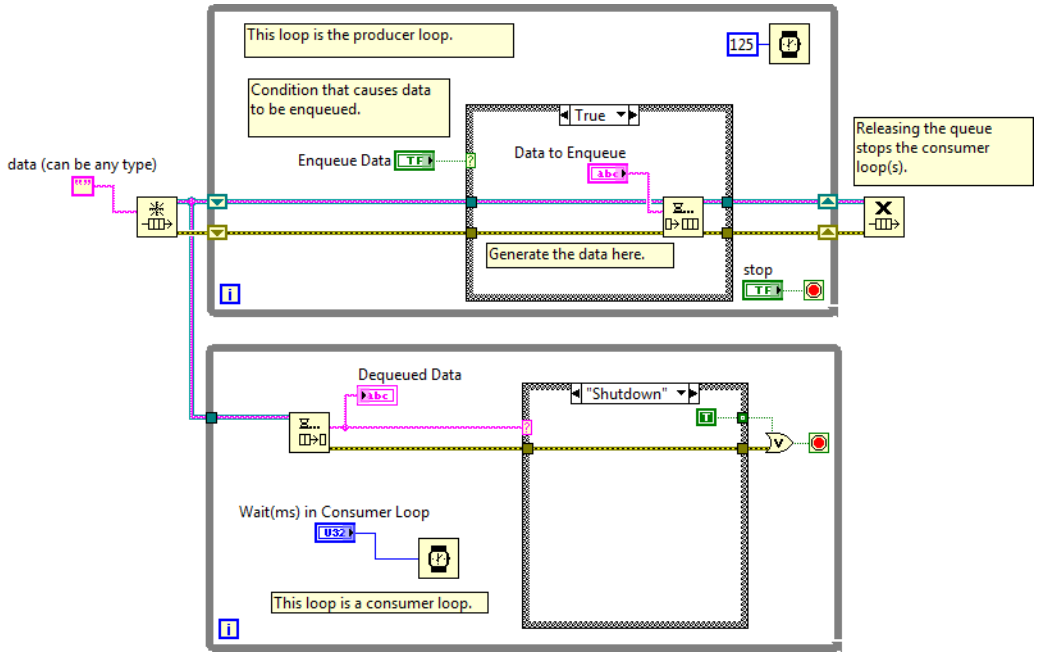
Producer/Consumer Design Pattern

The producer/consumer design pattern separates tasks that produce and consume data at different rates. Use the producer/consumer design pattern to acquire multiple sets of data that must be processed in order. Use queues to communicate and buffer data between loops.



Demonstration: Producer/Consumer (Data) Design Pattern

Navigate to `<Exercises>\Demonstrations\Producer Consumer - Data\` and explore Producer Consumer Data.vi to learn more about how data is passed from the producer loop to the consumer loop.



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

Goal

As a group, explore the Producer Consumer template.

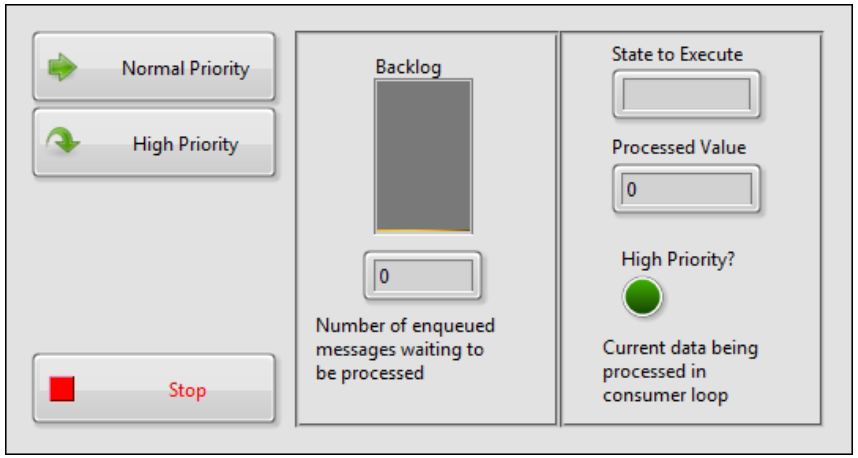
Scenario

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

D. Functional Global Variable Design Pattern

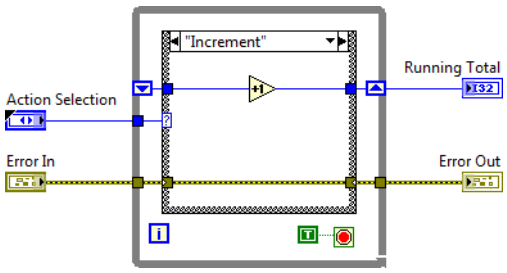
Objective: Describe and use the functional global variable design pattern.

Functional global variable A non-reentrant VI that uses uninitialized shift registers to hold global data.

Multimedia: Functional Global Variables

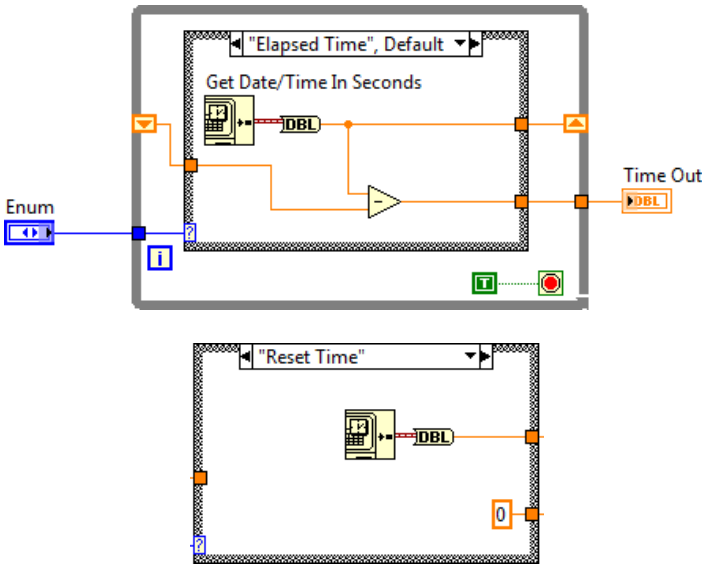
This module covers the basics of functional global variables, including the framework and advantages.

Complete the multimedia module, *Functional Global Variables*, available in the `<Exercises>\LabVIEW Core 2\Multimedia\` folder.



Timer FGVs

One powerful application of functional global variables is to perform timing in your VI. For example, you can create a functional global variable that measures the elapsed time between each instance a VI is called.



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

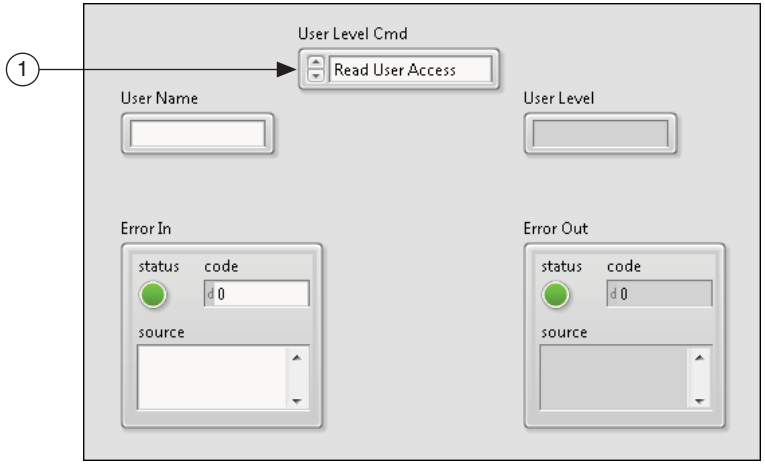
The following table describes the different actions you need to handle so you can implement user access control. In this exercise, you create a custom control to handle these items.

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

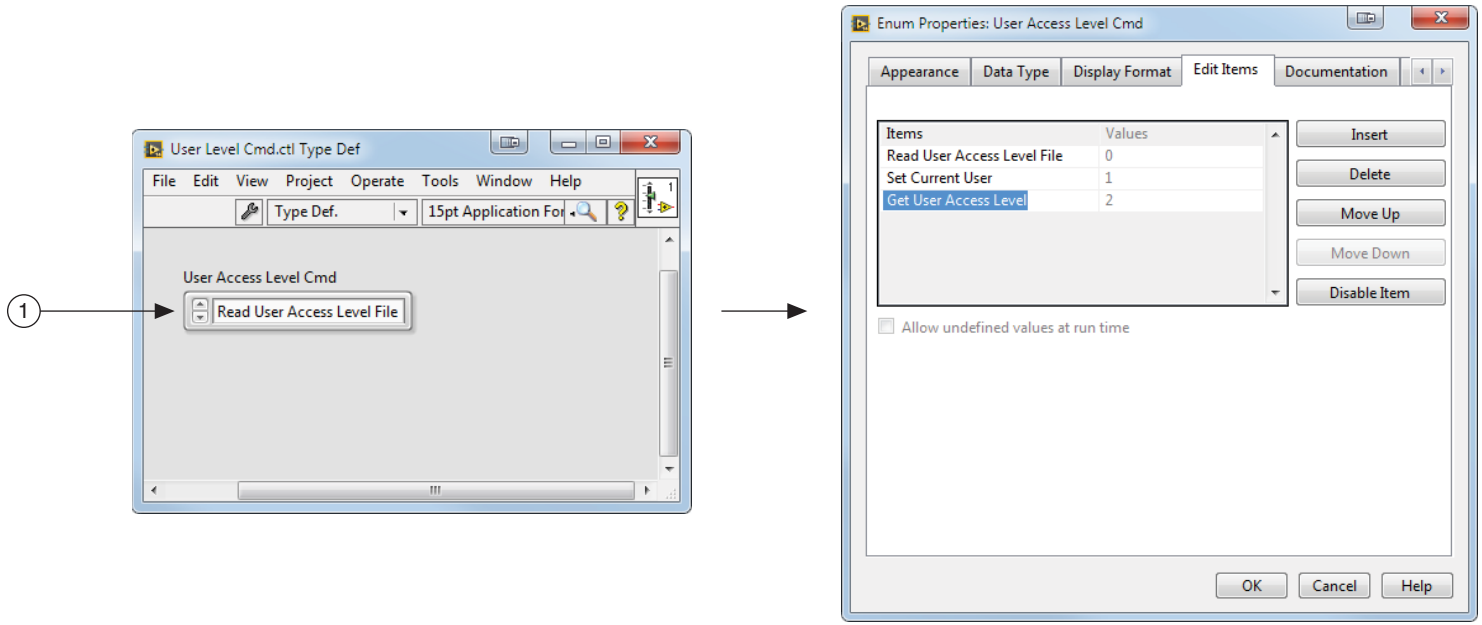


1 Enum (Silver)—Place an Enum (Silver) control on the front panel. Right-click the enum control and select **Make Type Def**.

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.

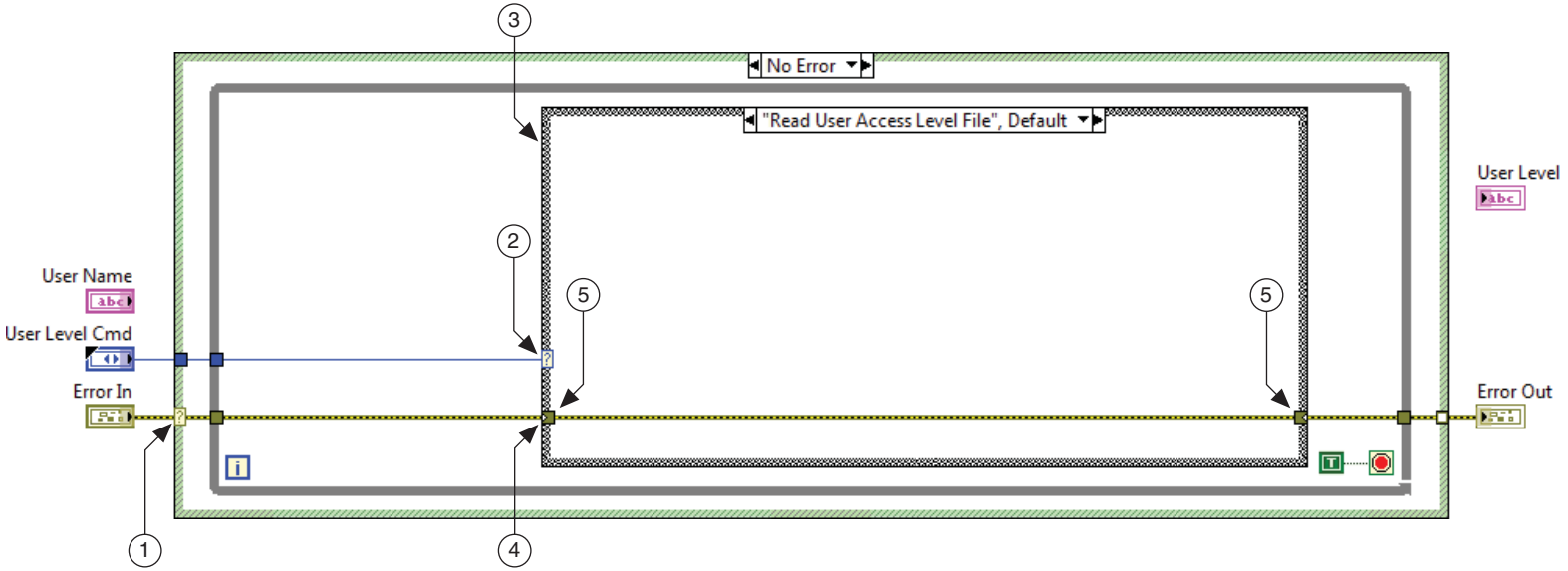
Figure 3-4. Assign the User Level Cmd Control to a Connector Pane Terminal

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

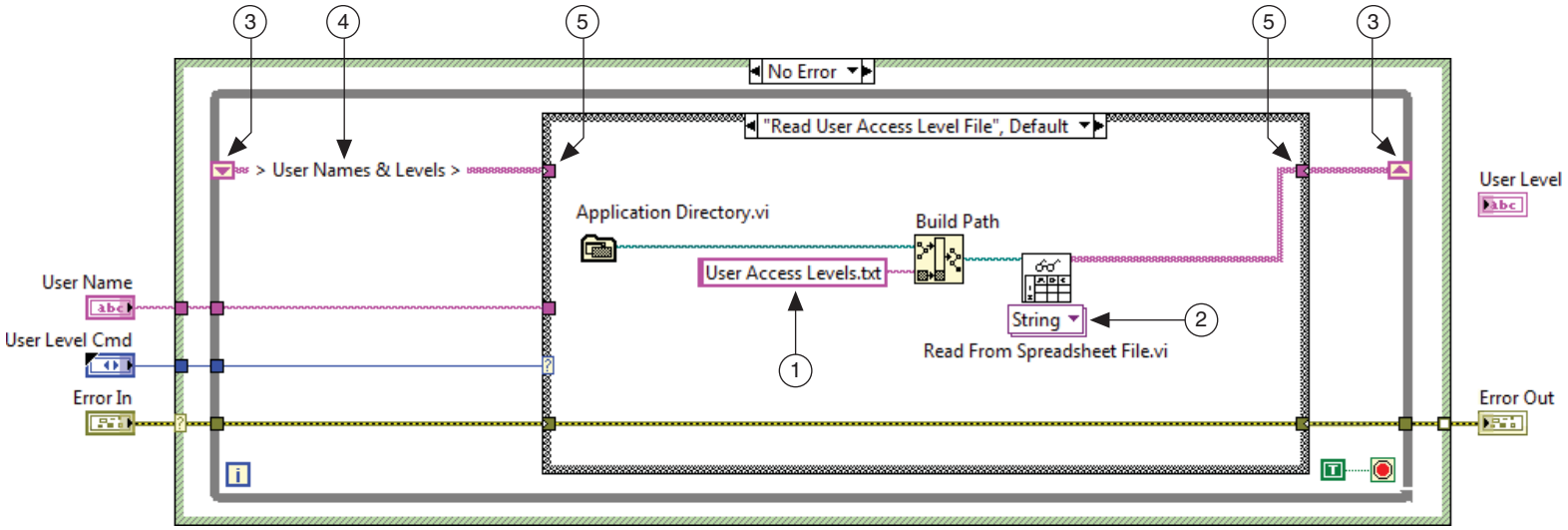


1 Case selector—Wire the **Error In** cluster to the case selector of the outer Case structure to set the error and no error cases.
2 Case selector—Wire **User Level Cmd** to the case selector.
3 Case structure—Right-click the Case structure and select **Add Case for Every Value** from the shortcut menu 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.

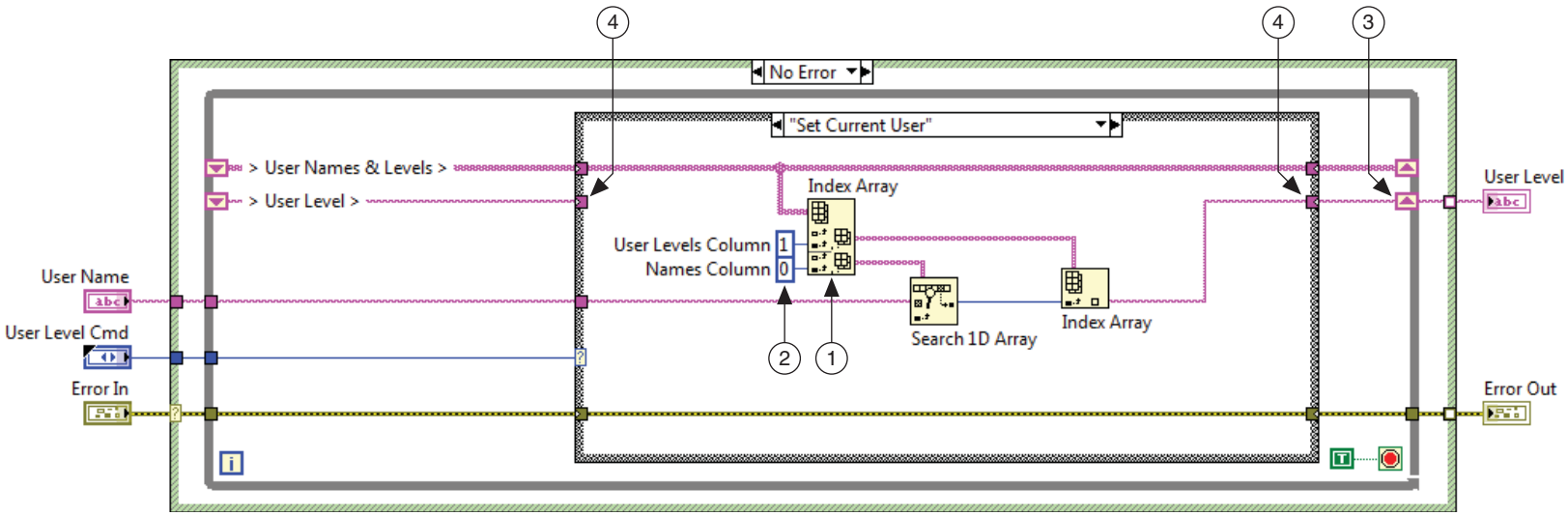
Figure 3-6. Configuring the Read User Access Level File Case



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.

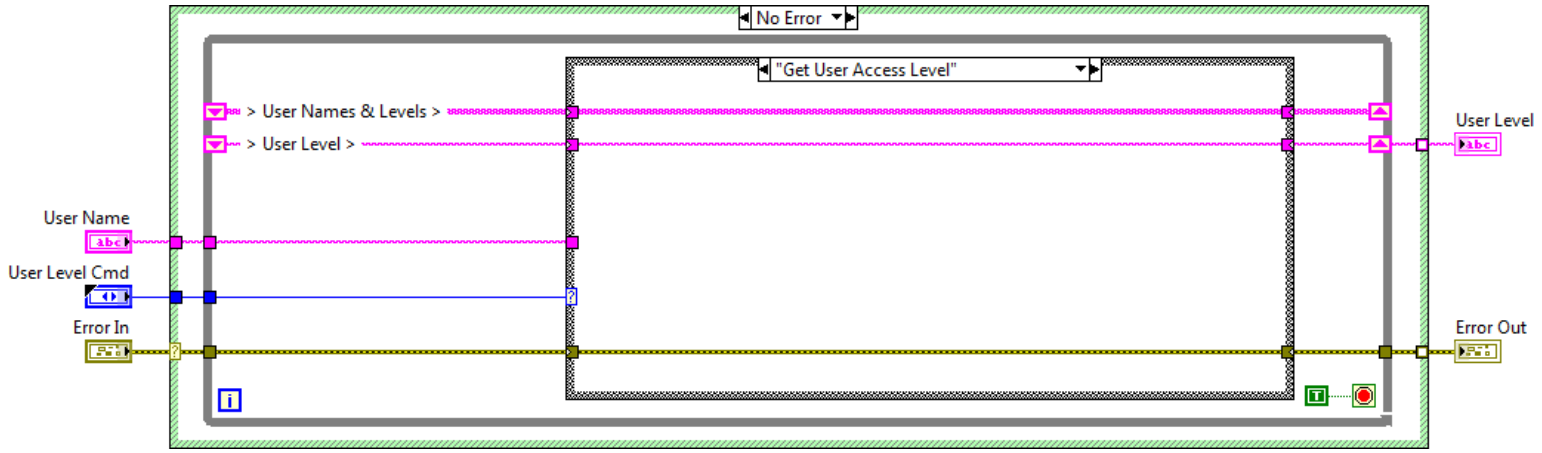
Figure 3-7. Configuring the Set Current User Case



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.

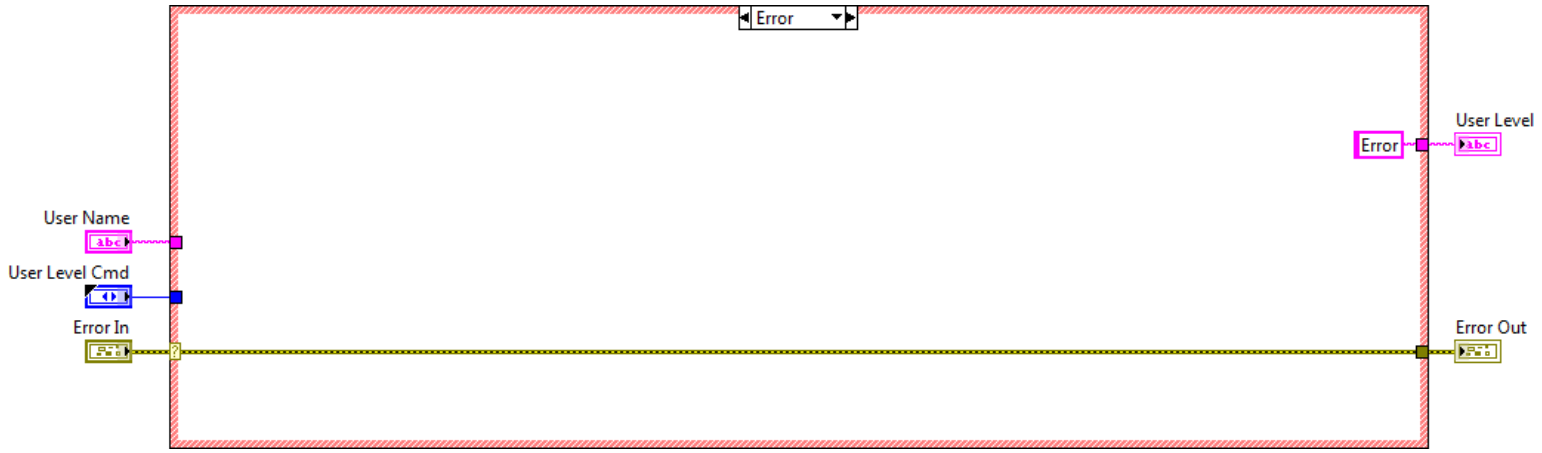
11. Leave the Get User Access Level case as shown in Figure 3-8.

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

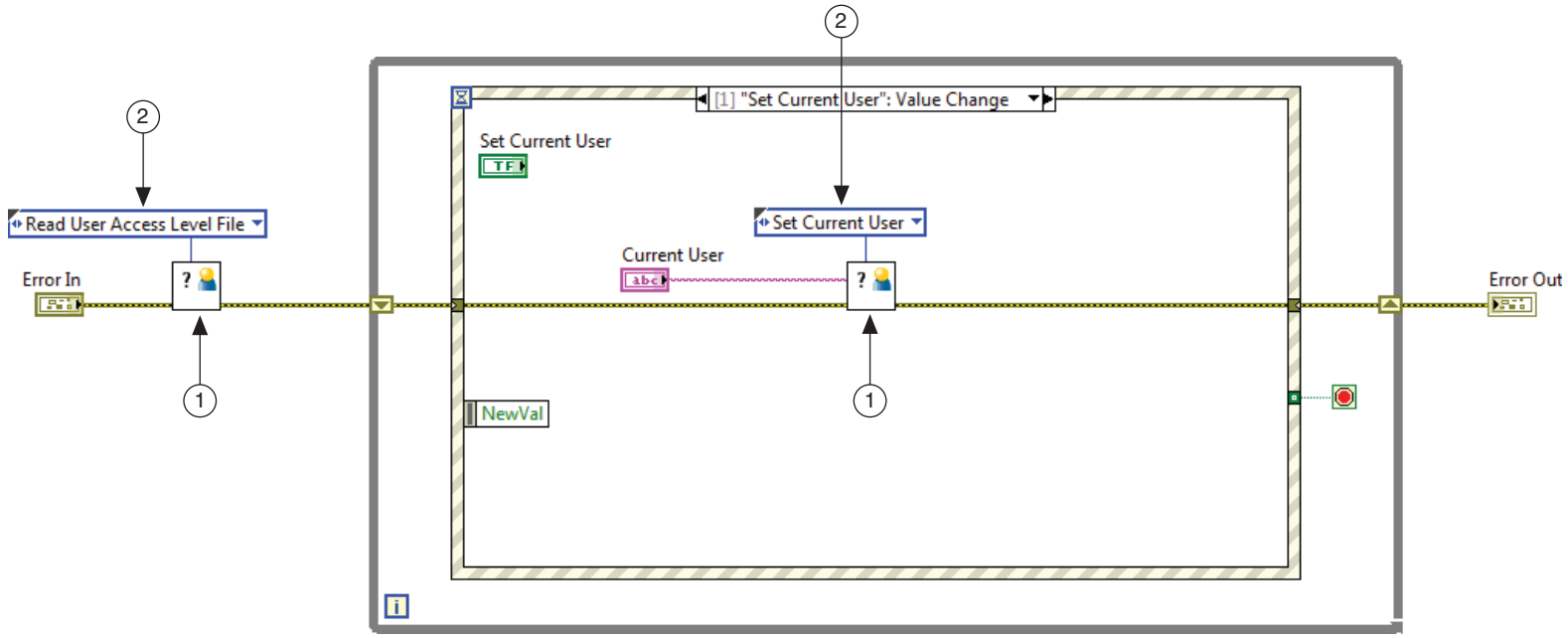
1. From the project, open `User Access Levels.txt` and review the contents of the file. This file contains the names of authorized users and their access levels. You can add additional user names and levels if you want.

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.

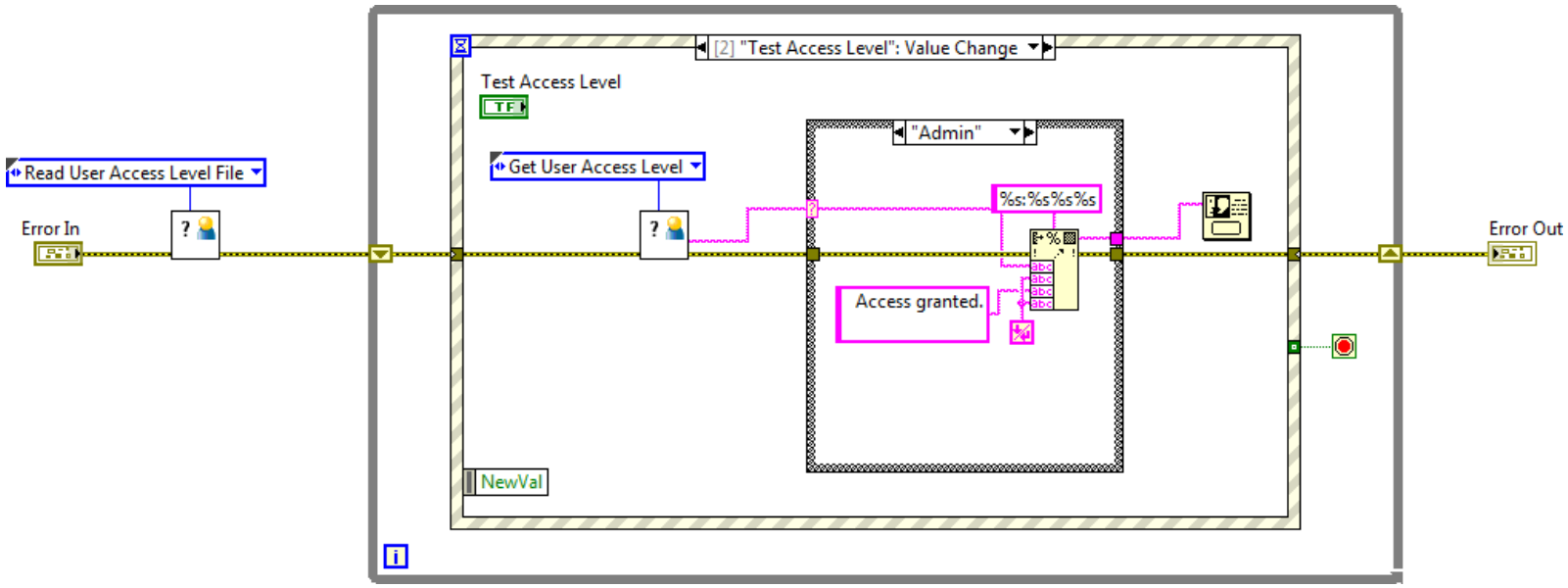
Figure 3-10. Completing the "Set Current User": Value Change Event



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

Activity 3-1: Identifying Design Patterns

Based on the scenario, identify the design pattern that you would use to implement that application. The first scenario has been completed for you.

Scenario	Design Pattern
Executing a pre-defined sequence of tests on a device	State Machine
An application that reads, modifies, or writes user data	
Acquire one temperature sample from a thermocouple	
Continuously acquire pressure data	
Testing a device and logging results continuously in parallel	
Executing tests in the order defined by the user	

Activity 3-2: Design Patterns Job Aid

Use the table below to determine the best uses for the design patterns described in this lesson.

Design Pattern	Use	Advantage	Disadvantage
Simple	Standard subVIs Calculations/algorithms; modular processing LabVIEW equivalent of a subroutine in other programming languages	Allows for modular applications	Not suitable for user-interface design or top-level VIs
General	Standard control flow Good for quick prototypes or simple, straight-forward applications that will not grow in complexity	Distinct initialize, run, and stop phases	Unable to return to a previous phase

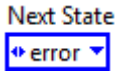
Design Pattern	Use	Advantage	Disadvantage
State Machine (Polling)	Controls the functionality of a VI by creating a system sequence	Controls sequences Code maintenance is easy because new states can be added quickly For simple applications, do not have to manage both event and state machine diagrams	Polling-based UI is not scalable as application grows This design pattern is not inherently parallel.
State Machine (Events-based)	Controls the functionality of a VI by creating a system sequence	Controls sequences Code maintenance is easy because new states can be added quickly Use of event structure more efficient than polling controls	This design pattern is not inherently parallel.
Producer/Consumer (Data)	Processes or analyzes data in parallel with other data processing or analysis	Buffered communication between application processes	Does not provide loop synchronization Limited to one data type, although data can be clustered
Producer/Consumer (Events)	Responds to user interface with processor-intensive applications	Separates the user interface from processor intensive code	Does not integrate non-user interface events well
Functional Global Variable	Use as a subVI that needs to hold global data and perform actions on that data	Holds data as long as VI is in memory Executes operations based on input selection Good way to protect critical sections of code to eliminate race conditions	Not suitable for reentrant VIs Problematic when duplicating or scaling global data with multiple copies and performing actions on the copies

E. Error Handlers

Objective: Use error handlers in design patterns to manage code execution when an error occurs.

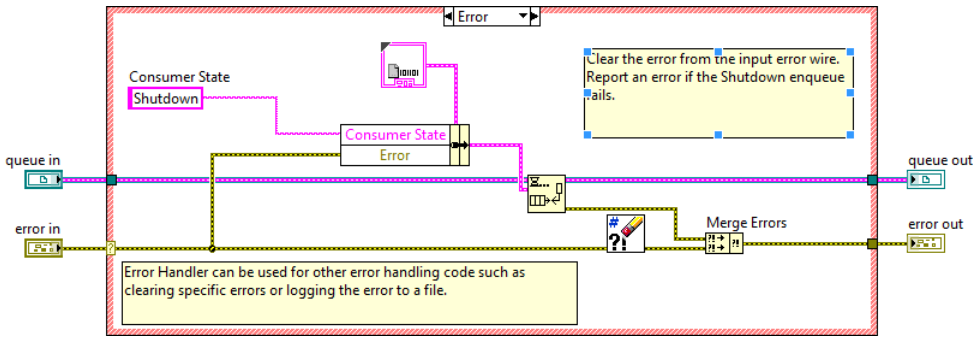
Examples of Error Handlers

Error handler VI or code that changes normal flow or program execution when an error occurs.

Simple Error Handler VI		Displays a dialog box with error information when an error occurs.
General Error Handler VI		Same functionality as the Simple Error Handler VI, except this VI allows you to define custom errors as well.
State machine error handler		Transitions the state machine to an error or shutdown state when an error occurs.

Demonstration: Producer Consumer Error Handler VI

Navigate to `<Exercises>\Producer Consumer – Error\` and explore how errors are handled in `Error Handler.vi`.



F. Generating Error Codes and Messages

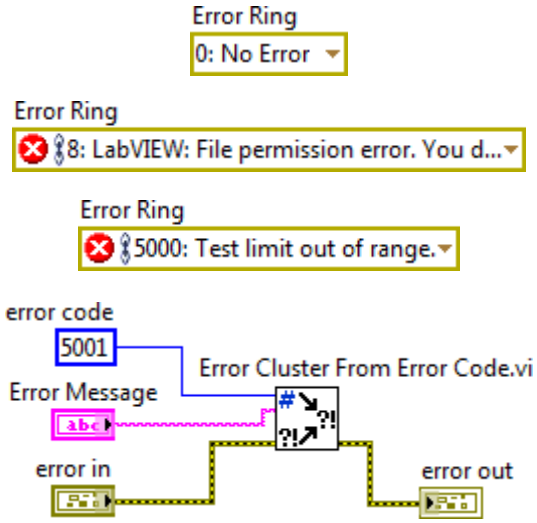
Objective: Use error codes and messages in design patterns.

Error Reporting Options

Use existing error reporting mechanisms to report error conditions detected with your code, such as the following:

- Invalid inputs to subVIs
- File and resource errors
- LabVIEW-generated messages

You can use either pre-defined errors or user-defined errors. You may want to override LabVIEW-generated error messages to make your code more usable. Use use error rings to report pre-defined LabVIEW errors or define your own custom errors. You can also use the Error Cluster From Error Code VI to generate a custom error.



Demonstration: Use the Error Ring to Generate Errors

You can use an error ring to generate pre-defined or user-defined error codes and messages.

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

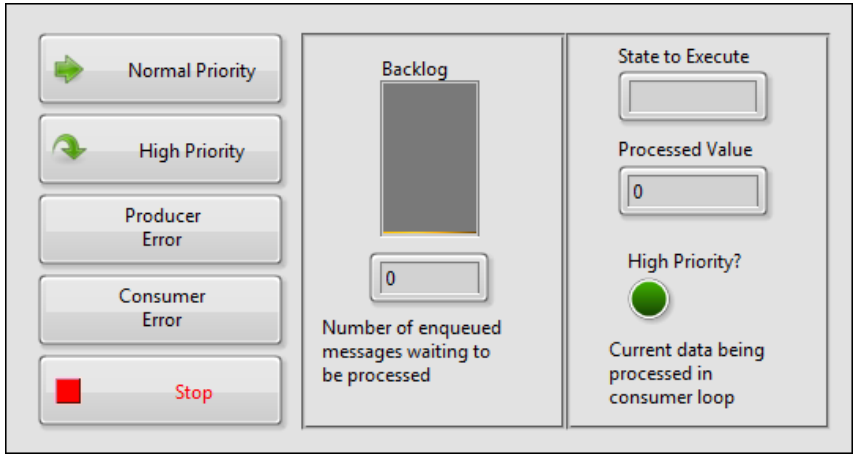
Add buttons to inject simulated errors in the producer loop and the consumer loop. Use Error Ring constants to produce simulated errors. Update the producer loop to handle an error case.

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.

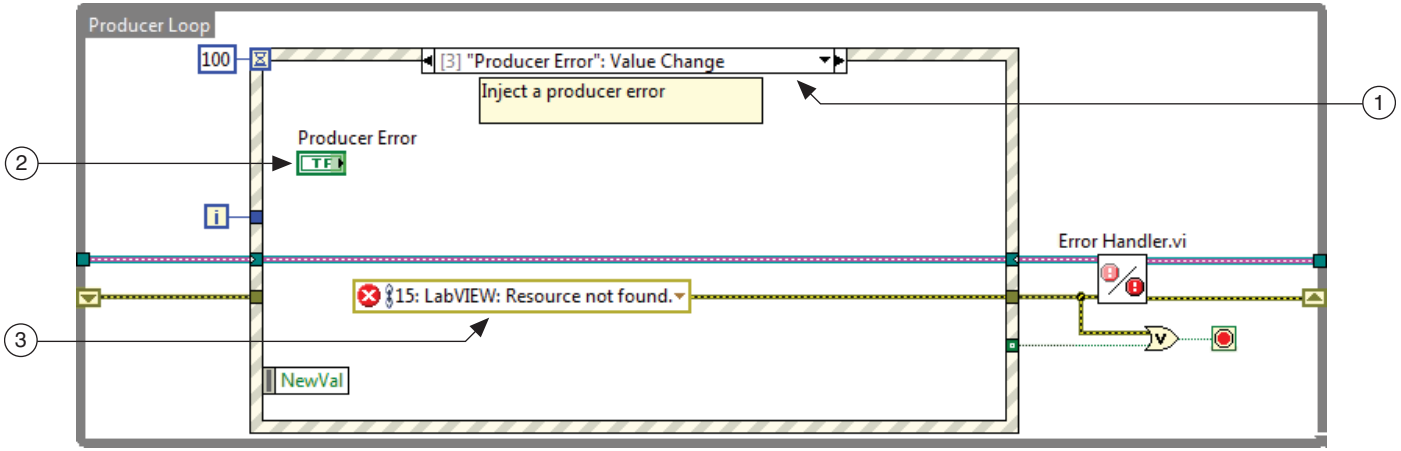
   This VI is similar to the one you used in Exercise 3-1. The Producer Error and Consumer Error buttons are provided on the front panel, as shown in Figure 3-12. You modify the block diagram to enable the buttons and test error handling in this VI.

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.

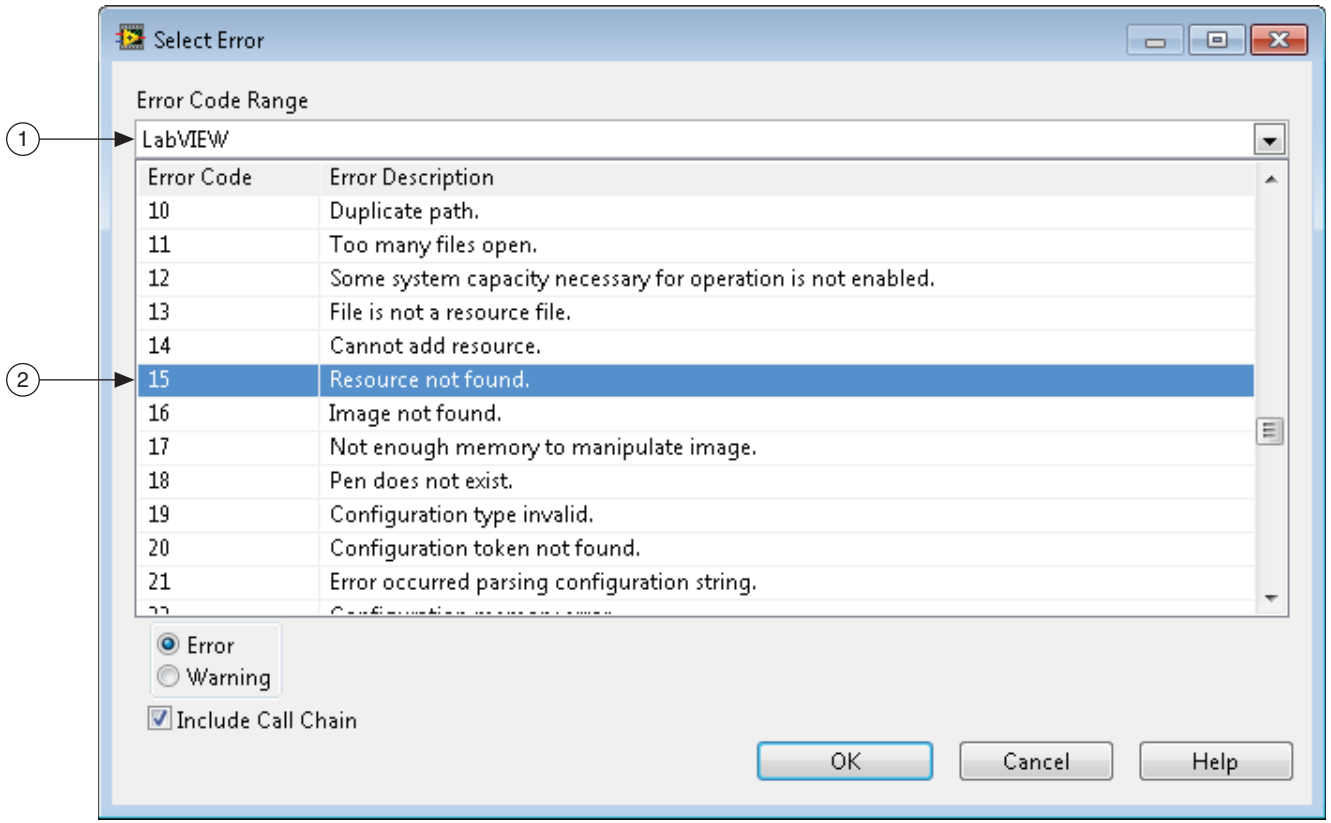
Figure 3-13. Producer Loop "Producer Error": Value Change Event



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.

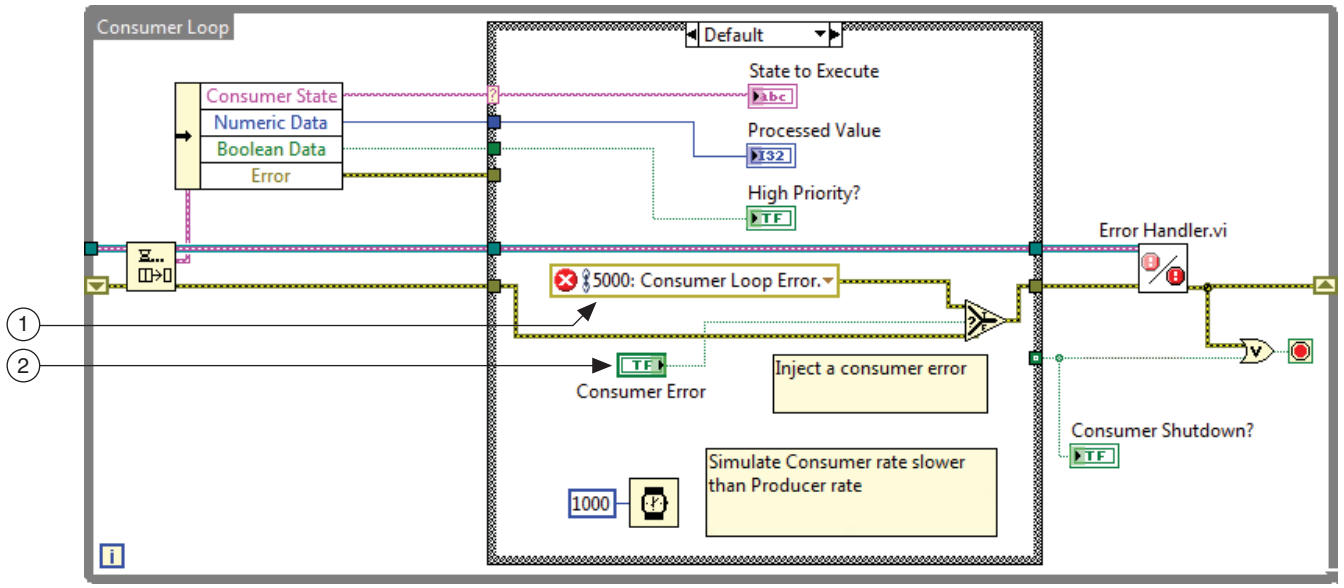
Figure 3-14. Select Error Dialog Box



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.

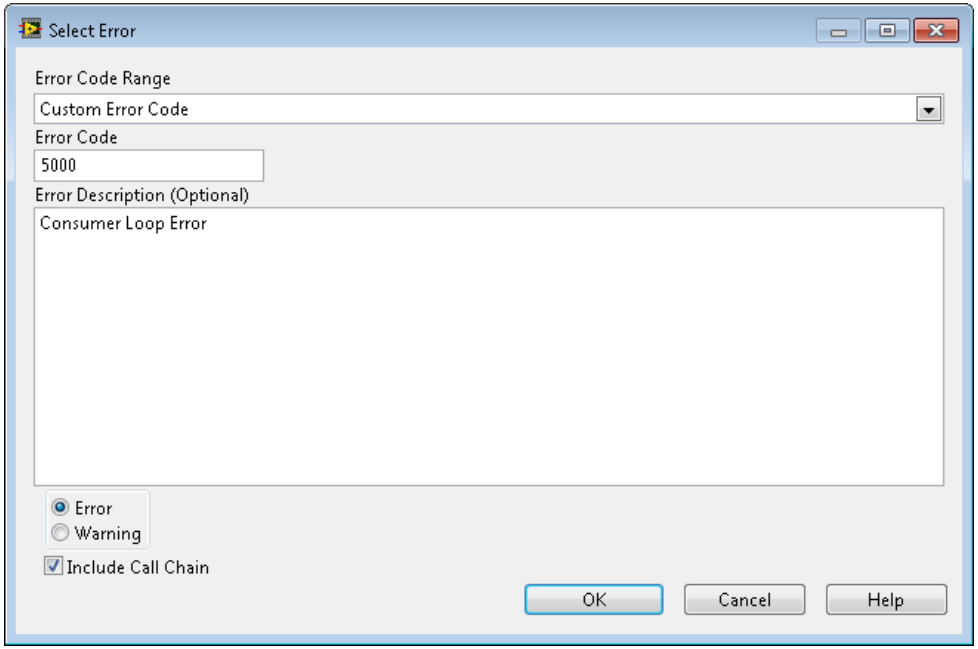
Figure 3-15. Consumer Loop Custom Error Code



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

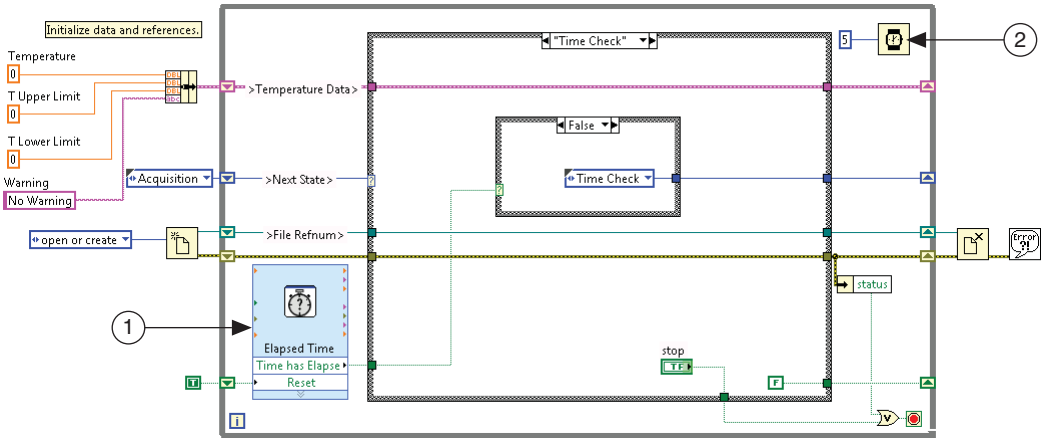
G. Timing a Design Pattern

Objective: Use timing functions in design patterns.

Execution and Software Control Timing

Execution timing Code that uses timing functions to give the processor time to complete other tasks.

Software control timing Code that involves timing a real-world operation to perform within a set time period or controls the frequency at which a loop executes.

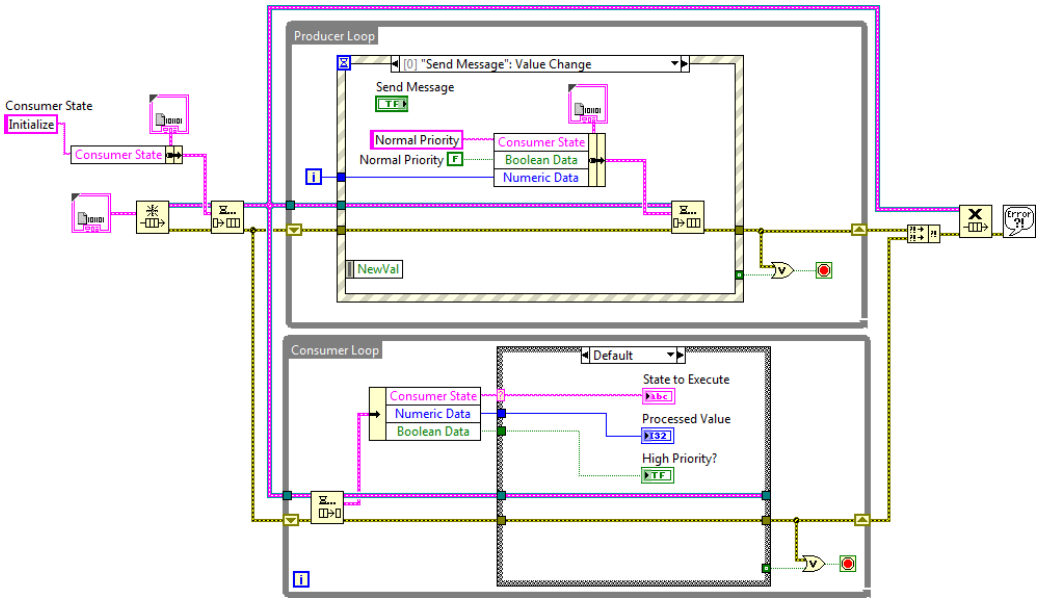


1 Software control timing
2 Execution timing

Execution Timing

Explicit timing can use a function that specifically allows the processor time to complete other tasks, such as the Wait Until Next ms Multiple function. Execution timing can also be based on events. When the timing is based on event, the design pattern waits for some action to occur before continuing and allows the processor to complete other tasks while it waits.

Use explicit timing for polling-based design patterns such as the producer/consumer design pattern (data) or the polling-based state machine.



Execution Timing Functions

Function Name	Icon
Wait (ms)	
Wait Until Next ms Multiple	

Software Control Timing

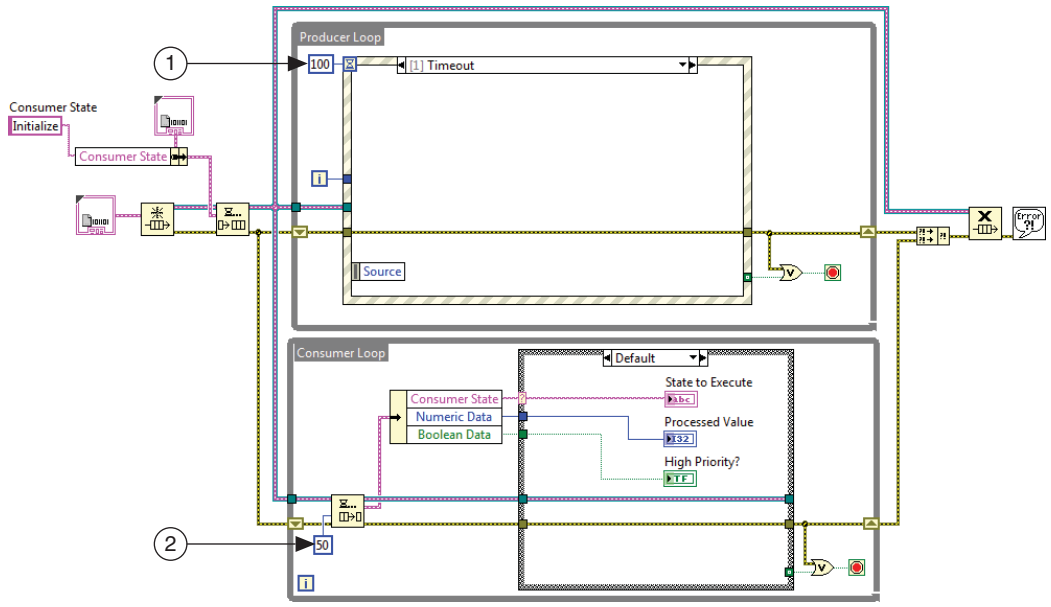
Software control timing keeps the application executing while monitoring a real-time clock.

Two approaches to software control timing:

- Functional Timeout
- Timestamps

Functional Timeout

Functional timeout allows code to be executed at regular intervals in both the consumer and producer loops, even if no other events occur or the queue is empty.



1 Wiring a millisecond value to the timeout terminal of an Event structure wakes the Event structure and executes the code in the Timeout case. here, the Producer loop executes every 100 ms even if no other events occur.
2 The Consumer loop executes every 50 ms even if the queue is empty.

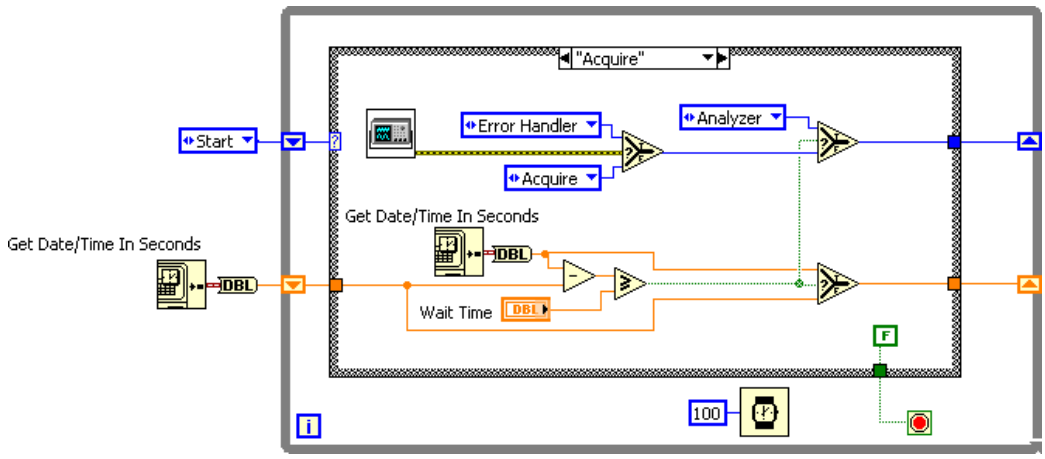
Obtaining Timestamps

If you want to measure the amount of time that passes between two iterations of a loop, use the following functions to calculate relative time durations:

Function	Connector Pane
Tick Count (ms)	millisecond timer value
High Resolution Relative Seconds	relative seconds
Get Date/Time in Seconds	current time

Get Date/Time in Seconds

Use the Get Date/Time In Seconds function for working with elapsed times. The Get Date/Time In Seconds function returns a timestamp of the current time.



Activity 3-3: Identifying Timing Options

Determine whether each scenario indicates a need for execution timing or software control timing.

Scenario	Timing Approach
Monitor the time that has passed between iterations of a loop to determine whether or not a given amount of time has elapsed.	Software control timing
Acquire 1 sample/second for 1 minute.	
Other applications on your system slow down as a result of running a VI.	

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.

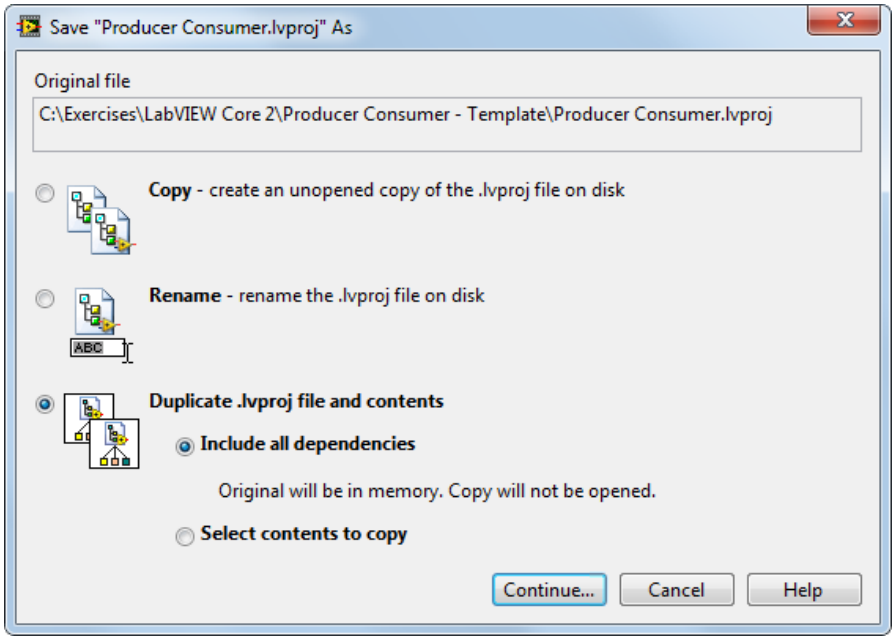
Implementation

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**.

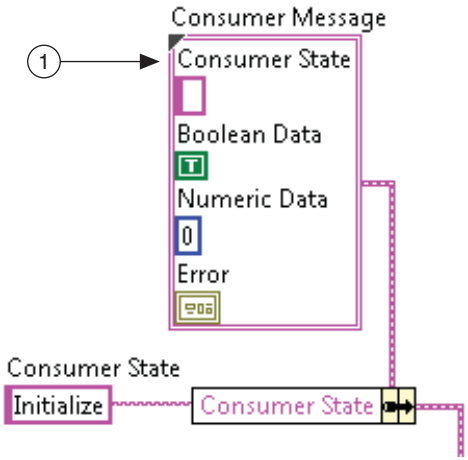
   ☐ Rename the VI as `Histogram Main.vi` and click **OK**.

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.

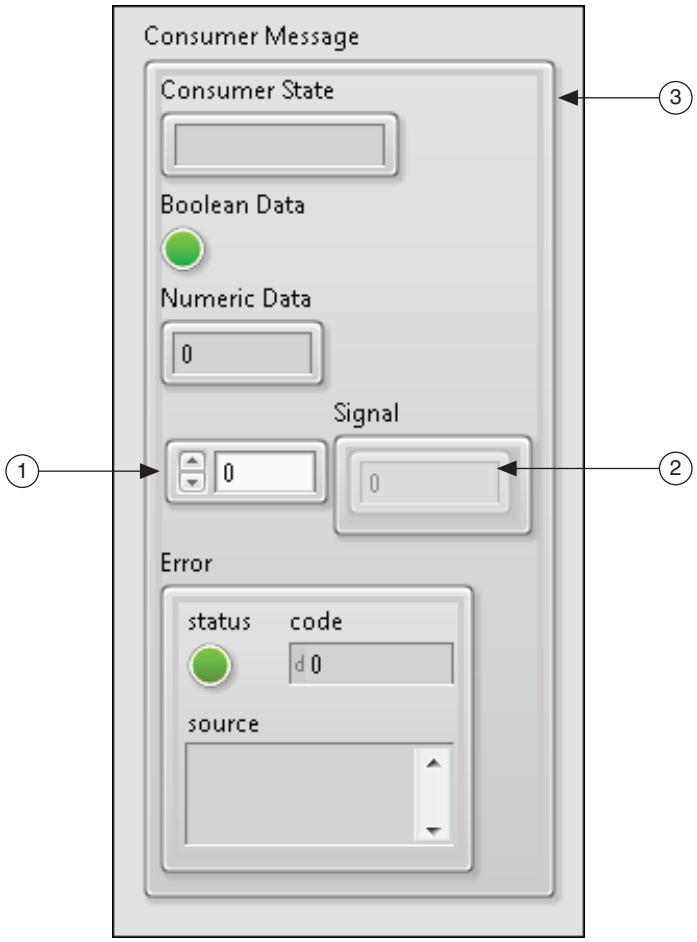
Figure 3-18. Consumer Message Type Definition Terminal on Histogram Main VI Block Diagram



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

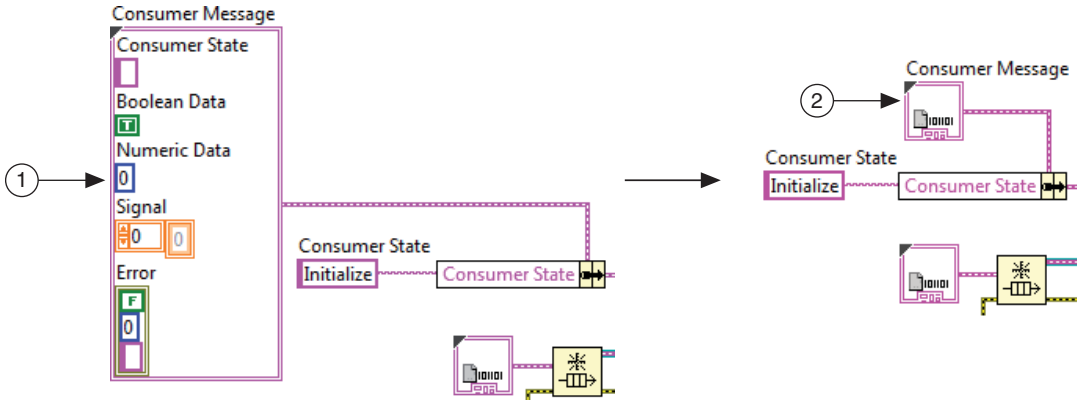


1 Array—Add an array to the type definition so it can handle waveform data. Rename the Array `Signal`.
2 Numeric Indicator—Add a numeric indicator to the array.
3 Right-click the cluster border and select **Reorder Controls In Cluster** and arrange them so that the Signal control is directly below the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

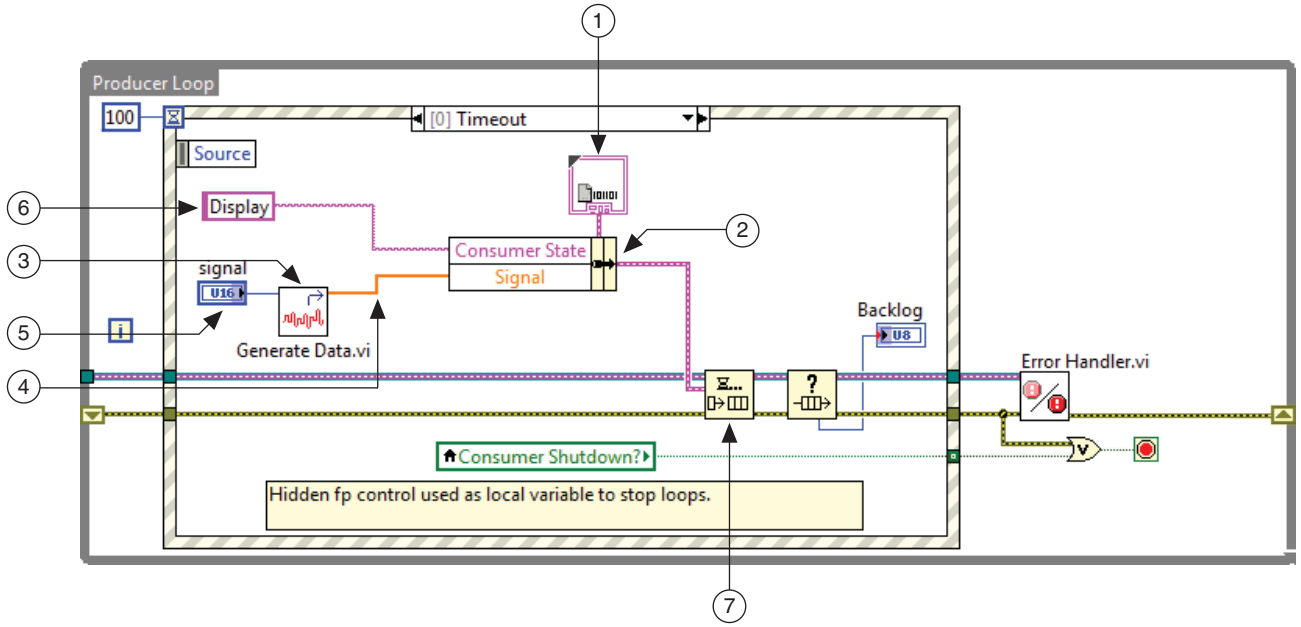
Figure 3-20. Viewing a Type Definition as an Icon



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.

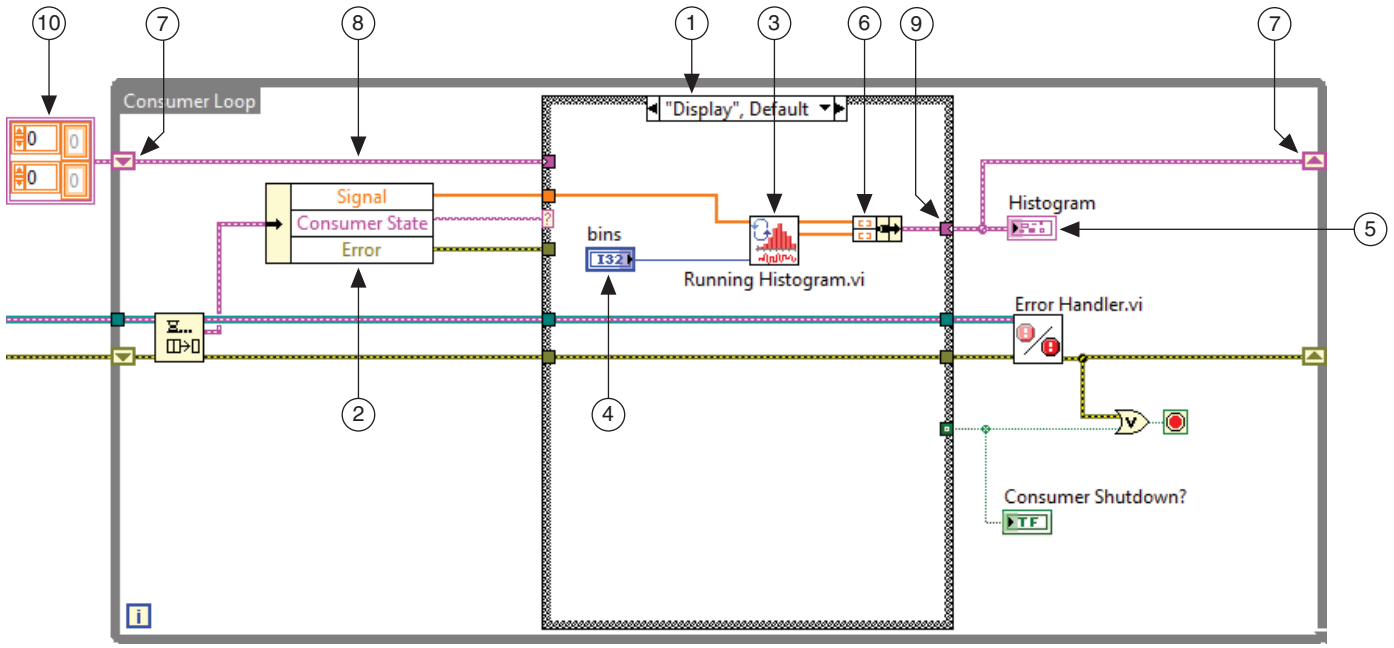
Figure 3-21. Updating the Producer Loop Timeout Event



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.

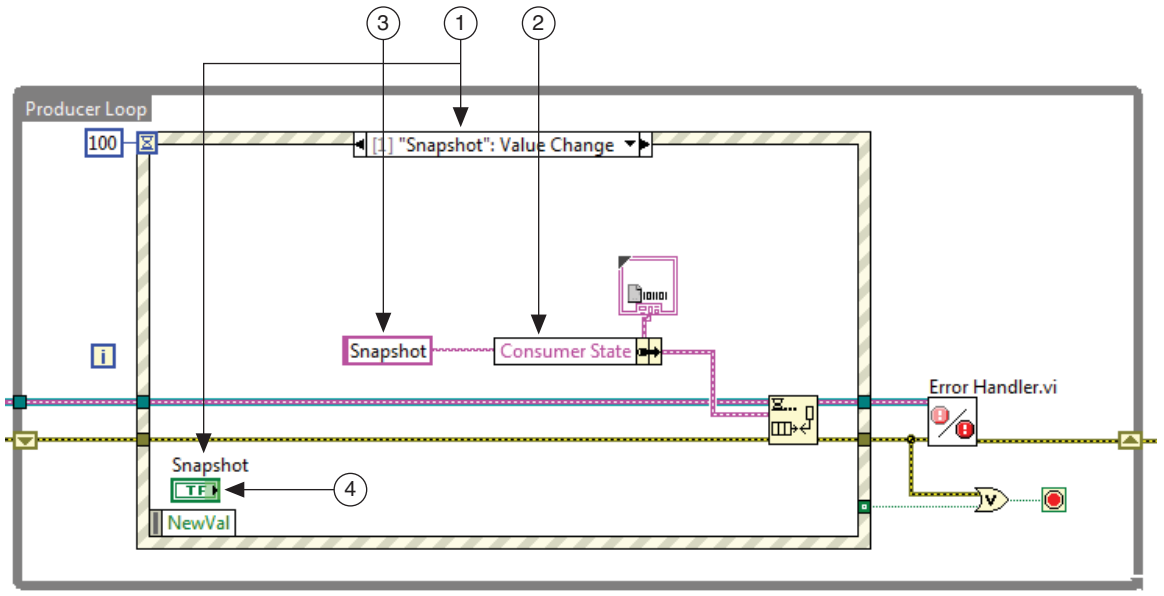
Figure 3-22. Updating the Consumer Loop Display Case



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.

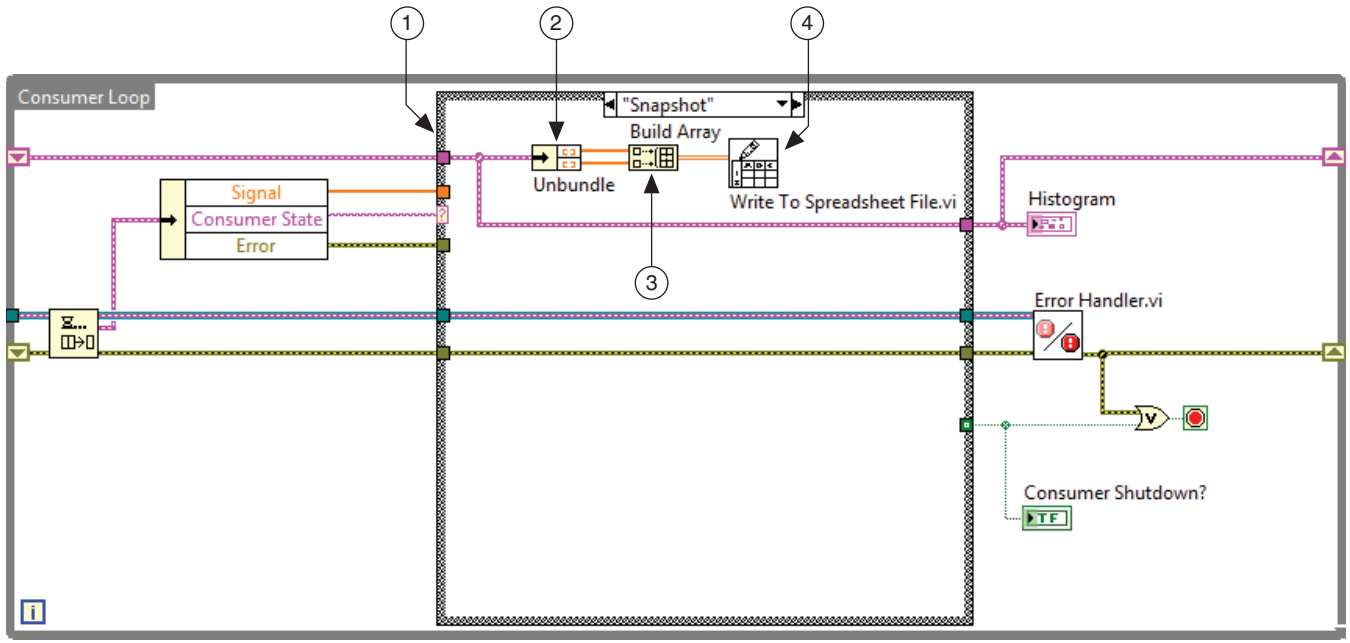
Figure 3-23. Updating the Producer Loop “Snapshot”: Value Change Event



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

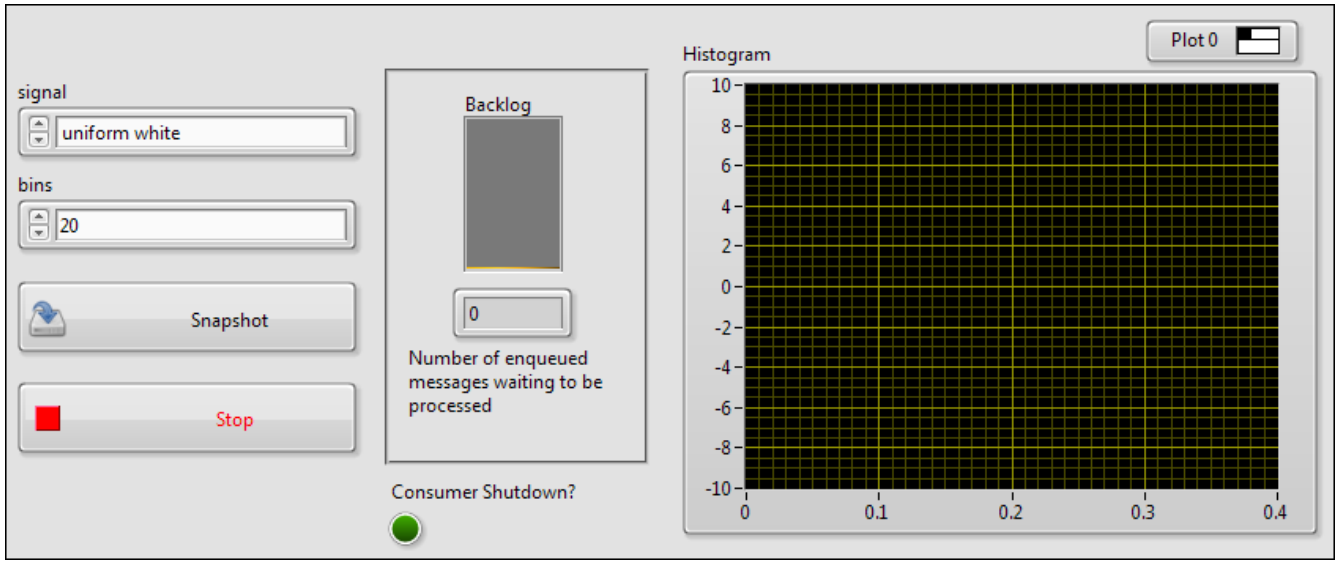


1 Duplicate the Initialize case—Right-click the case structure and select **Duplicate Case**. Rename the duplicate case `Snapshot`.
2 Unbundle function—After you wire the input, the Unbundle function contains two 1D arrays.
3 Build Array function—Wire both **1D Array** outputs to the Build Array function.
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

Additional Resources

Learn More About	LabVIEW Help Topic
Create custom error codes	*Defining Custom Error Codes*
Creating state machine applications	*Creating VIs from Templates and Sample Projects*
Functional Global Variables	*Suggestions for Using Execution Systems and Priorities*
Timing a design pattern	*Selecting a Timing Source for a Timed Structure*

Activity 3-4: Lesson Review

1. Which of the following are reasons for using a multiple loop design pattern?
   a. Execute multiple tasks concurrently
   b. Execute different states in a state machine
   c. Execute tasks at different rates
   d. Execute start up code, main loop, and shutdown code

2. Which of the following are examples of error handling code?
   a. Displays a dialog box used to correct a broken VI
   b. Generates a user-defined error code
   c. Displays a dialog box when an error occurs
   d. Transitions a state machine to a shutdown state when an error occurs

Activity 3-4: Lesson Review - Answers

1. Which of the following are reasons for using a multiple loop design pattern?
   - **a. Execute multiple tasks concurrently**
   - b. Execute different states in a state machine
   - **c. Execute tasks at different rates**
   - d. Execute start up code, main loop, and shutdown code

2. Which of the following are examples of error handling code?
   - a. Displays a dialog box used to correct a broken VI
   - b. Generates a user-defined error code
   - **c. Displays a dialog box when an error occurs**
   - **d. Transitions a state machine to a shutdown state when an error occurs**

4 Controlling the User Interface

In this lesson you will learn to use property nodes, invoke nodes, and control references to programmatically control front panel objects.

Topics

+ VI Server Architecture
+ Property Nodes
+ Invoke Nodes
+ Control References

Exercises

Exercise 4-1 Display Temperature and Limits
Exercise 4-2 Customize the VI Window
Exercise 4-3 Create SubVIs for Common Operations

A. VI Server Architecture

Objective: Describe the purpose of the VI Server and the class hierarchy of properties and methods.

VI Server Purpose and Use

The VI Server provides programmatic access to LabVIEW and LabVIEW applications. The VI server performs many functions and in this lesson concentrates on how to use the VI Server to control front panel objects and edit the properties of a VI and LabVIEW.

You can use the VI Server to perform the following actions:

- Programmatically control front panel objects and VIs
- Dynamically load and call VIs
- Run VIs on a computer or remotely across a network
- Programmatically access the LabVIEW environment and editor (Scripting)

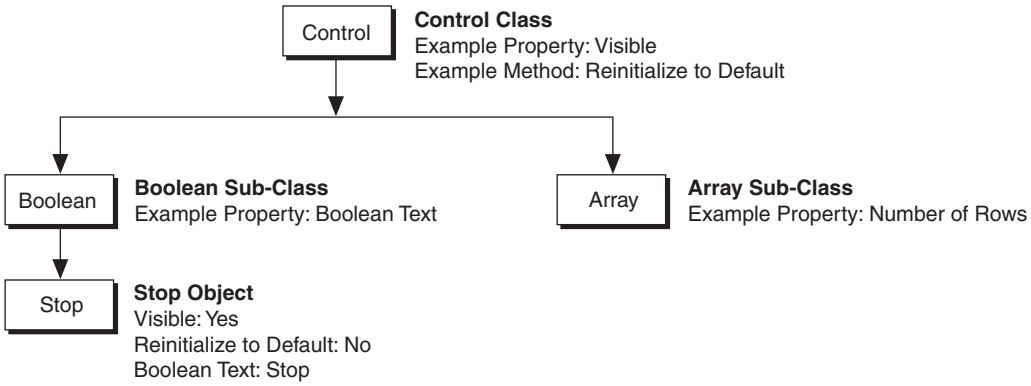
Properties and Methods

Properties Single-valued attributes of the object: read/write, read only, write only

Methods Functions that operate on the object

VI Server—Class Hierarchy

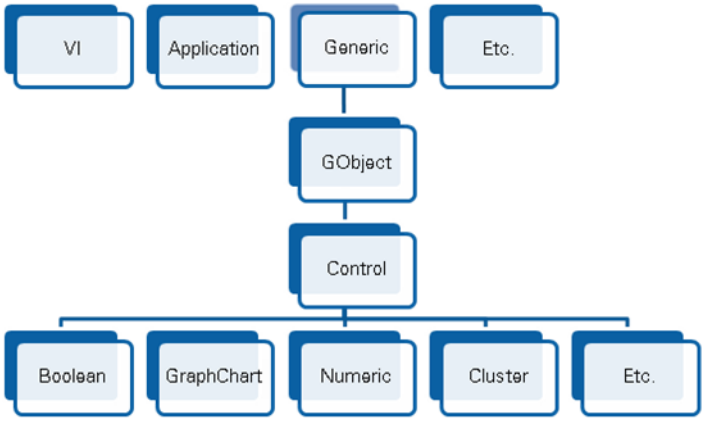
LabVIEW front panel objects inherit properties and methods from a class. When you create a Stop control, it is an object of the Boolean class and has properties and methods associated with that class.

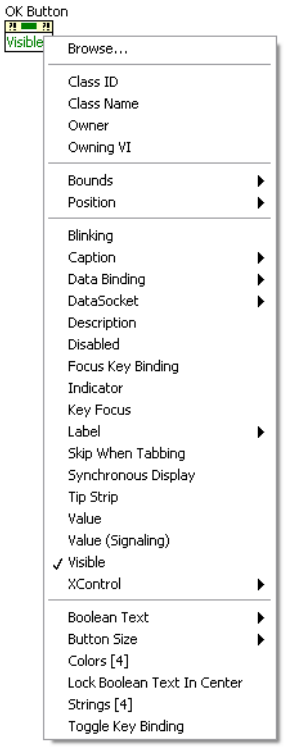


Class Hierarchy

Object Is a member of a class.

Class Defines the object type, what an object is able to do, what operations it can perform (methods), and what properties it has.





B. Property Nodes

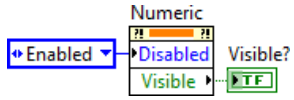
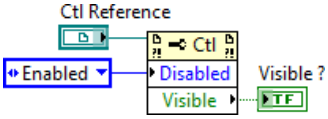
Objective: Demonstrate how to create property nodes and explain execution order.

Property Nodes

Some of the ways you can use property nodes include the following actions:

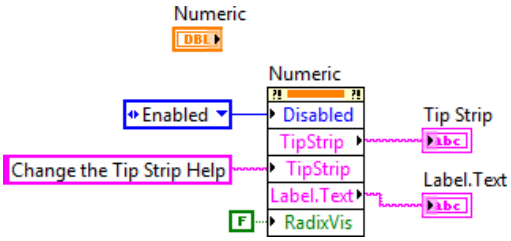
- Read and write the properties of an object
- Make modifications programmatically
- Use Context Help to get information about properties

The property nodes in the following table are equivalent approaches. You will learn more about explicitly linked property nodes later in this lesson.

Two Types of Property Nodes	
Implicitly Linked	
Explicitly Linked	

Multimedia: Using Property Nodes

Complete the multimedia module, *Using Property Nodes*, available in the `<Exercises>\LabVIEW Core 2\Multimedia\` folder.



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

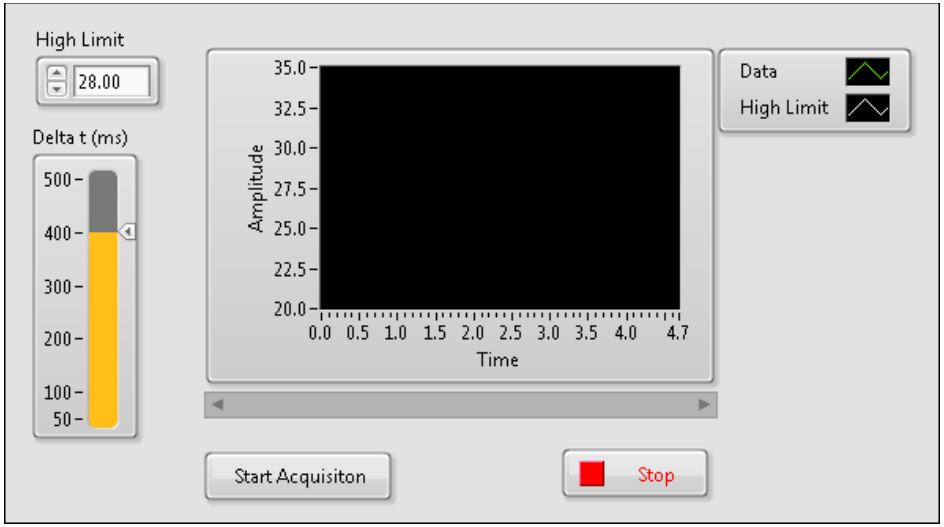
Part 4—Challenge: Change the Plot Color

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



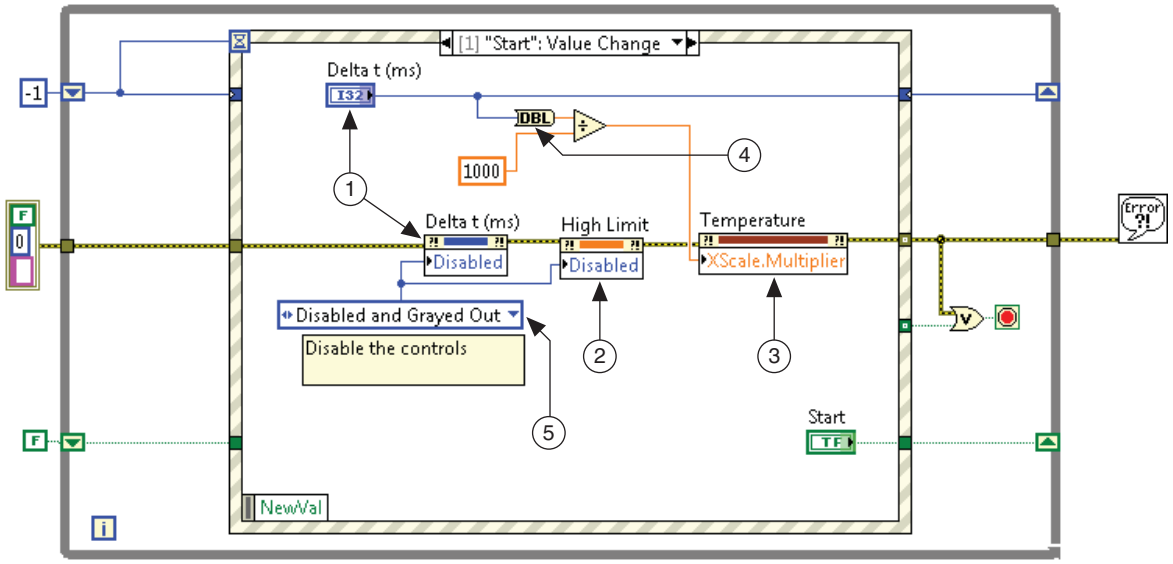
2. Run the VI and then click the **Start Acquisition** button.

   ☐ 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

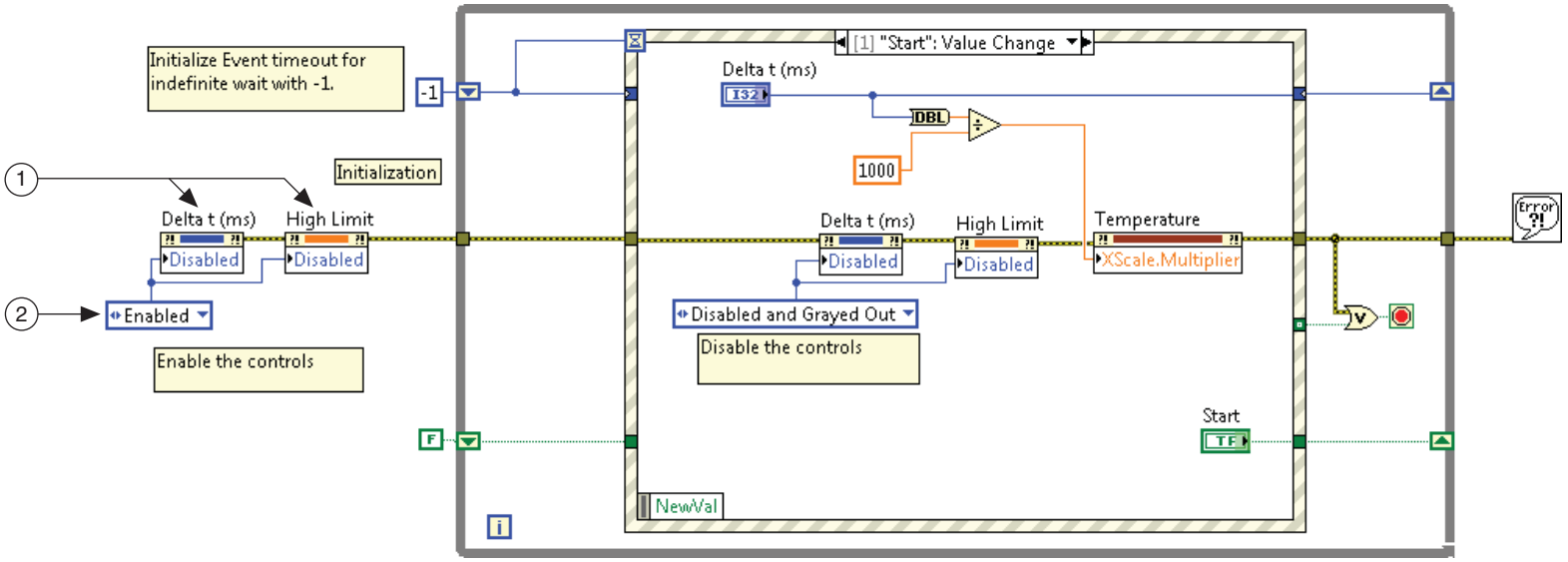
1. Run the VI and click the **Start Acquisition** button. The Delta t (ms) and High Limit controls are disabled and grayed out.
2. Stop the VI.
3. Run the VI a second time, click the **Start Acquisition** button and notice that the controls are still disabled.

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

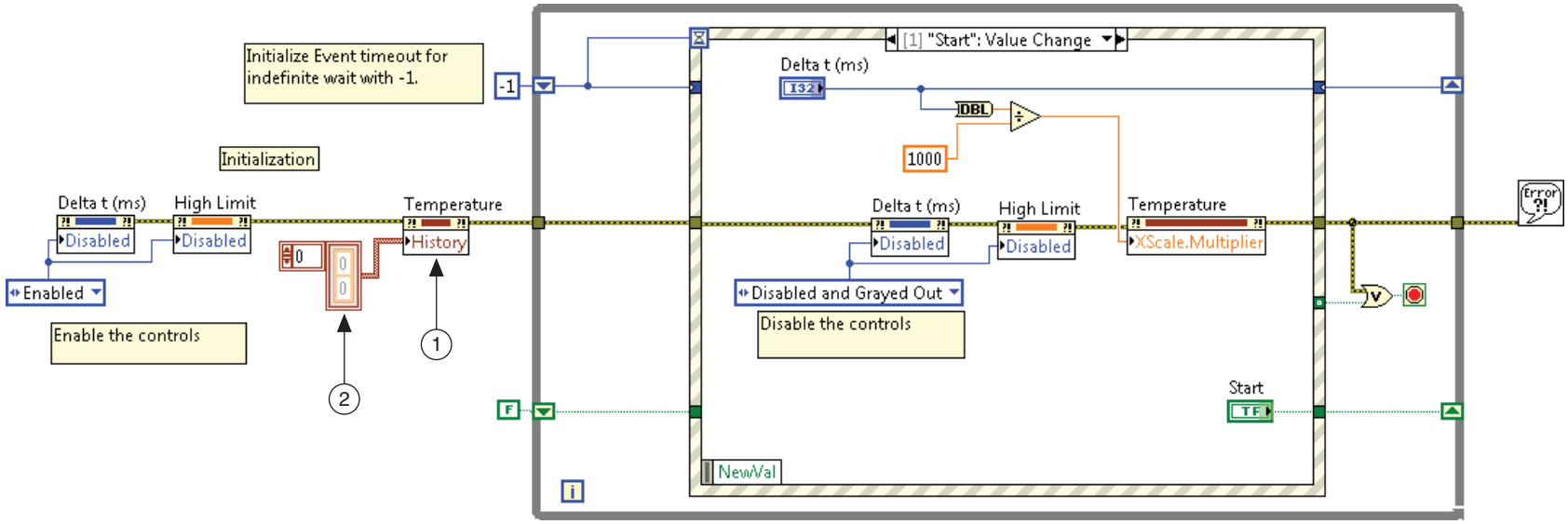
1. Run the VI and notice that the controls are enabled again before you click the **Start Acquisition** button.
2. Set different values for the controls and click the **Start Acquisition** button. Notice that the data displayed on the chart starts from where it stopped the last time you ran the VI.

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

C. Invoke Nodes

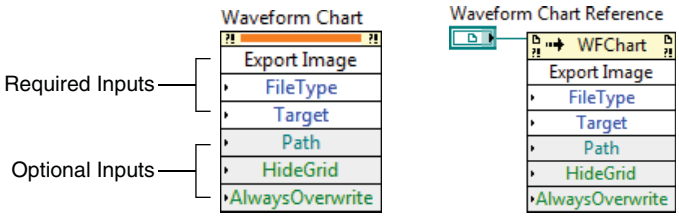
Objective: Demonstrate how to use Invoke Nodes to perform actions on controls of VIs.

Invoke Nodes

Use Invoke Nodes to perform the following:

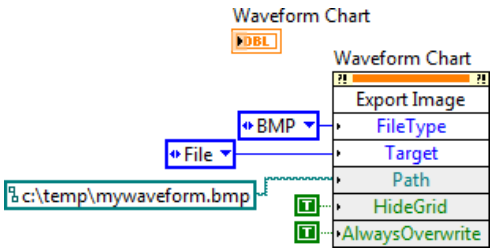
- Call methods or actions on objects.
  Examples: Get VI Version, Print VI panel, and Reinitialize All to Default
- Perform actions on referenced items

Invoke nodes can be implicitly linked or explicitly linked as shown in the following figure.



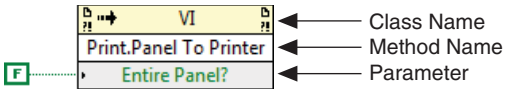
Demonstration: Create an Invoke Node

Create an Invoke Node for a front panel indicator.



VI Methods

Use a VI Server Reference to associate an Invoke Node with the current VI.



To create a VI method:

1. Place an Invoke Node on the block diagram.
2. Right-click and select **Select Class** to choose a class.
3. Right-click again and select **Select Method** to choose a method.

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

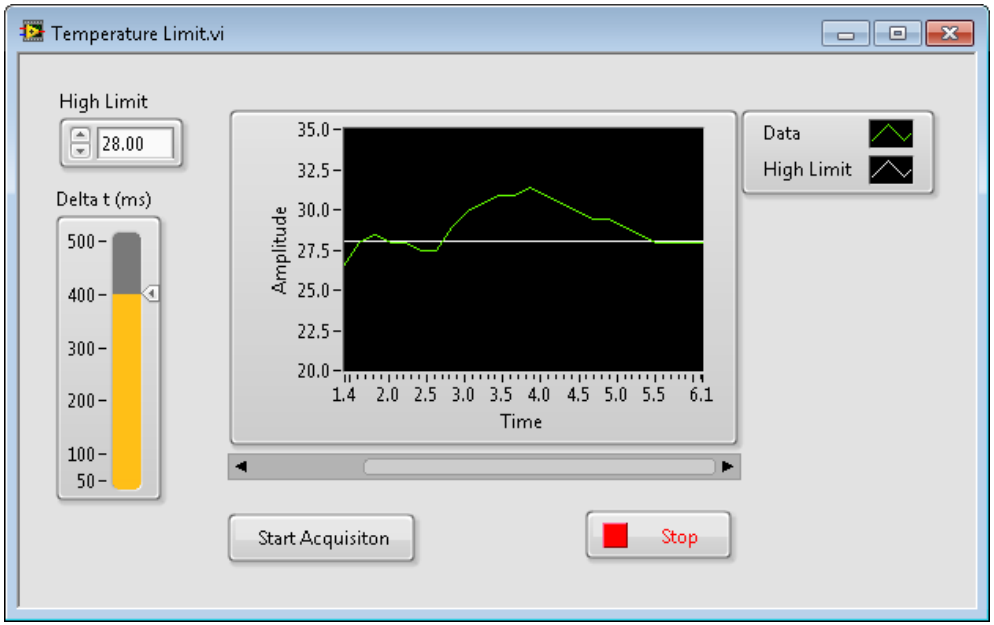
Unlike properties, a method has an effect every time you call it. Therefore, you should call methods only when you want to perform an action. For example, if you call the Fp.Center method during each iteration of a loop, the VI is continually centered, thereby preventing the user from moving it. You 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



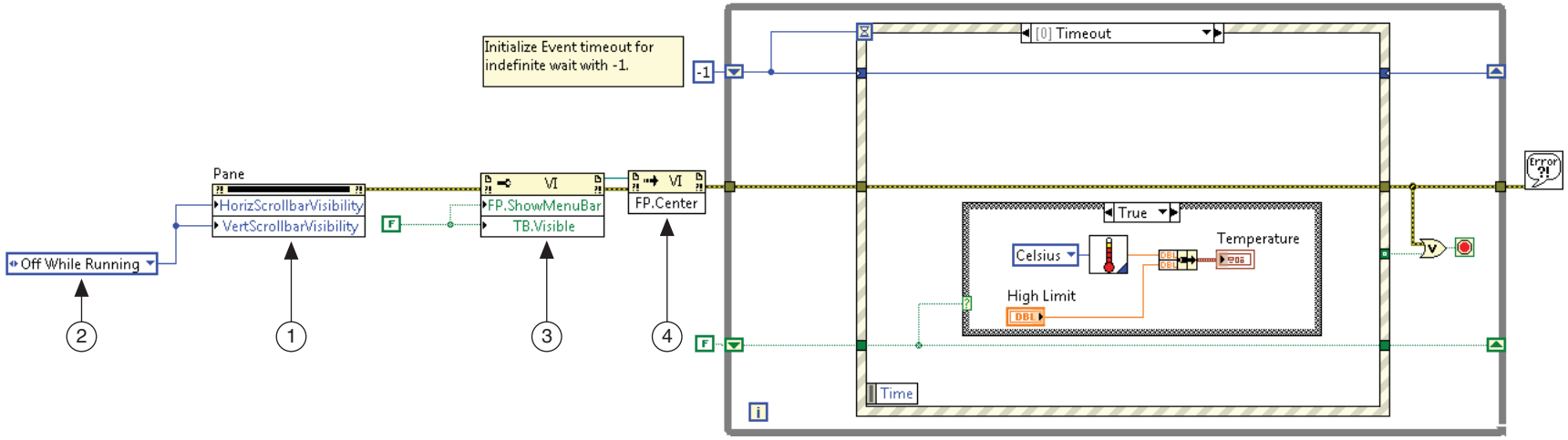
Implementation

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

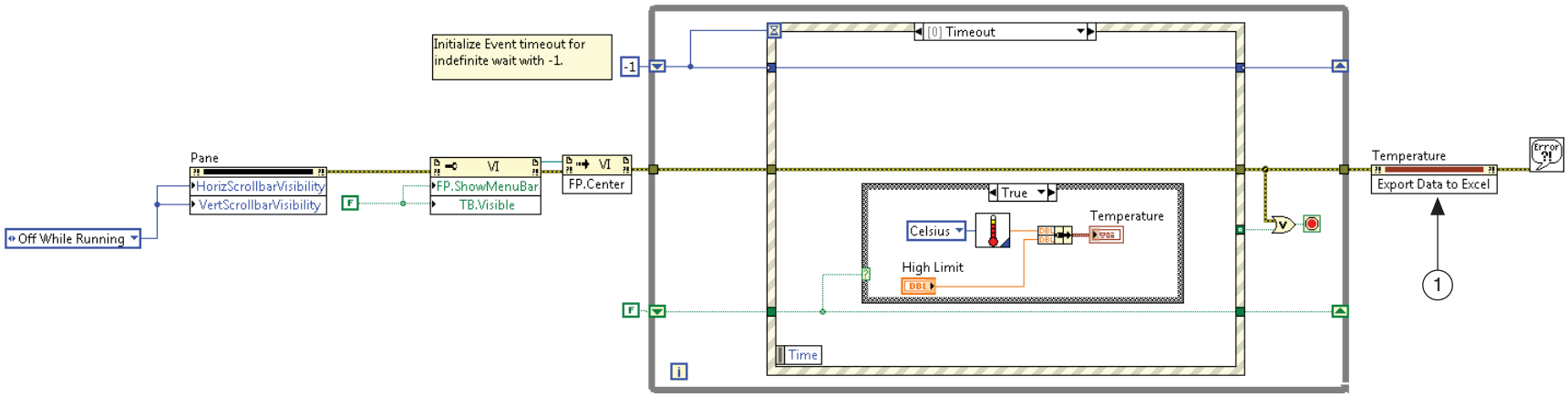
1. Run the VI.
2. Verify that the scroll bars, tool bar, and menu bar are not displayed, and that the front panel window is centered on the screen while the VI runs.
3. Stop the VI.

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.

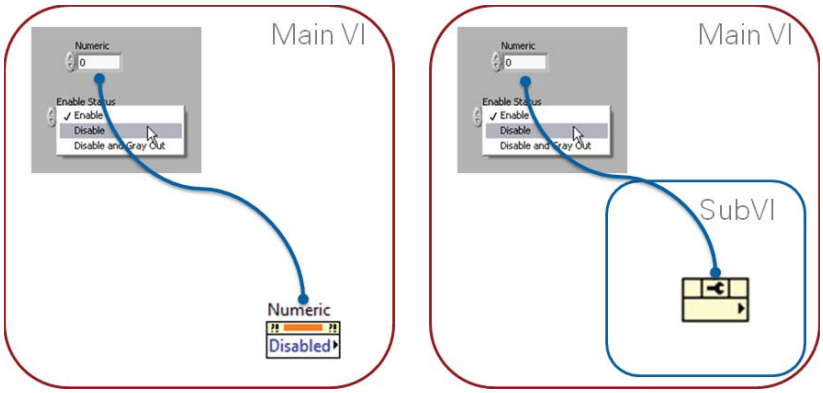
End of Exercise 4-2

D. Control References

Objective: Practice creating control references and explain the difference between strictly typed and weakly typed control references.

Control reference A reference to a front panel object.

Implicitly and Explicitly Linked Nodes



① ②

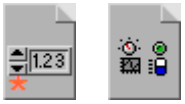
1 Implicitly Linked Property Node—The Property Node is linked to the front panel object.

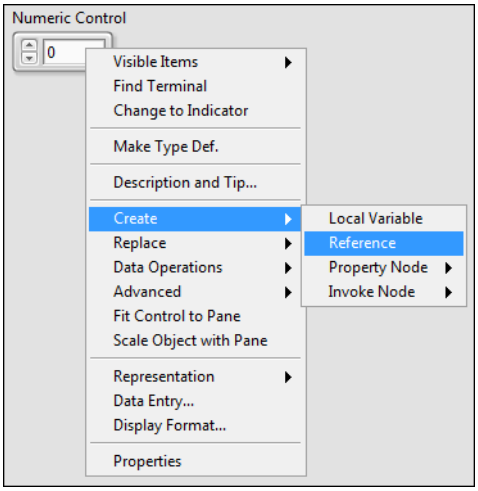
2 Explicitly Linked Property Node—You must use explicitly linked Property Nodes if the Property Node will be part of a subVI.

Multimedia: Creating Control References

This module covers the basics of creating control references and using them to programmatically modify front panel objects from a subVI.

Complete the multimedia module, *Creating Control References*, available in the `<Exercises>\LabVIEW Core 2\Multimedia\` folder.





Selecting the VI Server Class

You can specify a more generic or a more specific class.

	Advantages	Disadvantages
Specifying a more generic class, such as a Ctl Refnum instead of a Boolean Refnum	Allows the subVI to accept a reference to any type of front panel control.	Limits the properties available.
Specifying a more specific class such as a Num Refnum instead of a Ctl Refnum	Accesses more properties.	Makes the subVI more restrictive. A mismatch of refnum types results in an edit-time error.

Weakly typed Control refnums that are more flexible in the type of data they accept. For example, if the type of a weakly typed control reference is slide, you can wire a 32-bit integer slide, single-precision slide, or a cluster of 32-bit integer slides to the control reference terminal.

Strictly typed Control refnums that have data type information included in the connector pane of the VI.

Exercise 4-3 Create SubVIs for Common Operations

Goal

Use control references to create subVIs that modify VI properties.

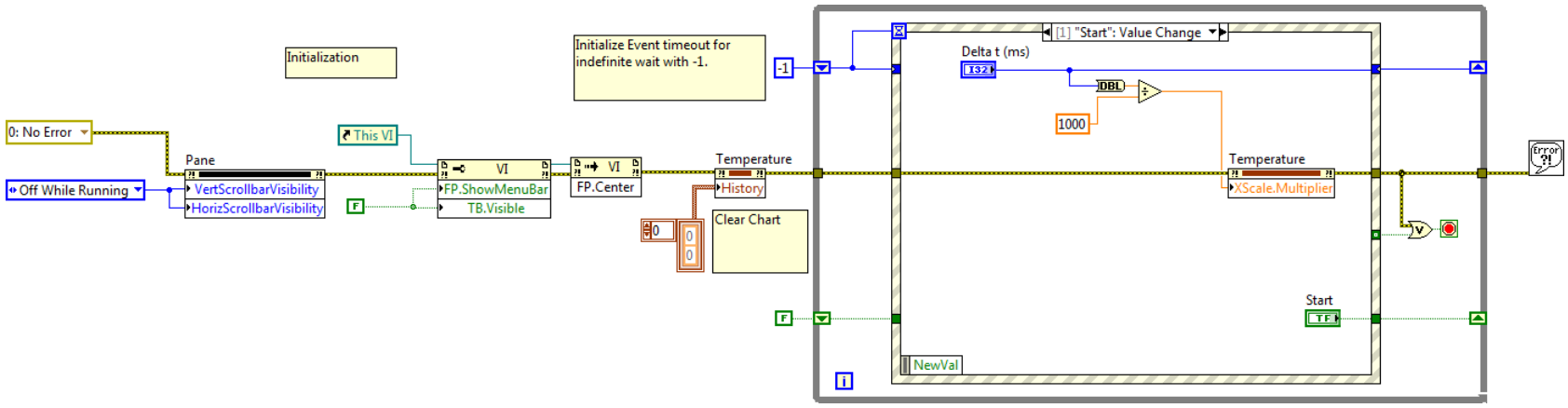
Scenario

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

Implementation

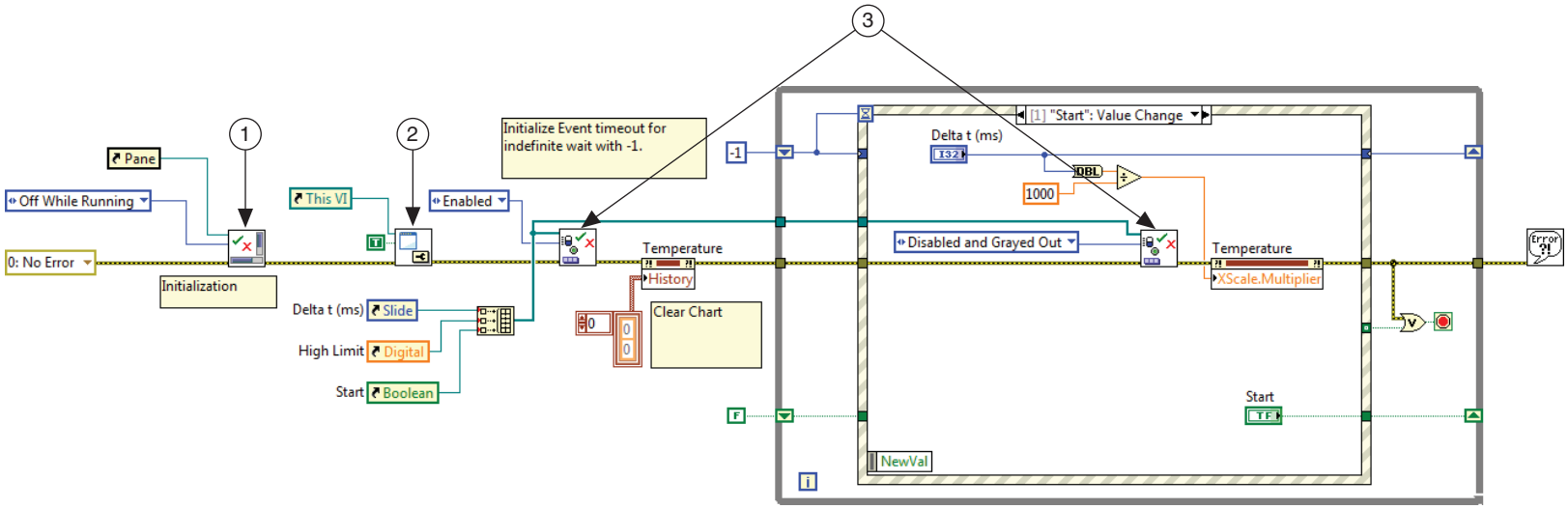
1. Open the Temperature Limit VI from the Temperature Limit Project located in the `<Exercises>\LabVIEW Core 2\Temp Limit - SubVIs` directory.
2. The block diagram, as shown in Figure 4-10 includes code to:
   - hide scroll bars while running
   - set dialog properties
   - reenable the controls when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

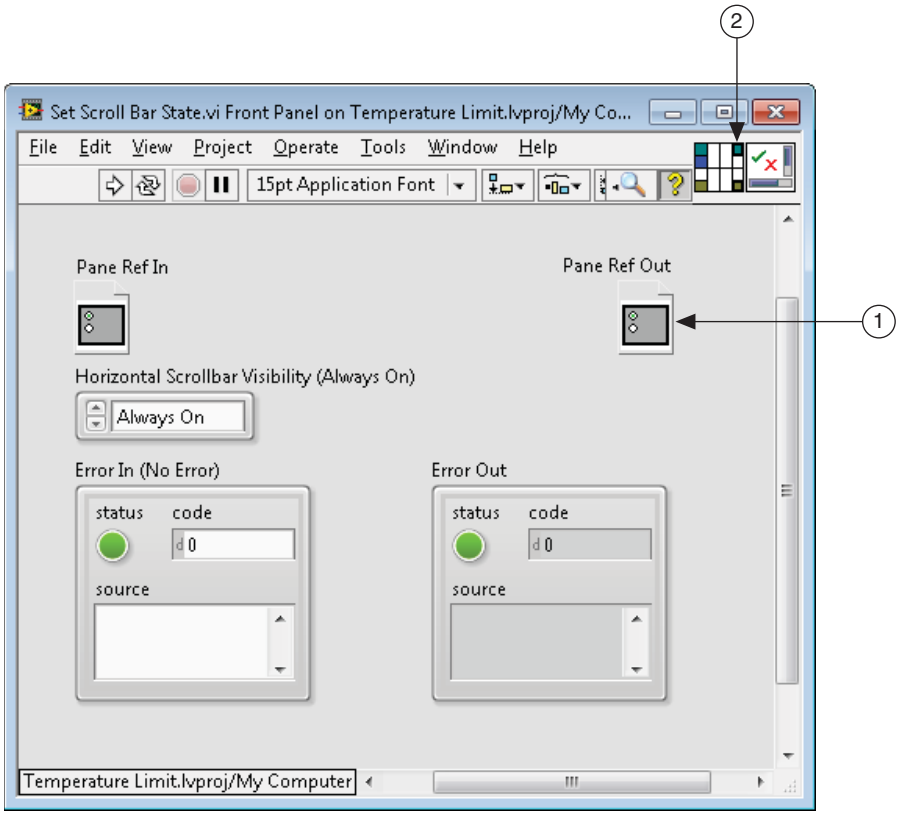
☐ On the Temperature Limit block diagram, highlight the code shown in Figure 4-12 and select **Edit»Create SubVI**.

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

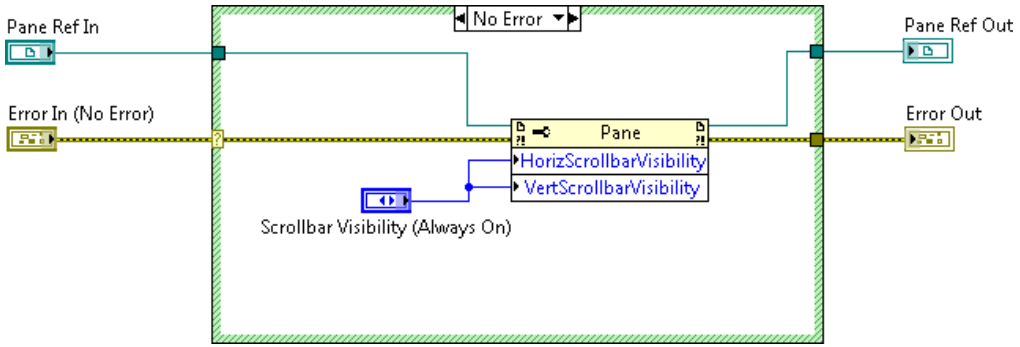


1 Pane Ref Out Indicator—Create a copy of the Pane Ref In control. Right-click the copy and select **Change to Indicator** and change the label.
2 Assign the Pane Ref Out indicator to the top right terminal of the connector pane. Connections for the other controls and indicators should already be created.

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



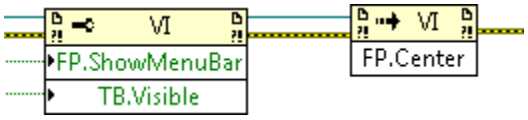
7. Wire the reference and error wires through the Error case.

8. Save and close the subVI.

9. Create the Set Dialog Properties subVI.

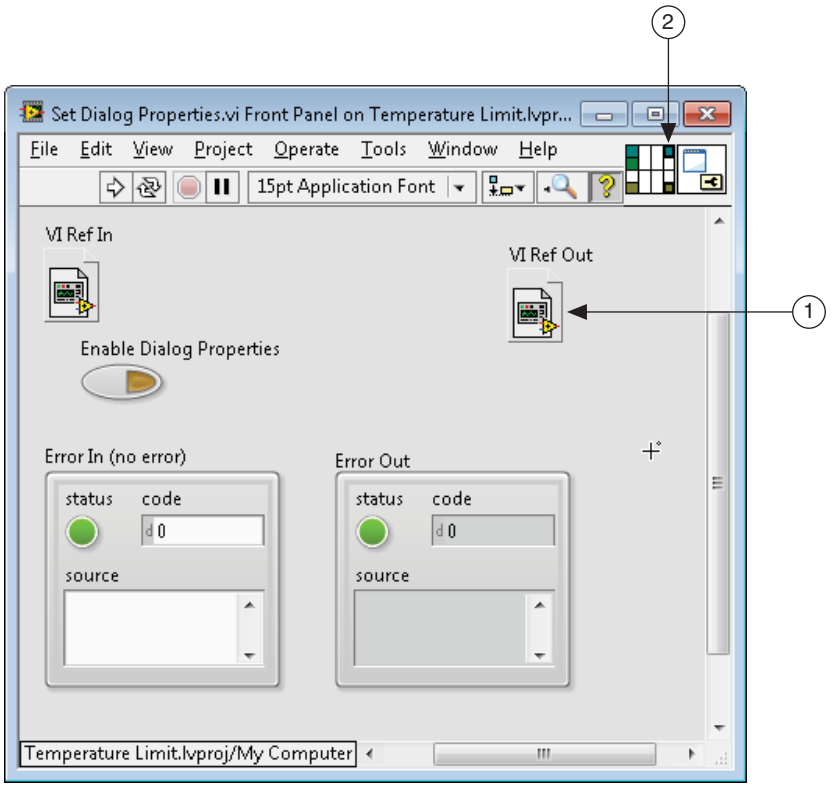
   ☐ On the Temperature Limit VI, highlight the code shown in Figure 4-15 and select **Edit»Create SubVI**.

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

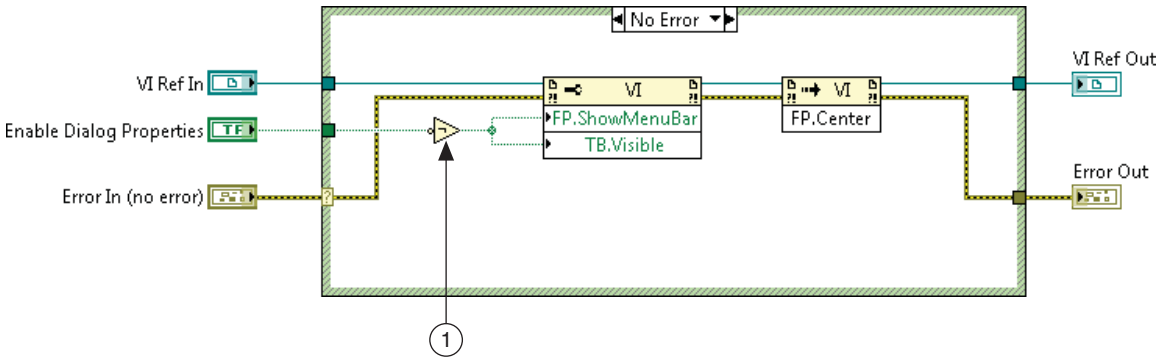


1 VI Ref Out Indicator—Create a copy of the VI Ref In Control. Right-click the copy and select **Change to Indicator** and change the label.
2 Assign the VI Ref Out indicator to the top right terminal of the connector pane. Connections for the other controls and indicators should already be created.

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.

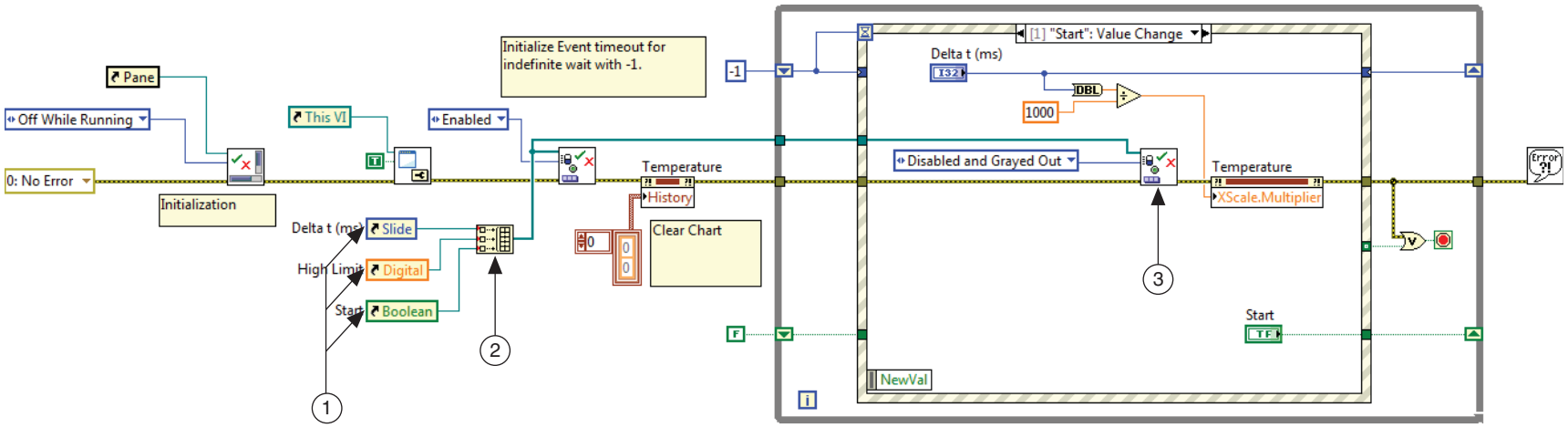
14. Save and close the subVI.

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

Additional Resources

Learn More About	LabVIEW Help Topic
VI Server architecture	*Capabilities of the VI Server* *Dynamically Loading and Calling VIs*
Property nodes and Invoke nodes	*Linking Property Nodes and Invoke Nodes to Terminals*
Control references	*Creating Control Reference SubVIs from Property Nodes and Invoke Nodes* *Switching Between Strictly Typed and Weakly Typed Control Refnums*

Activity 4-1: Lesson Review

1. For each of the following items, determine whether they operate on a VI class or a Control class.
   a. Format and Precision
   b. Visible
   c. Reinitialize to Default Value
   d. Show Tool Bar

2. You have a Numeric control refnum, shown below, in a subVI. Which of the following control references could you wire to the control refnum terminal of the subVI? (multiple answers)

   a. Control reference of a knob
   b. Control reference of a numeric array
   c. Control reference of a thermometer indicator
   d. Control reference of an LED

Activity 4-1: Lesson Review - Answers

1. For each of the following items, determine whether they operate on a VI class or a Control class.
   a. Format and Precision: **Control**
   b. Visible: **Control**
   c. Reinitialize to Default Value: **Control**
   d. Show Tool Bar: **VI**

2. You have a Numeric control refnum, shown below, in a subVI. Which control references could you wire to the control refnum terminal of the subVI?

   a. Control reference of a knob
   b. Control reference of a numeric array
   c. Control reference of a thermometer indicator
   d. Control reference of an LED

5 File I/O Techniques

In this lesson you will learn to create modular LabVIEW code that reads or writes measurement data stored in a file.

Topics

+ File Formats
+ Creating File and Folder Paths
+ Write and Read Binary Files
+ Work with Multichannel Text Files with Headers
+ Access TDMS Files in LabVIEW and Excel

Exercises

Exercise 5-1 Create File and Folder Paths
Exercise 5-2 Write Multiple Channels with Simple Header
Exercise 5-3 Write and Read TDMS Files

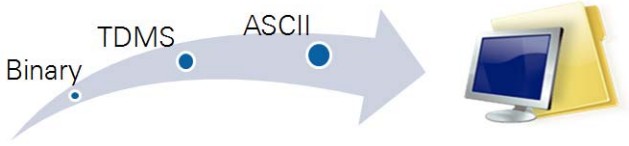
A. File Formats

Objective: Recognize appropriate use cases and the pros and cons of each file type.

Files store data as a series of bits.

At their lowest level, all files written to your computer's hard drive are a series of binary bits. However, many formats for organizing and representing data in a file are available. There are three common file types in LabVIEW:

- ASCII file format
- TDMS file format
- Direct binary data storage



Compare File Formats

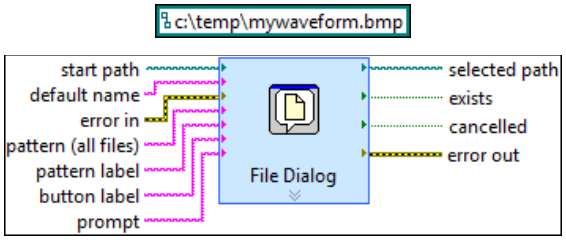
	ASCII	TDMS	Direct Binary
Numeric Precision	Good	Best	Best
Share Data	Best (Any program easily)	Better (NI programs easily; Excel)	Good (only with detailed format information)
Efficiency	Good	Best	Best
Ideal Use	Share data with other programs when file space and numeric precision are not important.	Store measurement data and related metadata. High-speed streaming without loss of precision.	Store numeric data compactly with ability to random access.

B. Creating File and Folder Paths

Objective: Programmatically create file and folder paths using LabVIEW VIs and tools.

Methods of Creating File and Folder Paths

You have several choices for creating file names and destination directories.

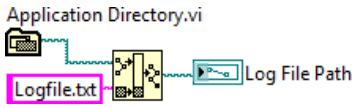


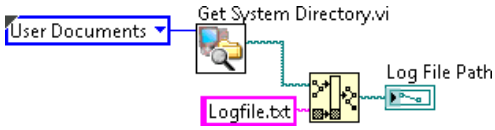
Method	Ideal Uses
Hard-coded paths	Quick prototypes
File dialog boxes	Application where a user needs to specify the path to a file or directory. You should customize the file dialog box options to limit the file types that the user can save.
Programmatic creation	Applications in which you want to create consistent filenames and extensions. Example: testdata_001.txt, testdata_002.txt, etc. Applications in which you want to specify a consistent location.

Multimedia: Creating File and Folder Paths

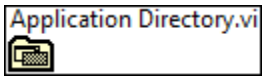
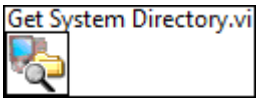
This module covers the basics of programmatically creating file and folder paths.

Complete the multimedia module, *Creating File and Folder Paths*, available in the `<Exercises>\LabVIEW Core 2\Multimedia\` folder.





Caveats of Programmatic Paths Creation

Application Directory VI	Get System Directory VI
	
Result depends on the calling VI.	Only works on Windows-based systems.
If you have not saved the VI or project, returns <Not a Path>	

Exercise 5-1 Create File and Folder Paths

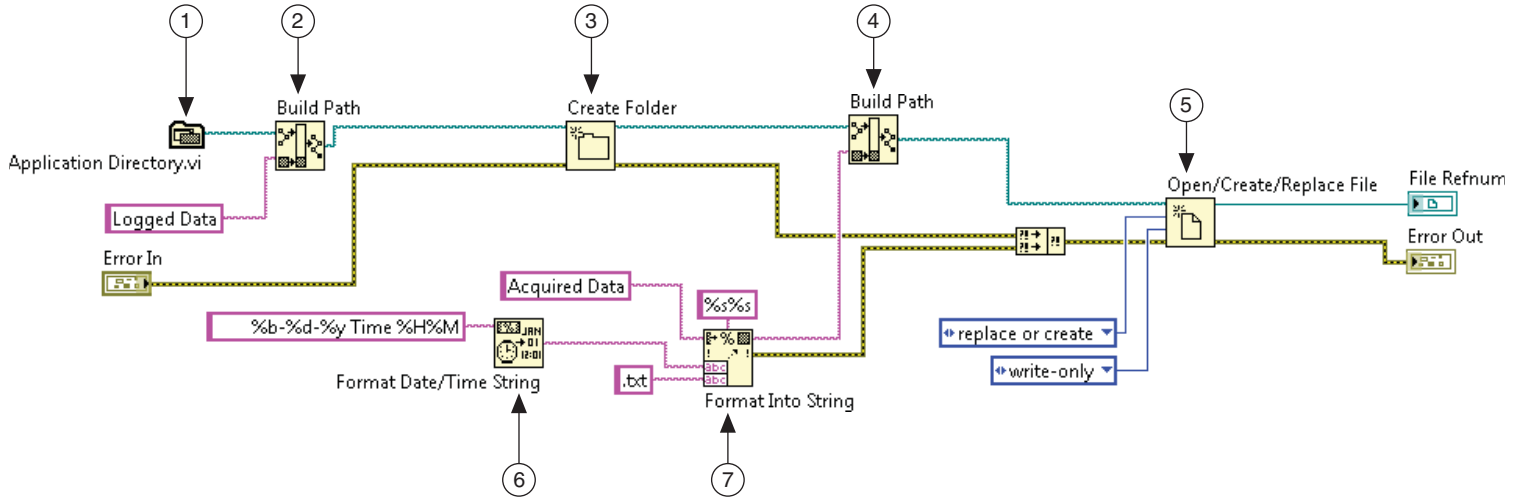
Goal

Modify a VI to programmatically create a folder if none exists or to replace the folder if it already exists.

Implementation

1. Open the `Create folder and file.lvproj` located in the `<Exercises>\LabVIEW Core 2\Create folder and file` directory.
2. Right-click **Create folder and file.lvproj** in the Project Explorer window and select **Explore**. Notice the files in the folder and notice that there is no folder called `Logged Data`.
3. Open **Create Data File.vi** from the Project Explorer window.
4. Examine the block diagram, as shown in Figure 5-1.

Figure 5-1. Create Data File VI Block Diagram—Start



1 Application Directory—Returns the path to the directory containing the application. This is useful because you are using a subVI to create the data file.
2 Build Path—Adds "Logged Data" to the file path name.
3 Create Folder—If the Logged Data folder does not exist, this function creates it.
4 Build Path—This instance of the Build Path function appends the auto generated file name `Acquired Data <Date><Time>.txt` to the file path.
5 Open/Create/Replace File—Replaces or creates the file.
6 Format Date/Time String—Generates a pre-formatted string containing the current date and time.
7 Format Into String—Concatenates the strings to build the file name.

5. Use the Context Help and Detailed Help for the Format Date/Time String function to identify the meaning of the time-related format codes used in this VI.

   ☐ Open the Context Help if it is not already open.

   ☐ Move the cursor over the Format Date/Time String function and then click the **Detailed Help** link in the Context Help window.

   ☐ Read the Format Date/Time String Function help topic to learn the meaning of the following codes:

   - %b
   - %d
   - %y
   - %H
   - %M

6. Run the VI to test the functionality.

   ☐ Turn on execution highlighting.

   ☐ Run the VI once and watch the flow of data on the block diagram.

   ☐ Open the `<Exercises>\LabVIEW Core 2\Create folder and file` directory and notice that a new folder named `Logged Data` was created and contains an empty file. Check that the filename includes the date and time formatted components created using the codes used in the VI.

7. Run the VI a second time.

   ☐ Notice that an error occurs at the Create Folder function.

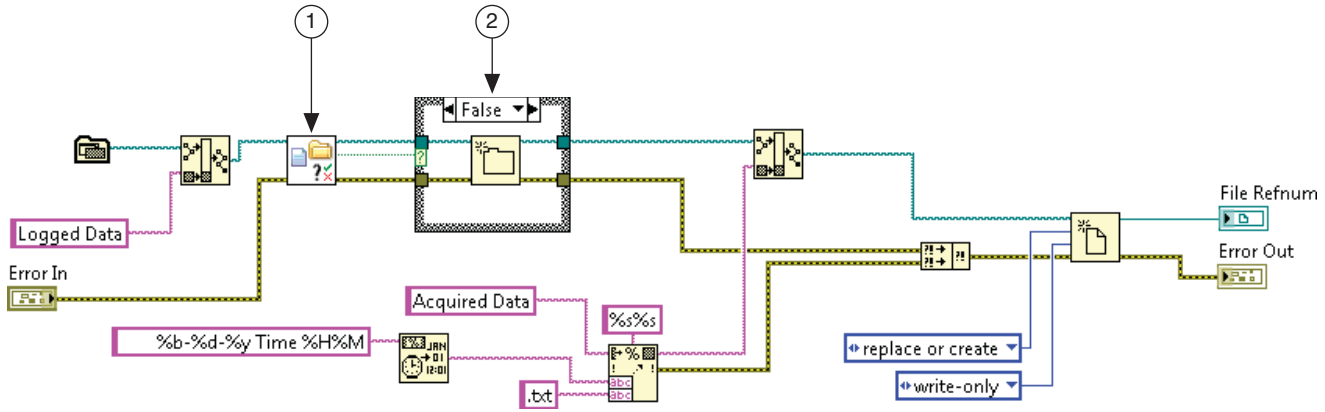
   ☐ Open the front panel, right-click the Error Out cluster and select **Explain Error**. The error code 10 Duplicate Path is returned because the folder already exists.

8. Modify the code to check if the `Logged Data` folder already exists as shown in Figure 5-2.

9. Only if the folder does not exist, create it. Therefore, you enclose the Create Folder function in a Case structure. You use the output of the Check if File or Folder Exists VI to conditionally call the Create Folder function.

10. Modify the block diagram to conditionally create the folder.

Figure 5-2. Create Data File VI Block Diagram—Complete



1 Check if File or Folder Exists VI—Checks to see if the file or folder exists and outputs this information to the Create Folder function.
2 Case Structure—Add a Case Structure around the Create Folder function and then click the Case structure border and select **Make this Case False**. The folder is created only if it does not already exist.
   ☐ Wire the reference and error wires through the True case.

11. Save the VI and run it again. Notice that a new .txt file is created and the VI doesn't return an error.

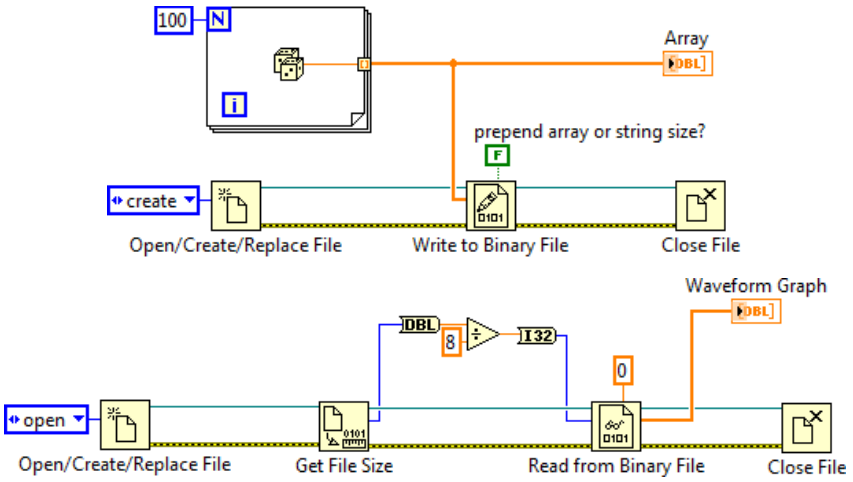
End of Exercise 5-1

C. Write and Read Binary Files

Objective: Identify binary file I/O VIs and functions from the File I/O palette and identify the pros and cons of using them.

Using Binary File Functions

Although all file I/O methods eventually create binary files, you can directly interact with a binary file by using the Binary File functions.



- The use of binary files should only be considered after evaluating using text and TDMS files.
- Compared to other formats, binary files are complex to work with, requiring an intimate knowledge of the file format.
- Debugging code can add considerable time to the development cycle if things go wrong.
- Developers should review the binary file examples that ship with LabVIEW, including DAQmx examples, for more information.

Demonstration: Writing a Bitmap File

Navigate to `<Exercises>\Bitmap File Writer\` and explore how to use Binary File I/O to write a bitmap format file.

D. Work with Multichannel Text Files with Headers

Objective: Programmatically create and manipulate header information and data stored in text files.

Creating Text Files with Headers

You create headers to aide understanding of the data.

Constant Name	Icon
Tab	
End of Line	

No Header Data

24.45		34.54	
23.41		35.32	
22.97		35.98	
21.56		36.76	

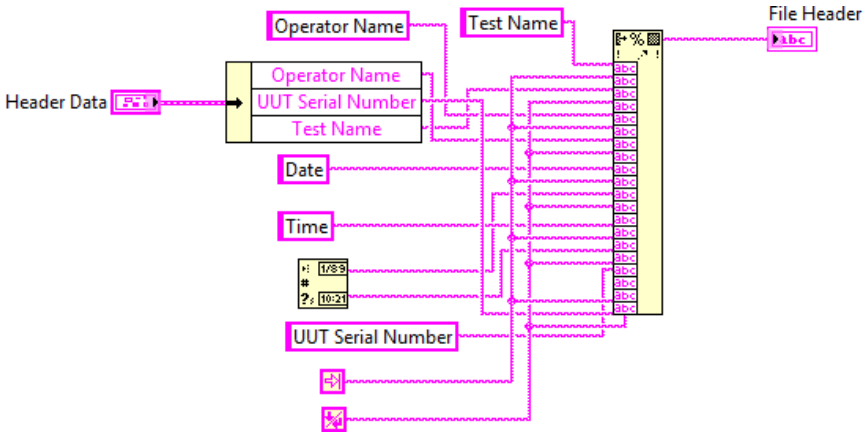
With Header Data

Operator Name		David			
UUT S/N		A1234			
Test Name		Pressure			
Channel Name		Temperature		Pressure	
Units		Kelvin		PSI	
Max. Value		24.45		36.76	
		24.45		34.54	
		23.41		35.32	
		22.97		35.98	
		21.56		36.76	

Tip TDMS files may be a better solution for creating files with complex or dynamic headers. A TDMS file allows the developer to write data and modify the properties (headers) at any time.

Creating Text Files with Headers—Hard-Coding

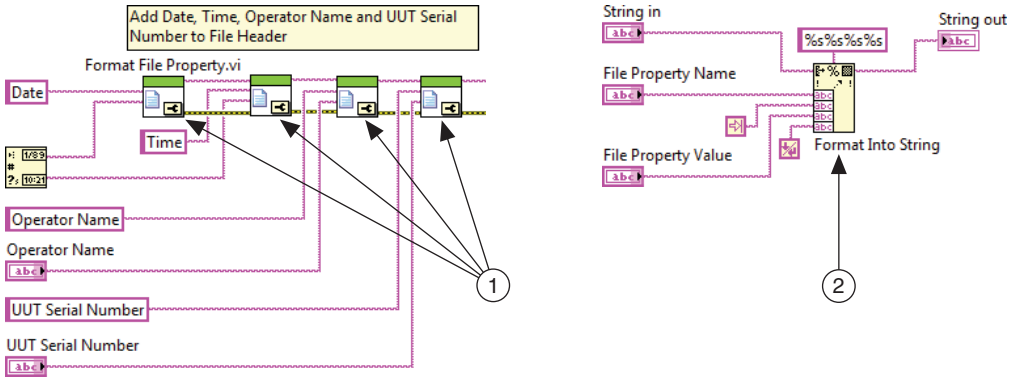
You can hard-code file header information by using the Format Into String function to create a tab-delimited line of text as shown in the figure below.



Using the Format Into String function is good for quickly coding simple headers. However, using this method for complex strings can make it difficult to debug and modify the code.

Creating Text Files with Headers—SubVIs

Using subVIs to create header information is more scalable and easier to manage than hard-coding file headers.



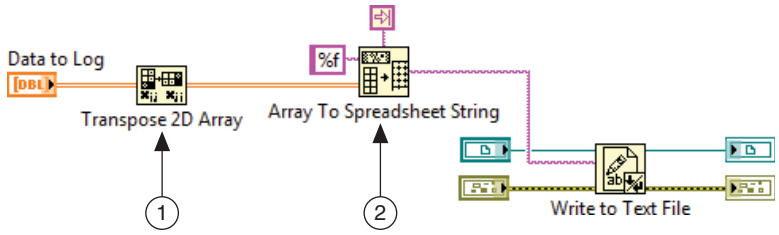
1 Using a subVI to handle formatting, you need to modify only one VI instead of changing the code in four places.
2 The subVI adds data to the string and a column separator (tab or comma).

Writing Multiple Channels

LabVIEW stores multichannel data in multidimensional arrays using row-major order. You can identify rows by the first index of a 2D array and columns by the second index. For example, a 2 × 3 element 2D array in LabVIEW looks like this:

1	2	3
4	5	6

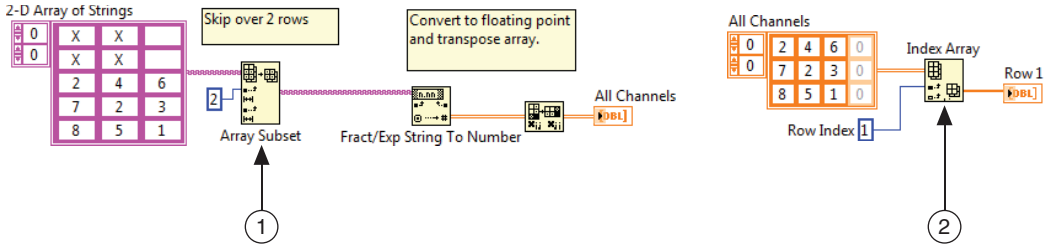
Transpose data before writing to file to view channel data in column format.



1 Transpose the array to create a column format.
2 Converts the array to a table of strings. For two-dimensional arrays, the resulting table separates column elements with a user-specified delimiter and separates rows with an end of line character. The default column separator is a tab character.

Reading Channel Data

The figure below demonstrates how to extract measurement data from a string table that includes header information (represented by "X" elements).



1 Use the Array Subset function to skip over the first two rows of data if you want to ignore the header information (represented by "X" elements) and read only the measurement data.
2 The Index Array function returns the element or subarray of n-dimension beginning at the row index you specify.

Exercise 5-2A Write Multiple Channels with Simple Header

Goal

Build cohesive, modular, and readable VIs that allow for application scalability and maintainability.

LabVIEW arrays store channels in rows and text files typically store channel data in columns (for example, Excel prefers data in columns).

Use LabVIEW functions to easily convert numeric arrays into strings.

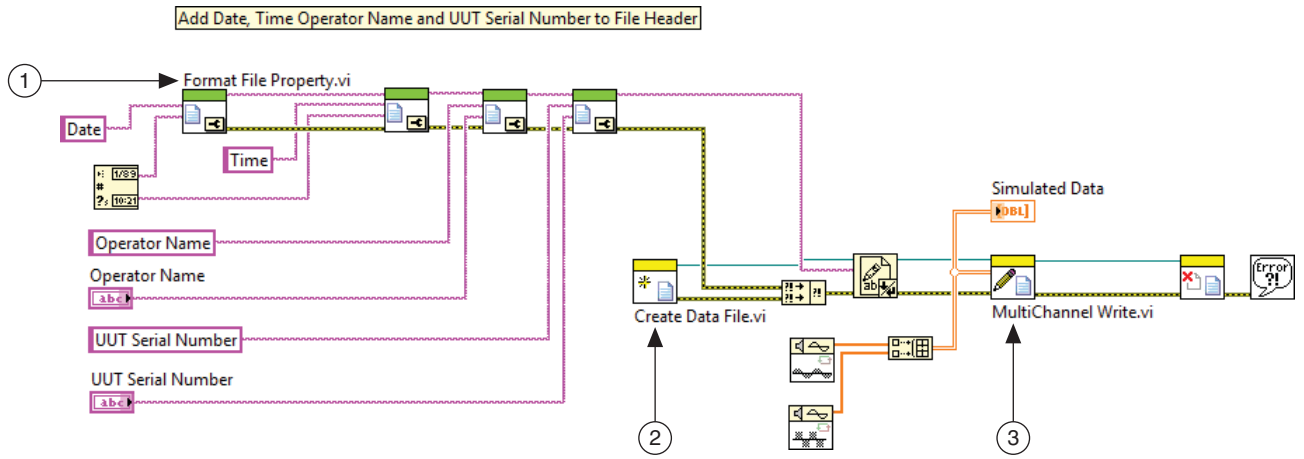
Implementation

1. Open the `Write Multiple Channels with Simple Headers.lvproj` located in the `<Exercises>\LabVIEW Core 2\File IO - MultiChannel with Header` directory.

2. Open **Write Multiple Channels with Simple Header.vi** from the **Project Explorer** window and open the block diagram.

   This VI uses several subVIs in order to modularize the code and make it more readable. In this exercise you complete the Format File Property VI and the MultiChannel Write VI.

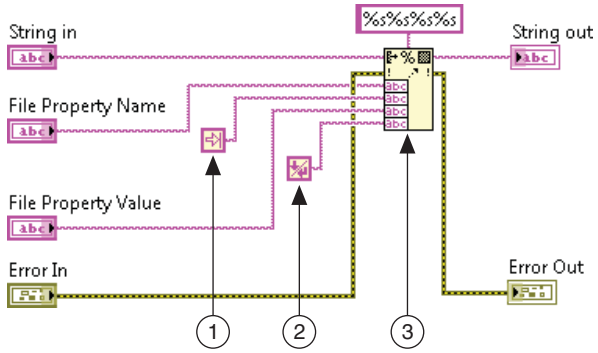
Figure 5-3. Write Multiple Channels with Simple Header VI Block Diagram



1 Format File Property VI—Four instances of this VI are used to add Date, Time, Operator Name, and UUT Serial Number to the file header.
2 Create Data File VI—As you saw in Exercise 5-1, this VI programmatically creates and saves a file.
3 MultiChannel Write VI—The VI modularizes the writing of data to a text file. You modify this VI in step 5.

3. From the block diagram of the Write Multiple Channels with Simple Header VI, double-click the Format File Property subVI and modify the block diagram as shown in Figure 5-4.

Figure 5-4. Format File Property VI Block Diagram



1 Tab Constant—Creates a constant string containing the ASCII HT (horizontal tab) value.
2 End of Line Constant—Creates a constant string containing the platform-dependent end-of-line value.
3 Format Into String—Double-click the node to open the **Edit Format String** dialog box and select **Format string (abc)** from the **Selected operation (example)** pull-down menu.

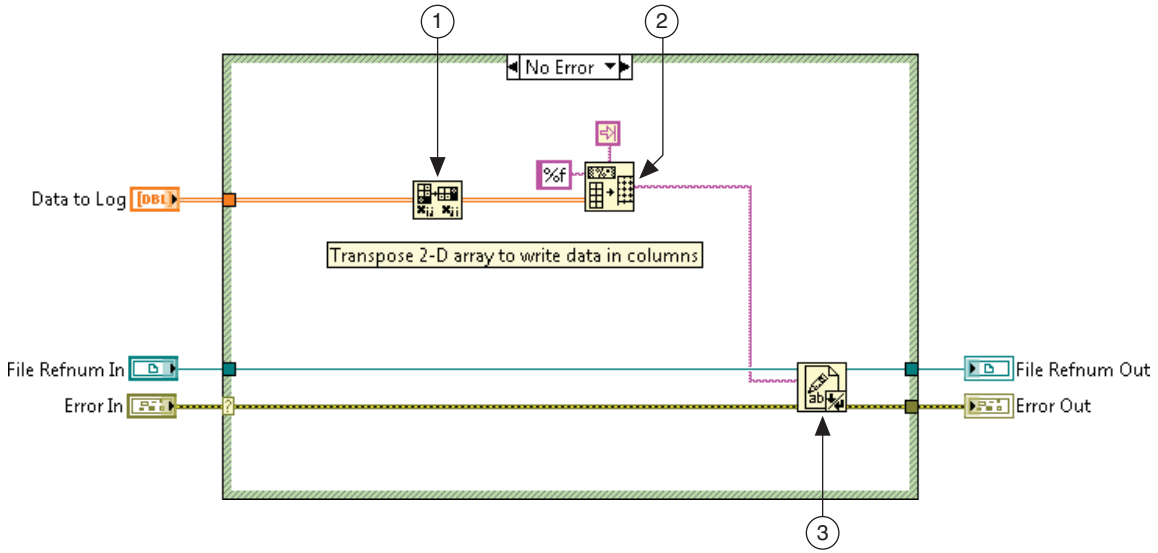
- ☐ Click the **Add New Operation** button three times to add three more Format string operations to the **Current format sequence** listbox.
- ☐ Click **OK** to close the dialog box.
- ☐ Notice that the Format Into String function now has four inputs.

Note When you add operations to the Current format sequence in the **Edit Format String** dialog box, LabVIEW adds a space between each one by default. You can remove the spaces in the **Corresponding format string** section of the dialog box.

4. Save and close the Format File Property VI.

5. From the block diagram of the Write Multiple Channels with Simple Header VI, double-click the MultiChannel Write subVI and complete the No Error case, as shown in Figure 5-5.

Figure 5-5. MultiChannel Write SubVI Block Diagram



1 Transpose 2D Array—Because LabVIEW stores array data in rows by default, this function transposes your array to write the data into columns.
2 Array to Spreadsheet String—Converts the array to a table in string form.
3 Write to Text File—Writes a string or an array of strings to lines in a file.

6. Save and close the MultiChannel Write VI.

Test

1. Run the Write Multiple Channels with Simple Header VI and create a data file.

   This VI uses the Create Data File VI that you used in Exercise 5-1 to programmatically create the `Logged Data` directory and the data file in the same directory containing the application.

2. Add the `Logged Data` directory to your project.

   ☐ In the Write Multiple Channels with Simple Header LabVIEW Project Explorer window, right-click **My Computer** and select **Add»Folder (Auto-populating)**.

   ☐ Navigate to `<Exercises>\LabVIEW Core 2\File IO - MultiChannel with Header\Logged Data` and click the **Select Folder** button.

   ☐ Open the `Logged Data` directory from the **Project Explorer** window and notice that it contains the file you created when you ran the Write Multiple Channels with Simple Header VI.

   ☐ Open and examine the generated file in the `Logged Data` directory.

3. Read the data file.

   ☐ Run the Read Multiple Channels with Simple Header VI.

   ☐ Navigate to `<Exercises>\LabVIEW Core 2\File IO - MultiChannel with Header\Logged Data` and select the data file you just created.

4. Save and close all open VIs. Save and close the Write Multiple Channels with Simple Header project.

Exercise 5-2B Challenge

Goal

Create a VI that takes the property name as input and returns the associated property value.

Scenario

Your file header includes a variable number of property names and values. After reading the file, you want to programmatically access the property value for a given property. For example, in this exercise, one of the properties was “UUT Serial Number”. You want to programmatically search for “UUT Serial Number” and then return the serial number value (e.g., “A12345”).

Description

After reading the spreadsheet string into memory, use a modular approach to create a VI which finds the row index of a property name. Then use the row index to return the property value. If the property name isn't found, the VI should return a descriptive error.

Exercise 5-2C Challenge

Goal

Create a VI that creates tab-delimited column headers for an arbitrary number of channels.

Scenario

The MultiChannel Write VI can write data for an arbitrary number of channels using a two dimensional array. You want to create a channel header for each channel of data. Since the number of channels is arbitrary, you need a scalable approach which allows for a variable number of channel headers. For example, if you have three channels of data—temperature, pressure, and strain—you want to write the data to a file with the following headers: Temperature, Pressure, and Strain. To format the file correctly, you want to use the same delimiters in the header as in the channel data. For example, if the channel data is tab-delimited with each line terminated with an end-of-line character, you want your header to be formatted the same way.

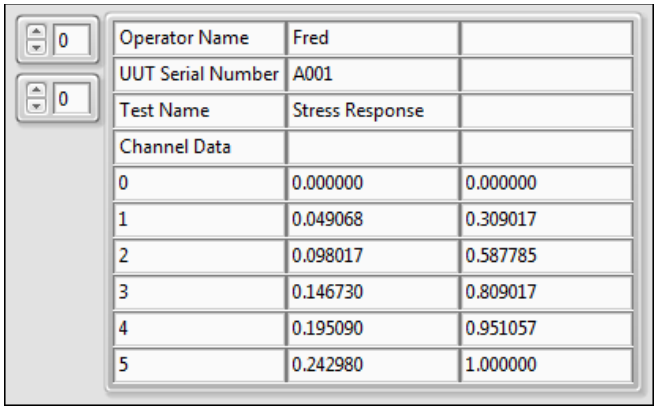
Description

Create a Write Channel Data with Headers VI which you call prior to your MultiChannel Write VI that writes a channel header for each channel of data. The VI should take an array of strings as input. Your subVI should write the strings to file so that a tab separates each string and the line is terminated by an end-of-line character.

End of Exercise 5-2

Activity 5-1: Read Data and Extract Information

Consider the measurement data shown below. Consider that you have a variable number of file header properties.



Operator Name	Fred	
UUT Serial Number	A001	
Test Name	Stress Response	
Channel Data		
0	0.000000	0.000000
1	0.049068	0.309017
2	0.098017	0.587785
3	0.146730	0.809017
4	0.195090	0.951057
5	0.242980	1.000000

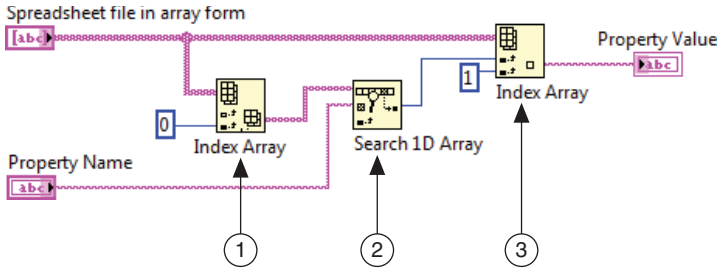
How could you programmatically extract a specific property value given the property name?

How could you programmatically read the UUT Serial Number value?

How can you programmatically extract a single channel of data?

Given a Property Name, Find the Value

The code below illustrates how you can programmatically extract a file property value by searching for a property name match in the first column and reading the associated value in the second column.

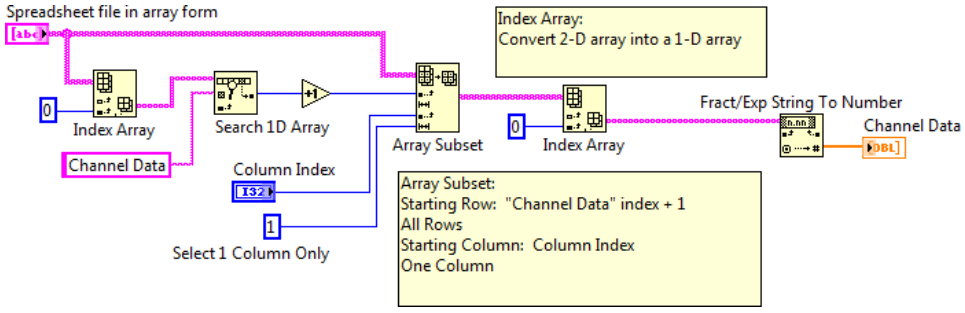


1. Use the Index Array function to put column 0 in a 1D array. You can extract the first column of an array by specifying 0 in the column index and leaving the row index unwired.
2. Use the Search 1D Array function to find a Property Name match in the array and return the Property Name index. Because the search is linear, you do not need to sort the array first. LabVIEW stops searching as soon as the element is found.
3. Use the Index Array function again and specify the row index returned from the Search 1D Array function and a column index of 1 to extract a specific element.

Tip If the Search 1D Array function fails to find a match, the function returns a value of -1. You should check for the value of -1 and might want to force an error if the Search 1D Array function returns a -1.

Extracting a Data Channel

The following illustration shows how to extract one column of the measurement data if you have a variable number of file properties preceding measurement data in a file.



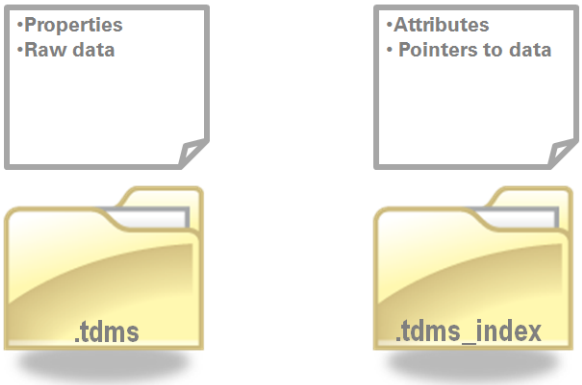
Mark the start of the data with an entry called Channel Data. If you can detect the Channel Data entry you know that the measurement data starts on the next row. Use the matching property name technique to detect the “Channel Data” entry in column 0. Then use the Array Subset function to extract all the measurement data starting in the next row.

E. Access TDMS Files in LabVIEW and Excel

Objective: Identify TDMS file I/O VIs and functions from the File I/O palette and identify the pros and cons of using them.

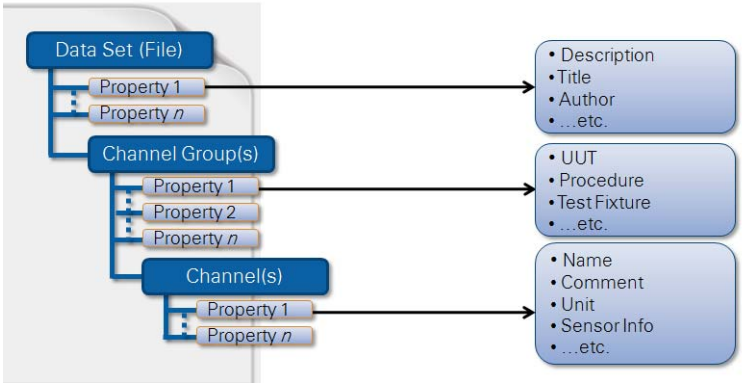
TDMS File Format

The TDMS file format was designed specifically for saving measurement data to disk. Use the TDMS file format to store channel-specific properties such as channel name, measurement units, test limits, and sensor information.



TDMS Files—Data Hierarchy and Properties

Use TDMS files to organize your data in channels and channel groups.

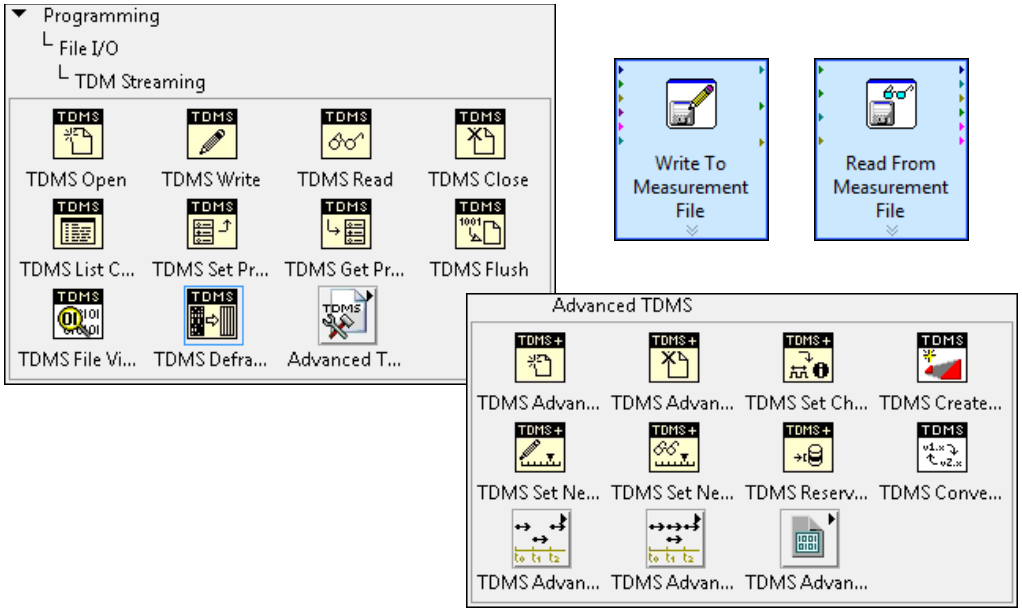


Channel A segment of a TDMS file that stores measurement signals or raw data as binary data. Each channel can have properties that describe the data.

Channel group A segment of a TDMS file that contains properties to store information as well as one or more channels. Use channel groups to organize your data and to store information that applies to multiple channels.

TDMS Functions

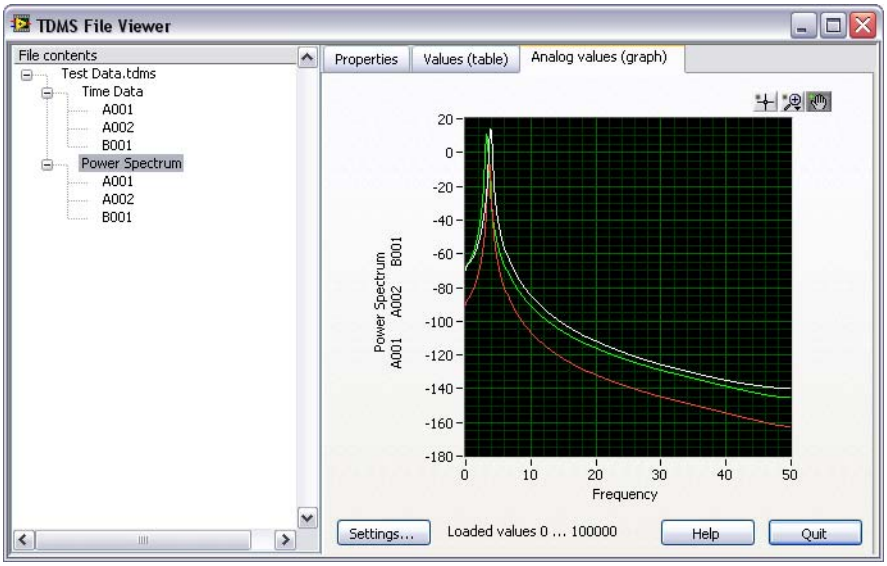
The TDM Streaming palette includes both standard and advanced TDMS VIs and functions. The Advanced TDMS VIs and functions enable you to perform advanced file I/O operations, such as asynchronous reads and writes, on .tdms files.



Note Use the standard TDMS functions when you do not need the features of the Advanced TDMS functions. Incorrectly using Advanced TDMS functions might result in a corrupt `.tdms` file.

TDMS Files—File Viewer

Use the TDMS File Viewer VI to automatically see everything written to a TDMS file. Place the TDMS File Viewer VI after the `.tdms` file is closed. The TDMS File Viewer launches another window in which you can view the data and properties inside the TDMS file.



Exercise 5-3 Write and Read TDMS Files

Goal

Log data to a TDMS file and read the same TDMS file to access information about a specific channel.

Scenario

You are given a TDMS Logger VI that generates measurement data for any number of units under test (UUTs). The UUT measurement data consists of a time domain waveform and the power spectrum of a waveform.

Run the TDMS Logger VI that accepts UUTs identified by serial numbers. The TDMS Logger VI retrieves the measurement data from the Generate Data VI, and logs the UUT data and additional properties to a TDMS file.

The TDMS file contains the author, timestamp, and two channel groups—Time Data and Power Spectrum Data. Each group contains a channel for each UUT. The serial number of the UUT names each channel and contains the matching signal data.

Saving data to a file serves no purpose unless you also can access the data. Create a reader VI to access data from the TDMS file you generated. The reader should return either time data or power spectrum data for a particular UUT serial number.

Note Optionally, if you have Microsoft Excel installed on your system, you can use the TDM Excel Add-In tool to load the TDMS file into Microsoft Excel.

Design—TDMS File Reference Information

- File Level Information
  - **Time Stamp**—contains the current time.
  - **Author**—contains the test operator name, acquired through a front panel control.
  - The file contains two channel groups, one for time data and one for the power spectrum data.
- Channel Group Level Information
  - **Name**—contains `Time Data` or `Power Spectrum Data`. This identifies the channel group.
  - Each channel group should contain a channel for each UUT.
- Channel Level Information
  - **Name**—contains the UUT Serial Number, which associates the numeric data with a particular unit.
  - **Signal**—contains an array of floating-point numeric data.
  - Several other properties, such as the signal minimum and maximum, will automatically be calculated and added to the file.

Design—TDMS Reader Inputs and Outputs

Table 5-1. TDMS Reader VI Inputs and Outputs

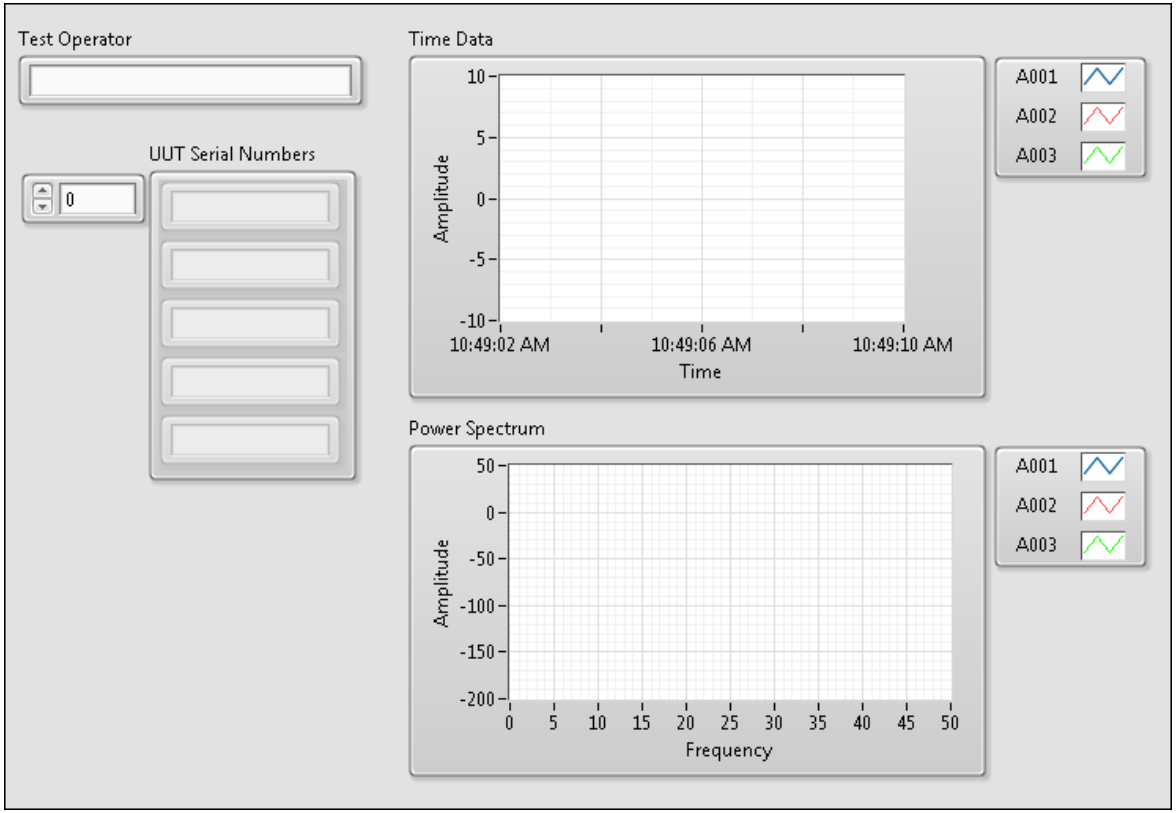
Type	Name	Properties
String Control	`Serial Number`	—
Combo Box	`Data Set`	Item 1 = “Time Data” Item 2 = “Power Spectrum”
Waveform Graph Indicator	`Channel Data`	—
String Indicator	`Test Operator`	—
Time Stamp Indicator	`Time of Test`	—

Your VI should begin by opening the TDMS file and reading the author and time stamp file properties, then read the time data or power spectrum data for the specified UUT and display the data on the Channel Data waveform graph.

Implementation—TDMS Logger

1. Open `TDMS Logger.vi` from the TDMS Logger and Reader Project located in the `<Exercises>\LabVIEW Core 2\TDMS Logger and Reader` directory. This VI is pre-built for you as shown in Figure 5-6.

Figure 5-6. TDMS Logger Front Panel



2. Test the TDMS Logger VI.

   ☐ Enter your name in the **Test Operator** field.

   ☐ Enter `A001`, `A002`, and `A003` in the **UUT Serial Numbers** control.

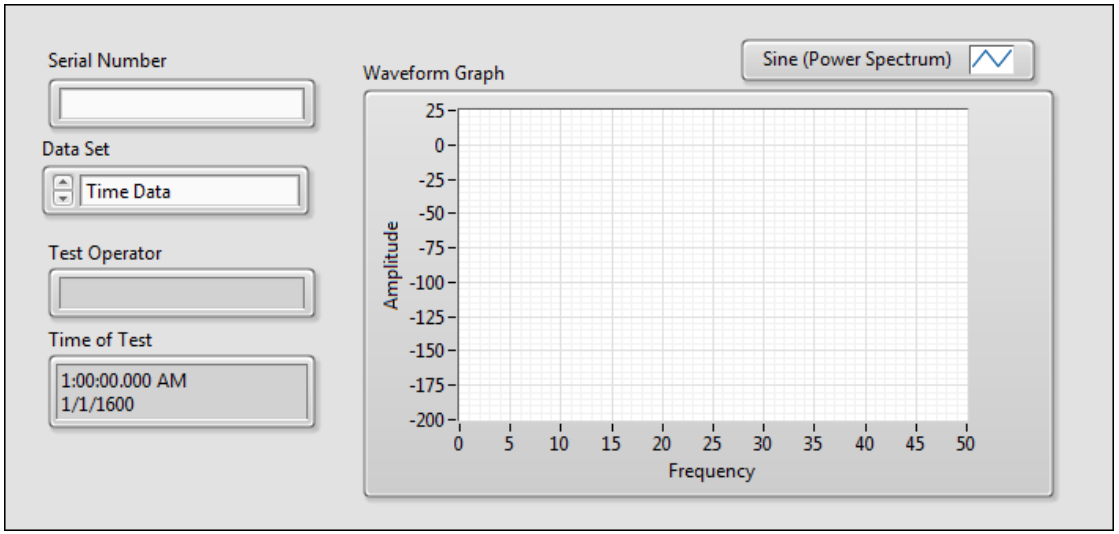
   ☐ Run the TDMS Logger VI and save the file as `test.tdms` in the `<Exercises>\LabVIEW Core 2\TDMS Logger and Reader` directory. When you save the TDMS file, the TDMS File Viewer window opens.

- ☐ View the logged data in the TDMS File Viewer window.
- ☐ Expand the **test.tdms** item in the File contents pane of the window and then expand the **Time Data** and **Power Spectrum** items and view the sample data for each of the UUT serial numbers you entered.
- ☐ Click **Quit** to close the TDMS File Viewer window.
- ☐ View the front panel of the TDMS Logger VI, which also displays a plot for each serial number you enter.
- ☐ Examine the block diagram.
- ☐ Close the TDMS Logger VI. Do not save any changes.

Implementation—TDMS Reader VI

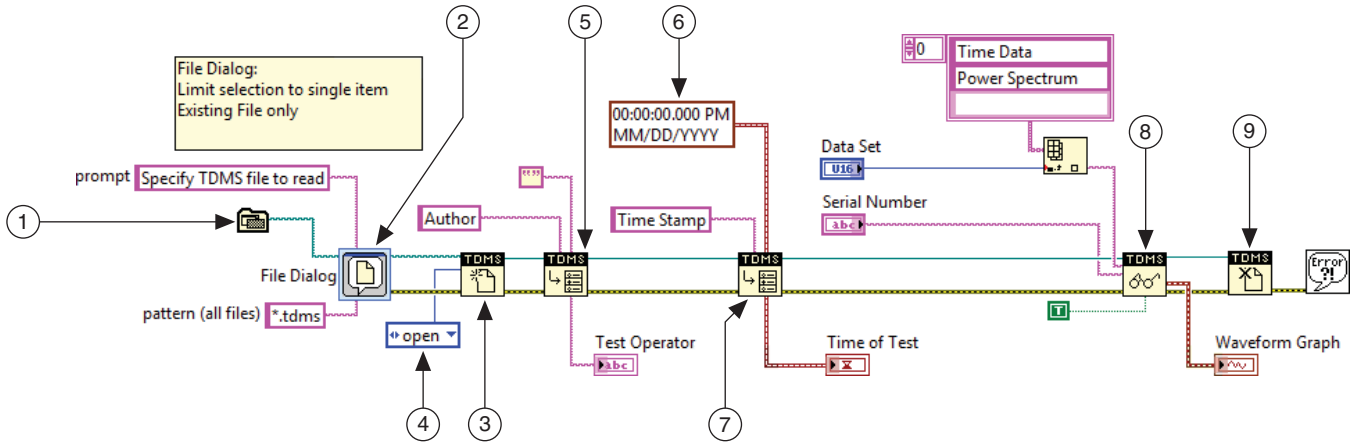
1. Open the TDMS Reader VI from the Project Explorer window of the TDMS Logger and Reader project.
2. The front panel is shown in Figure 5-7.

Figure 5-7. TDMS Reader Front Panel



3. Open and examine the block diagram as shown in Figure 5-8

Figure 5-8. TDMS Reader Block Diagram



1 Application Directory VI—Returns the path to the directory containing the application.
2 File Dialog Express VI—This VI is configured with the following options:
   - ☐ Limit selection to single item
   - ☐ File—The user can only select a file
   - ☐ Existing—The user can only select an existing file or folder
3 TDMS Open—Open a .tdms file for reading or writing.
4 Open constant—Right-click the operation input of the TDMS Open function and select **Create»Constant**.
5 TDMS Get Properties—This instance of the TDMS Get Properties function reads the Author file property.
6 Time Stamp Constant—Passes time and date values to the block diagram.
7 TDMS Get Properties—This instance of the TDMS Get Properties function reads the Time Stamp file property.
8 TDMS Read—Reads the time data or power spectrum data for the specified UUT and displays the data on the Waveform Graph.
9 TDMS Close—Closes the .tdms file you opened with the TDMS Open function. This function closes the .tdms file regardless of whether an error occurred in a preceding operation.

Test

1. Read and display the time domain data.

   ☐ On the VI front panel, ensure that the Data Set is set to **Time Data**.

   ☐ Enter `A001` in the **Serial Number** control.

Note `A001` was one of the serial numbers you entered when you ran the TDMS Logger VI.

- ☐ Run the VI and select the `test.tdms` file you saved in `<Exercises>\LabVIEW Core 2\TDMS Logger and Reader`. The following information is displayed on the front panel:
  - Waveform Graph—sine wave
  - Test Operator—your name
  - Time of Test—time stamp from when you ran the TDMS Reader
- ☐ Change the Serial Number to `A002`.
- ☐ Run the VI. A different sine wave should display.

2. Read and display the power spectrum data.
   - ☐ Change the **Data Set** control to **Power Spectrum**.
   - ☐ Run the VI and select your TDMS file. Power spectrum data should display in the Waveform Graph.
3. Close the VI.

TDM Excel Add-In (Optional)

Note This exercise requires Microsoft Excel version 2003 or 2007 and the TDM Add-in Tool available on `ni.com`.

1. Launch Microsoft Excel.
2. Click the **Add-Ins** tab and select the **TDM Importer: Import a TDM(S) File** icon in the **Custom Toolbars** section.
3. Browse to find the TDMS file you created earlier: `<Exercises>\LabVIEW Core 2\TDMS Logger and Reader\test.tdms`. The property information is displayed on the first worksheet. The Time data and Power Spectrum data are displayed on separate worksheets.
4. Exit Microsoft Excel and return to LabVIEW.

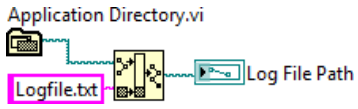
End of Exercise 5-3

Additional Resources

Learn More About	LabVIEW Help Topic
File formats	*Determining Which File Format to Use*
Creating file and folder paths	*Building Relative Paths*
Using TDMS Files	*When To Use TDMS Files* *Standard versus Advanced TDMS Functions*

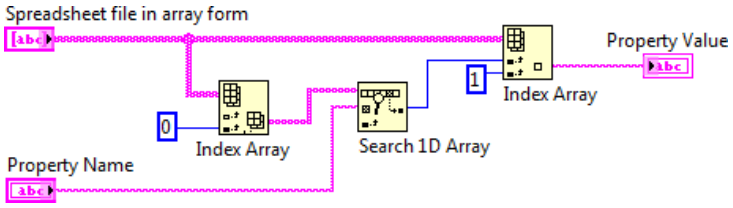
Activity 5-2: Lesson Review

1. Consider the code shown below. The resulting Log File Path contains a text file path in which folder?



   a. Same folder as the VI that executed the code
   b. Same folder as the LabVIEW Project
   c. Current user's AppData Directory
   d. Unknown

2. In the following example, what index value is returned from the Search 1D Array function if Property Name is not found in the input array?



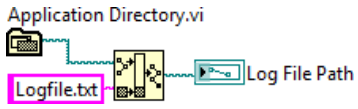
   a. NaN (Not a Number)
   b. 0
   c. -1
   d. Negative Infinity

3. You need to store data that other engineers will later analyze with Microsoft Excel. Which file storage format(s) should you use?
   a. Tab-delimited ASCII
   b. Custom binary format
   c. TDMS

4. TDMS files store properties at which of the following levels?
   a. File
   b. Channel Group
   c. Channel
   d. Value

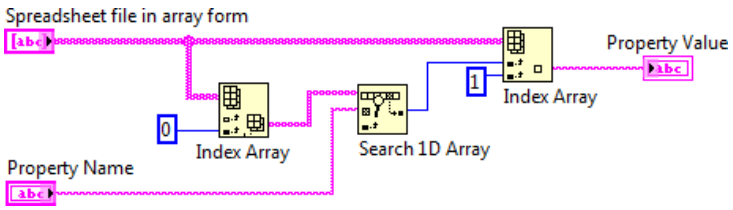
Activity 5-2: Lesson Review - Answers

1. Consider the code shown below. The resulting Log File Path contains a text file path in which folder?



   a. Same folder as the VI that executed the code
   b. Same folder as the LabVIEW Project
   c. Current user's AppData Directory
   d. Unknown

2. In the following example, what index value is returned from the Search 1D Array function if Property Name is not found in the input array?



   a. NaN (Not a Number)
   b. 0
   c. -1
   d. Negative Infinity

3. You need to store data which other engineers will later analyze with Microsoft Excel. Which file storage format(s) should you use?
   a. Tab-delimited ASCII
   b. Custom binary format
   c. TDMS

4. TDMS files store properties at which of the following levels?
   a. File
   b. Channel Group
   c. Channel
   d. Value

6 Improving an Existing VI

In this lesson you will learn methods to refactor inherited code and experiment with typical issues that appear in inherited code.

Topics

+ Refactoring Inherited Code
+ Typical Refactoring Issues

Exercises

Exercise 6-1 Refactoring a VI

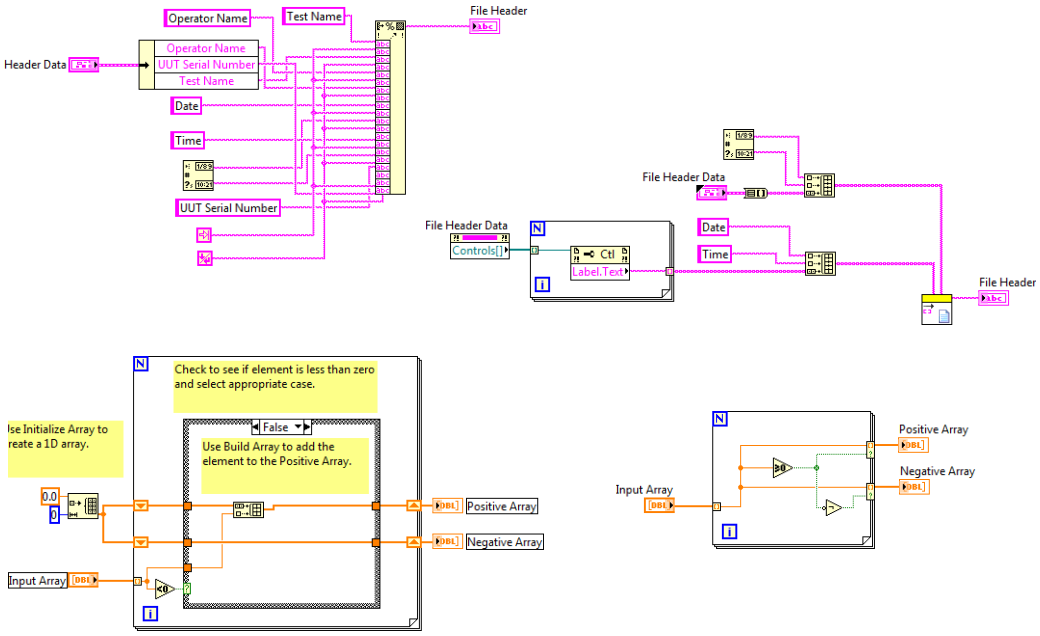
A. Refactoring Inherited Code

Objective: Explain the refactoring process and identify VIs that would benefit from refactoring.

Definition

Refactoring The process of redesigning software to make it more readable and maintainable so that the cost of change does not increase over time.

Refactoring changes the internal structure of a VI to make it more readable and maintainable, without changing its observable behavior.



When to Refactor

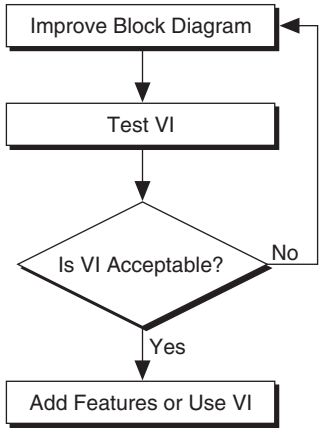
Consider refactoring when you are adding a feature to a VI or debugging it. Refactoring is usually beneficial. However, there is value in a VI that functions, even if the block diagram is not readable.

Good candidates for complete rewrites include the following types of VIs:

- VIs that do not function
- VIs that satisfy only a small portion of your needs

Refactoring Process

When you refactor to improve the block diagram, make small cosmetic changes before tackling larger issues. Cosmetic changes to the block diagram can be useful. For example, it is easier to find duplicated code if the block diagram is well-organized and the terminals are well-labeled.



B. Typical Refactoring Issues

Objective: Recognize common code issues that may require you to refactor your code.

Poor Naming

Inherited VIs often contain controls and indicators that do not have meaningful names.

Poor

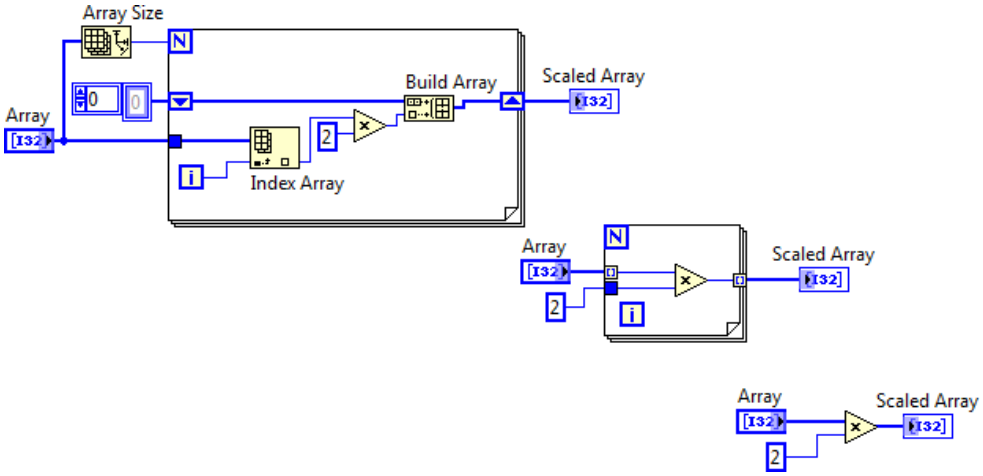
Better

Best

By renaming controls and VIs and creating meaningful VI icons, you can improve the readability of an inherited VI.

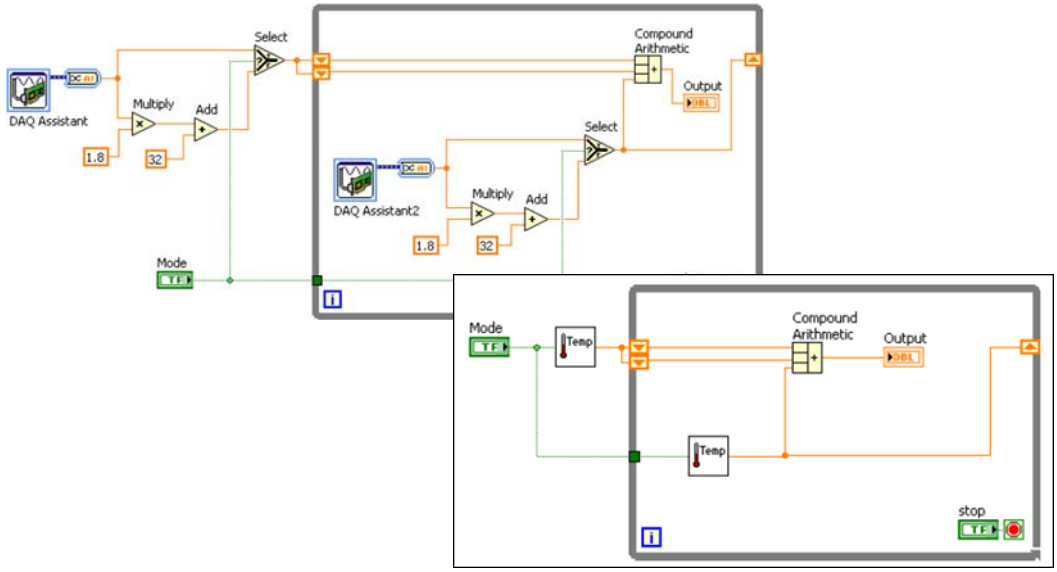
Overly Complicated

The following figure contains three versions of the same task. The top block diagram is overly complicated and contains unnecessary logic. The other two block diagrams simplify the task. Each section of code performs the same functionality. However, the code at the lower right is much easier to read and maintain.



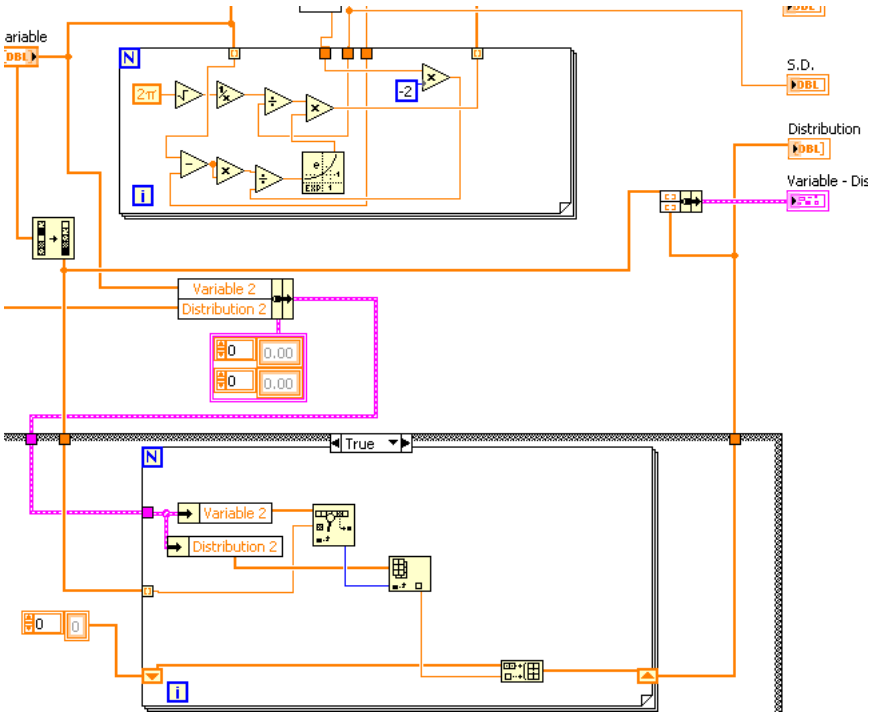
Duplicate Logic

If a VI contains duplicated logic, you always should refactor the VI by creating a subVI for the duplicated logic. This can improve the readability and testability of the VI. The example in the following figure shows how you can create a subVI from sections of code that are reused in two places on the block diagram.



Disorganization

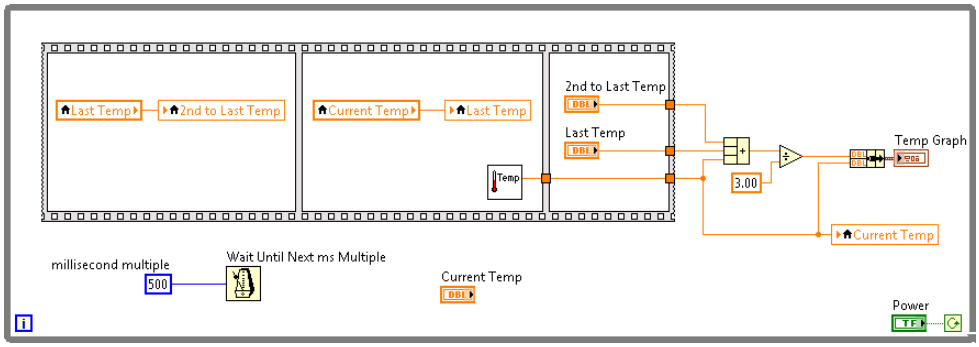
Diagrams that require scrolling in both directions are often too large and can benefit from refactoring. Improve the readability of a disorganized VI by relocating objects within the block diagram. You also can create subVIs for sections of the VI that are disorganized. Place comments on areas of a VI that are disorganized to improve the readability of the VI.



Tip You can use the Clean Up Diagram tool to help organize the block diagram, but it will not address all issues. For example, the Clean Up Diagram tool will not help remove deeply nested structures.

Breaks Dataflow

The block diagram does not use dataflow programming.



Solution

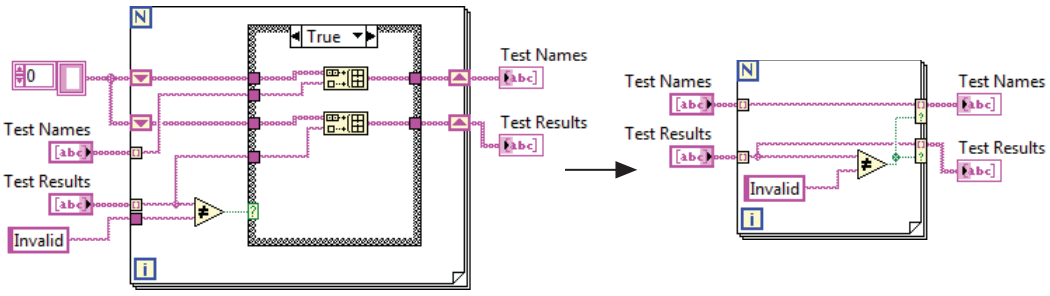
- Replace Sequence structures with state machines.
- Delete local variables and wire directly to controls or indicators.

Outdated Practices

The VI was created in an earlier version of LabVIEW and has outdated practices.

Solutions

- Replace polling-based design with event-based design.
- Use new features that simplify code.



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

Job Aid—Refactoring Checklist

Use the following refactoring checklist to help determine if you should refactor a VI. If you answer yes to any of the items in the checklist, refer to the guidelines in the *When to Refactor* section of this lesson to refactor the VI.

- ☐ Disorganized block diagram
- ☐ Overly large block diagram
- ☐ Poorly named objects and poorly designed icons
- ☐ Unnecessary logic
- ☐ Duplicated logic
- ☐ Lack of dataflow programming
- ☐ Complicated algorithms

Additional Resources

Learn More About	LabVIEW Help Topic
Replacing polling with events	*Events in LabVIEW*
Reducing duplicate logic	*Creating SubVIs from Selections*
Creating quality code	*LabVIEW Style Checklist*

7 Creating and Distributing Applications

In this lesson you will learn how to make necessary code modifications to select and build the appropriate deployment option for a LabVIEW application.

Topics

+ Preparing the Files
+ Build Specifications
+ Create and Debug an Application
+ Create an Installer

Exercises

Exercise 7-1 Preparing Files for Distribution
Exercise 7-2 Create and Debug a Stand-Alone Application
Exercise 7-3 Create an Installer

A. Preparing the Files

Objective: Identify tasks to complete when preparing project files for distribution.

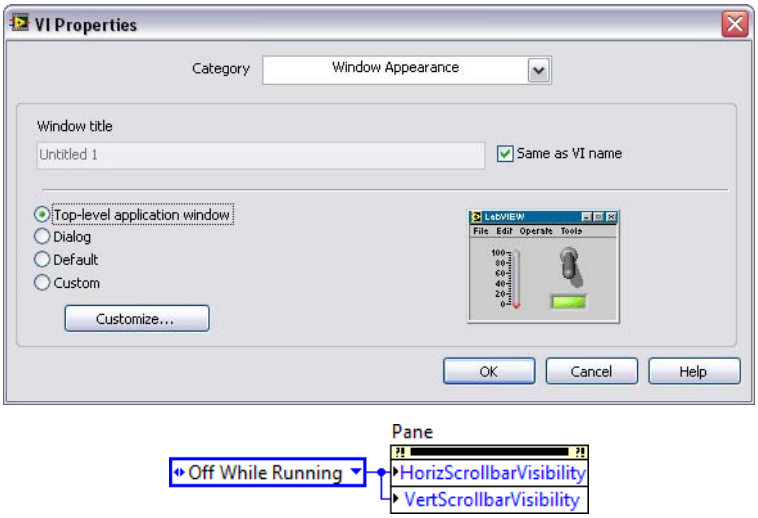
First, prepare your files.

Before you can create a stand-alone application with your VIs, you must first prepare your files for distribution.

1. Recompile and save changes to VIs.
2. Verify desired VI Properties settings.
3. Ensure paths generate correctly.
4. Conditionally call the Quit LabVIEW function.

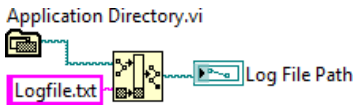
VI Properties

Use the VI Properties dialog box to customize the window appearance and size. You might want to configure a VI to hide scroll bars, or you might want to hide the buttons on the toolbar.



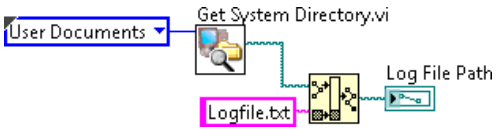
Paths

Consider the path names you use in the VI. Assume you read data from a file during the application, and the path to the file is hard-coded on the block diagram. Once an application is built, the file is embedded in the executable, changing the path of the file. Being aware of these issues will help you to build more robust applications in the future.



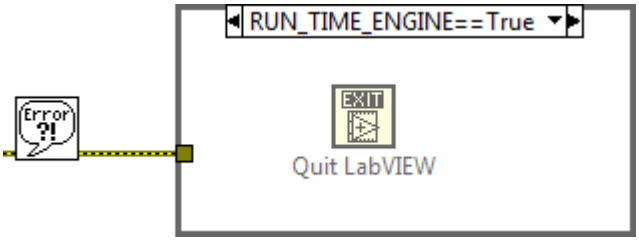
If the VI is called:	The VI returns:
From a stand-alone application.	The path to the folder containing the application executable.
From a LabVIEW project (.lvproj) in the LabVIEW Development Environment.	The path to the folder.

System Paths



Quit Application

In a stand-alone application, the top-level VI must quit LabVIEW or close the front panel when it is finished executing. To completely quit and close the top-level VI, you must call the Quit LabVIEW function on the block diagram of the top-level VI.



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.

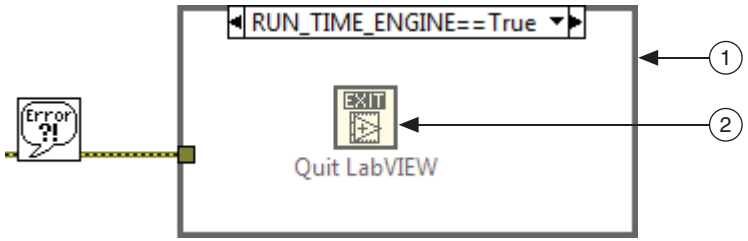
   ☐ In the Project Explorer window, double-click **Histogram Main.vi** to open the VI.

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

Job Aid: Building Applications Checklist—Preparing Files

☐ Prepare VIs for build specifications.

Before you create or edit a build specification, recompile and save changes to VIs in memory to ensure that the build is accurate.

☐ Edit settings in the VI Properties dialog box.

Make sure the settings in the VI Properties dialog box are correct. For example, you might want to configure a VI to hide scroll bars, or you might want to hide the buttons on the toolbar.

☐ Eliminate duplicate filenames.

If you build a stand-alone application, shared library, or Web service using the LabVIEW 8.x file layout or you build a source distribution with the default settings, files with the same name cannot go to the same destination. If you have two files with the same name, even if they belong to different libraries, rename the files on the Source File Settings page when you configure the build or move them to their own destinations. If you do not rename the files, LabVIEW moves the files to different folders for you to avoid a filename collision.

☐ Resolve all conflicts in the project.

The Application Builder does not allow building an application if the project contains conflicts. A conflict in the LabVIEW Project is a potential cross-link that occurs when LabVIEW tries to load a VI that has the same qualified name as an item already in the project. To correct a cross-linking conflict, select **Project»Resolve Conflicts**.

☐ Verify destination path length is within the limit set by the operating system.

To avoid receiving an error during the build process, ensure that the file paths for the destination directory, including the filename, generated for the build are less than 255 characters. You can specify the destination for the generated files on the Destinations page of the properties dialog box for the build specification you are creating.

☐ Ensure paths generate correctly.

If a VI loads other VIs dynamically using VI Server or calls a dynamically loaded VI through a Call By Reference node, make sure the application or source distribution creates the paths for the VIs correctly. To ensure paths generate correctly, use relative paths to load the VIs. The following table depicts the relative paths for a top-level VI, `foo.vi`, which calls `a.vi` and `b.vi`. `C:\..\Application.exe` represents the path to the application.

Path to source files	Path to files in application
`C:\Source\foo.vi`	`C:\..\Application.exe\foo.vi`
`C:\Source\xxx\a.vi`	`C:\..\Application.exe\xxx\a.vi`
`C:\Source\yyy\b.vi`	`C:\..\Application.exe\yyy\b.vi`

☐ Ensure functions work as expected.

For example, if a VI uses the Current VI's Path function, make sure the function works as expected in the application or shared library. In an application or shared library, the Current VI's Path function returns the path to the VI in the application file and treats the application file as an LLB. For example, if you build foo.vi into an application, the function returns a path of `C:\..\Application.exe\foo.vi`, where `C:\..\Application.exe` represents the path to the application and its filename.

B. Build Specifications

Objective: Recognize build specification settings and determine when a build specification is necessary.

Build Specifications

Build specifications contain all the settings for the build:

- Files to include
- Directories to create
- Settings for VIs

Build specification File that contains all the settings for the build, such as files to include, directories to create, and settings for VIs.

Why Use Build Specifications?

Deliverable	Description
Stand-alone applications	Use stand-alone applications to provide other users with executable versions of VIs. **(Windows)** Applications have a `.exe` extension. **(Mac OS)** Applications have a `.app` extension.
Installers	**(Windows)** Use installers to distribute stand-alone applications, shared libraries, and source distributions that you create with the Application Builder.
.NET Interop Assemblies	**(Windows)** Use .NET interop assemblies to package VIs for the Microsoft .NET Framework. You must install the Microsoft .NET Framework 2.0 or higher to build a .NET interop assembly using the Application Builder.
Packed project libraries	Use packed project libraries to package multiple LabVIEW files into a single file. Packed libraries have a `.lvlibp` extension.
Shared libraries	Use shared libraries if you want to call VIs using text-based programming languages, such as LabWindows/CVI, Microsoft Visual C++, and Microsoft Visual Basic. Using shared libraries provides a way for programming languages other than LabVIEW to access code developed with LabVIEW.
Source distributions	Use source distributions to package a collection of source files. You can configure settings for specified VIs to add passwords, remove block diagrams, or apply other settings. You also can select different destination directories for VIs in a source distribution without breaking the links between VIs and subVIs.
Zip files	Use zip files when you want to distribute files or an entire LabVIEW project as a single, portable file. A zip file contains compressed files, which you can send to users. Zip files are useful if you want to distribute selected source files to other LabVIEW users. You also can use the Zip VIs to create zip files programmatically.

C. Create and Debug an Application

Objective: Create and debug a stand-alone LabVIEW application.

System Requirements for Applications

Applications that you create with Build Specifications generally have the same system requirements as the LabVIEW development system. Memory requirements vary depending on the size of the application created.

You can distribute these files without the LabVIEW development system. However, to run stand-alone applications and shared libraries, users must have the LabVIEW Run-Time Engine installed.

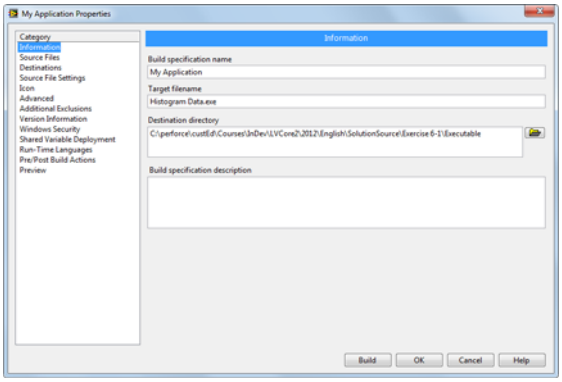
Configure the EXE Build Specification

In the Application Properties dialog box:

1. Specify the name of your executable.
2. Specify the known destination for generated executable files.
3. Identify a startup VI and include any dynamically linked files.
4. Enable debugging, if desired.
5. Preview the build.
6. Save the project and build the application or shared library.
7. Verify the build.

EXE Properties—Information

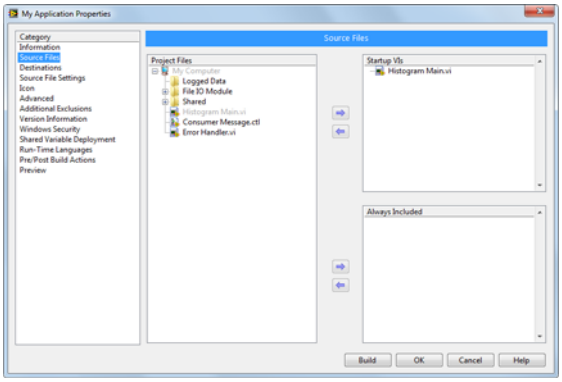
Use this category to name the stand-alone application and select the location to build the application. To avoid having several built files taking up space on the development machine, send all builds to a folder that is known to be cleaned up on a regular basis



EXE Properties—Source Files

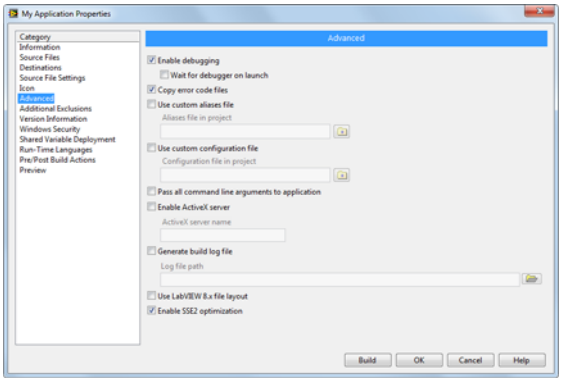
Use this category to add and remove files and folders from the stand-alone application and to specify the startup VIs to use in the build.

- Always include a Startup VI.
- Do not include statically linked files.
- Include any dynamically linked files in **Always Included** section.



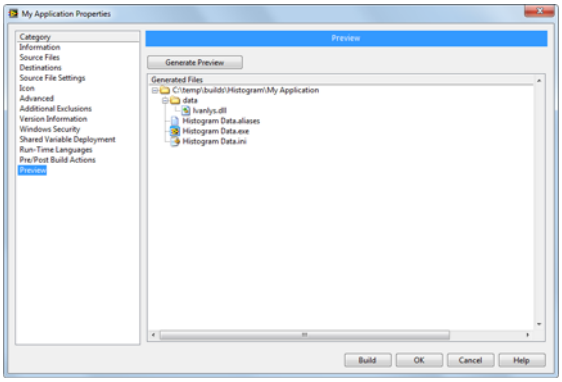
EXE Properties—Advanced

Use this category to enable and disable debugging. Enable debugging to identify run-time problems in deployed EXEs. However, enabling debugging increases the EXE size.



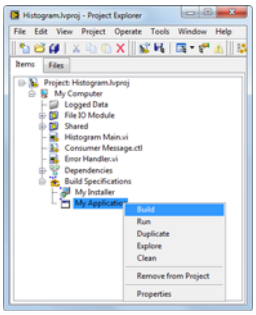
EXE Properties—Preview

Use this category to generate a preview of the build. Preview the build to see the directory structure and files that generate. Save the project before building.



Save the Project and Build the Executable

1. Click the Save All (This Project) button to save files all files associated with the project, including build specifications.

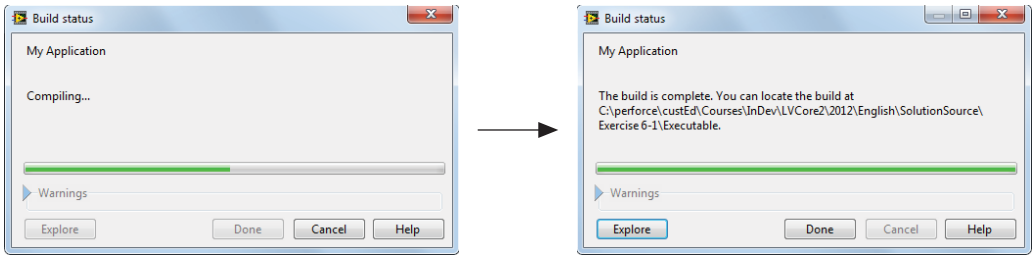


2. Right-click the build specification and select Build to create the executable.

Tip After building, you can also run the application through the shortcut menu.

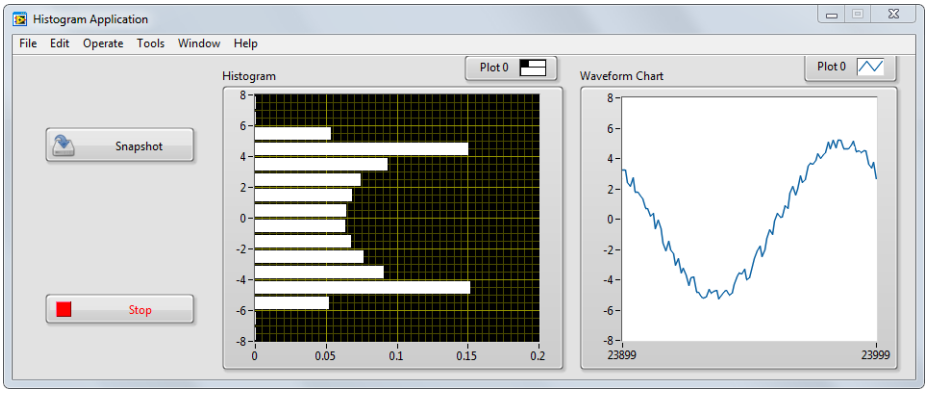
Build Status

1. Click the **Explore** button to open the directory where the new EXE is located.
2. Click the **Done** button to close the Build Status dialog box.



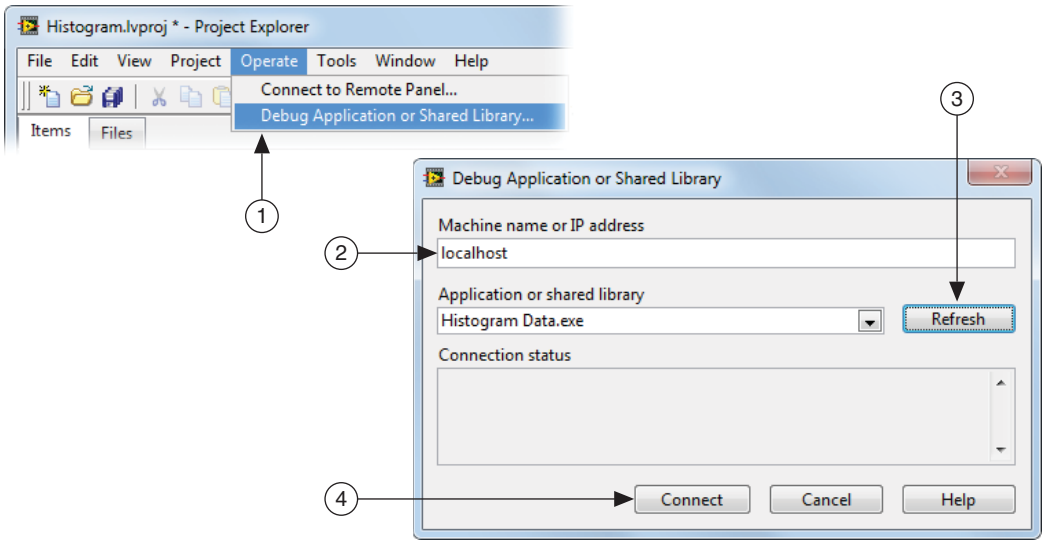
Run EXE and Verify Execution

Run the EXE by double-clicking .exe file that was created.



Debug EXE - LabVIEW on Same Machine

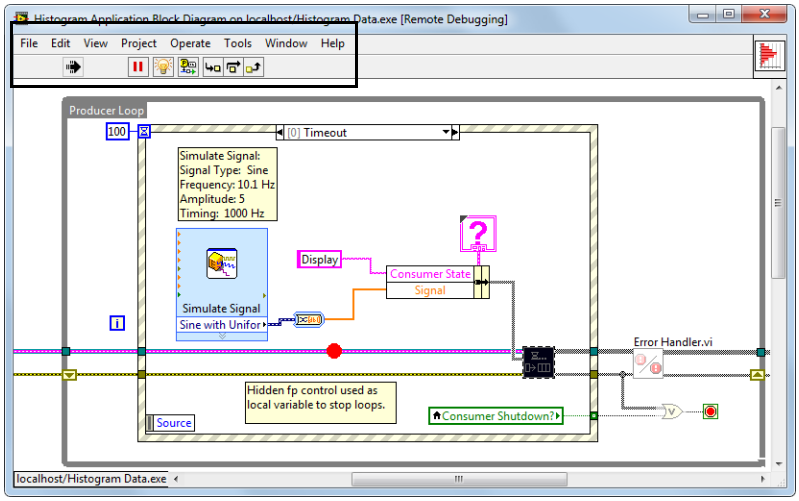
If you have enabled debugging in the executable, you can start a debugging session.



1. Select **Operate»Debug Application or Shared Library**.
2. Enter the Machine name or IP address of the destination computer. If LabVIEW is running on the same machine as the executable, enter `localhost` in the text box.
3. Click the **Refresh** button if you do not see the EXE you want in the pull-down menu.
4. Click the **Connect** button to start debugging.

Debug the EXE from LabVIEW

Use LabVIEW debugging tools to debug the application.



Why might an executable behave differently?

Application behavior can change when distributing to a different system for a number of reasons.

- File paths may change
- Missing drivers or support files
- System resources like memory or CPU speed may differ
- The application INI file differs from the LabVIEW INI file. The .ini file allows LabVIEW to store system settings. Different system settings can lead to changes in VI Server access.
- Not all features are supported in the Run-Time Engine

Job Aid: Building Applications Checklist

☐ Add dynamic VIs to the build.

If a VI loads other VIs dynamically using the VI Server or calls a dynamically loaded VI through a Call By Reference or Start Asynchronous Call node, you must add those VIs to the Always Included listbox on the Source Files page of the Application Properties, Shared Library Properties, Source Distribution Properties, or Web Service Properties dialog box. Make sure the application, shared library, or source distribution creates the paths for the VIs correctly.

☐ Include the LabVIEW Run-Time Engine.

Include the LabVIEW Run-Time Engine if you want to distribute a stand-alone application or shared library. (Windows) If you use the Application Builder to build an installer, you can add the LabVIEW Run-Time Engine to the installer build. From the Additional Installers page of the Installer Properties dialog box, place a checkmark in the NI LabVIEW Run-Time Engine checkbox in the National Instruments Installers to Include listbox. (Mac OS X and Linux) You can build a source distribution to distribute your stand-alone application or shared library with the LabVIEW Run-Time Engine.

☐ Determine whether the hardware target supports SSE2 instructions.

By default, the Application Builder configures new build specifications to include SSE2 compiler optimizations that improve the run-time performance of distributed VIs and built applications. To build a stand-alone application, .NET interop assembly, packed project library, shared library, source distribution, or Web service that runs on a hardware target that does not support SSE2 instructions, you must disable the SSE2 compiler optimizations before building the application. Otherwise, LabVIEW returns an error when the application runs on the target.

☐ Consider the need for additional drivers and installers.

If you build an installer for your application, you may want to consider including additional National Instruments drivers or third-party installers. For example, if you use DAQmx Express VIs, include the NI-DAQmx driver.

☐ Consider language requirements for the end user.

By default, stand-alone application and shared library build specifications provide support for Chinese (Simplified), English, French, German, Japanese, and Korean. You can view these settings on the Run-Time Languages page of the properties dialog box.

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

D. Create an Installer

Objective: Create, configure, build, and deploy a LabVIEW application installer.

Why Create an Installer?

Use the space below to list some reasons that you might want to create an installer for your LabVIEW application:

_____

_____

_____

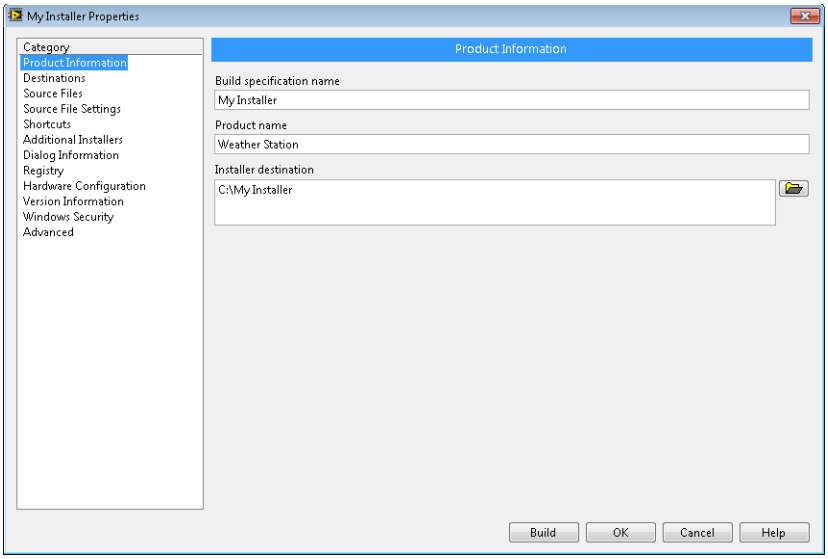
_____

Configuring Installer Build Specifications

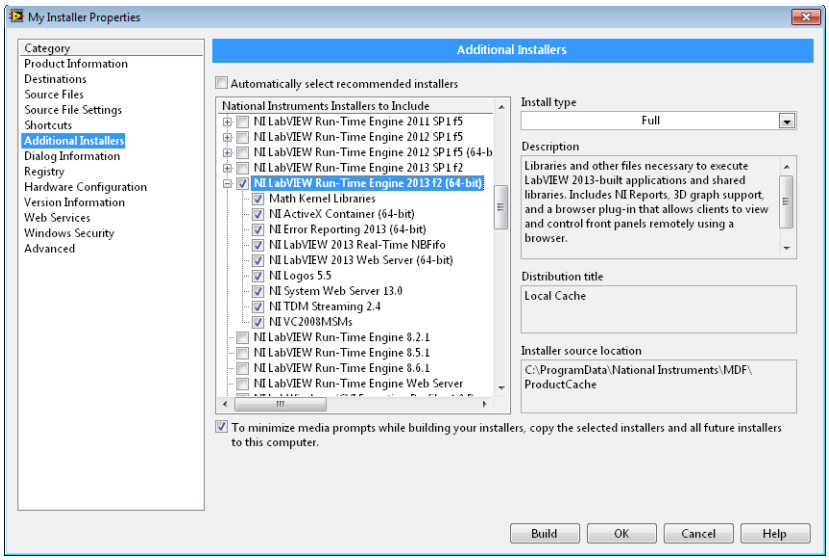
In the Installer Properties dialog box:

- Include the LabVIEW Run-Time Engine.
- Include drivers used in the application.

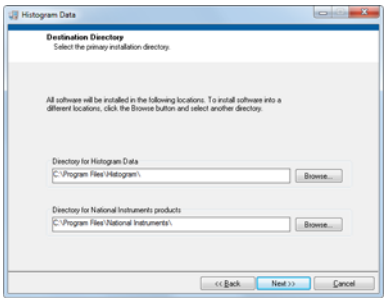
Installer Properties—Product Information

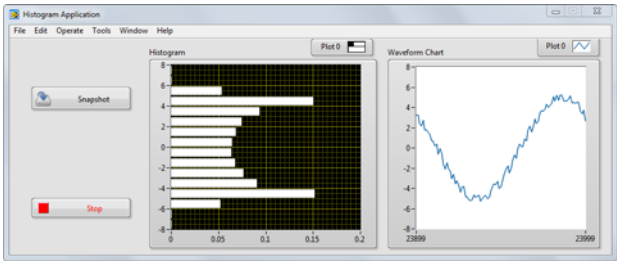


Installer Properties—Additional Installers

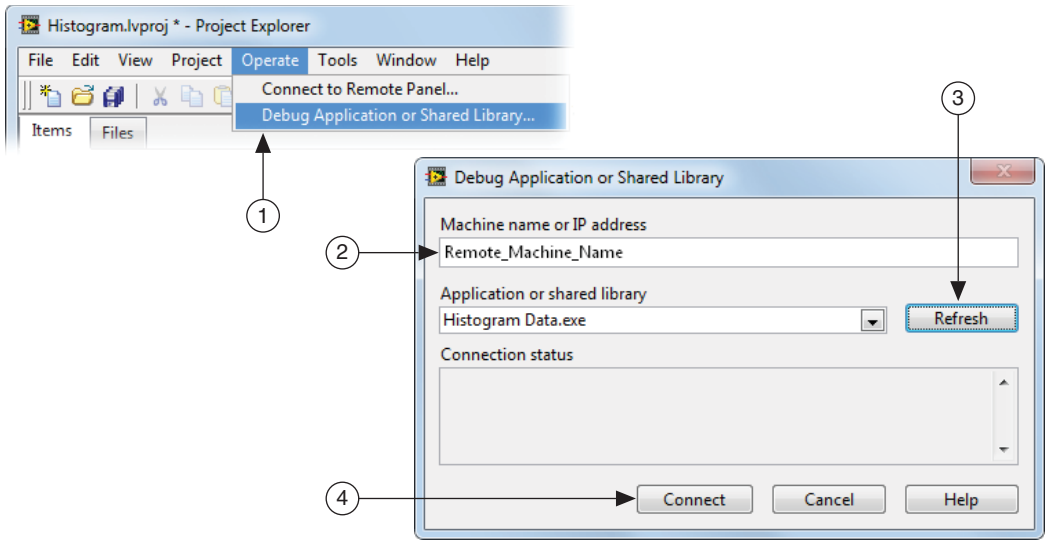


Install and Run on Destination Computer





Debug Executable on Destination Computer



1 Select **Operate»Debug Application or Shared Library**.
2 Enter the Machine name or IP address of the destination computer.
Note: The machine name (alias) might not work depending on whether the right ports are open in the firewalls for both machines.
3 Click the **Refresh** button if you do not see the EXE you want in the pull-down menu.
4 Click the **Connect** button to start debugging.

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.

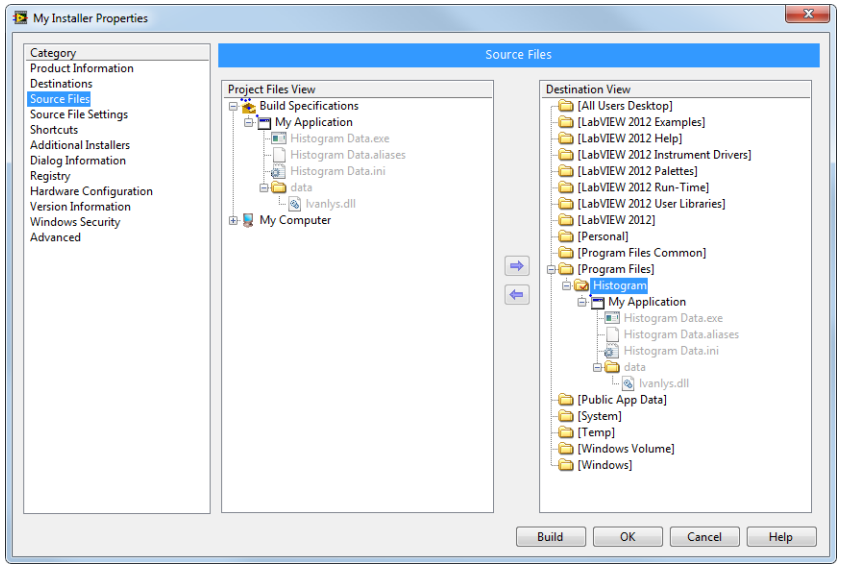
Creating an Installer Build Specification

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

Summary

- The Application Builder enables you to create stand-alone applications and installers.
- Test your application and installers regularly as you develop.
- Creating a professional, stand-alone application with your VIs involves understanding the following:
  - The architecture of your application
  - The programming issues particular to the application
  - The application building process
  - The installer building process

Job Aid: Building Applications Checklist—Distributing Builds

☐ Disable debugging for stand-alone applications and shared libraries.

Before you distribute, disable debugging for stand-alone applications and shared libraries. Allowing debugging can increase file size and slow performance at runtime.

Note If you plan to test the application before your final build, enable debugging by placing a checkmark in the Enable Debugging checkbox on the Advanced page of the properties dialog box.

☐ Create an About dialog box for a stand-alone application.

Consider creating an About dialog box to display general information about an application.

☐ Distribute a configuration file.

Consider distributing a configuration file, also called a preference file, that contains LabVIEW work environment settings with an application, such as the labview.ini file on Windows.

Note If you use the default configuration file, an error might occur if a user is running a separate application. When a user runs a separate application, in particular the LabVIEW development environment, and tries to run the stand-alone application at the same time, an error might occur if the VI Server and the application share the same port. If a user tries to run VI Server at the same time as the application, VI Server does not run and the user receives no warning that the port is already in use. Place a checkmark in the Use custom configuration file checkbox on the Advanced page of the Application Properties dialog box, to select a custom configuration file in the project.

☐ Distribute documentation.

Consider distributing documentation with an application or shared library so users have the information they need to use the application or shared library.

Note Do not distribute the LabVIEW product documentation. The LabVIEW product documentation is copyrighted material.

☐ Distribute legal information.

Review `<National Instruments>\_Legal Information.txt` for information on including legal information in installers built with NI products.

☐ For stand-alone applications and shared libraries that use shared variables, do not include the .lvlib file.

If you plan to distribute a stand-alone application or shared library that uses shared variables, do not include the .lvlib file in an LLB, the executable, or the shared library. Use the Source Files Setting page of the Application Properties dialog box or the Source Files Setting page of the Shared Library Properties dialog box to change the destination of the .lvlib file to a destination outside the LLB, executable, or shared library.

Additional Resources

Learn More About	LabVIEW Help Topic
Creating application for distribution	*Building and Distributing Applications* *Developing and Distributing an Application*
Creating installers	*Caveats and Recommendations for Building Installers*

Activity 7-1: Lesson Review

1. You need to use LabVIEW build specifications to distribute which of the following items?
   a. Installers
   b. ZIP files
   c. VIs
   d. Executables

2. Mark the following statements True or False.

Applications that you create with Build Specifications generally have the same system requirements as the LabVIEW development system used to create the VI or application.	True / False
You create executables to make sure the application behaves the same when distributed to different systems.	True / False
Memory requirements vary depending on the size of the application created.	True / False
You should always include the LabVIEW Run-Time Engine in the application build specification.	True / False

Activity 7-1: Lesson Review - Answers

1. You need to use LabVIEW build specifications to distribute which of the following items?
   - **a. Installers**
   - **b. ZIP files**
   - c. VIs
   - **d. Executables**

2. Mark the following statements True or False.

Applications that you create with Build Specifications generally have the same system requirements as the LabVIEW development system used to create the VI or application.	**True** / False
You create executables to make sure the application behaves the same when distributed to different systems.	True / **False**
Memory requirements vary depending on the size of the application created.	**True** / False
You should always include the LabVIEW Run-Time Engine in the application build specification.	True / **False**

A Setting Up Your Hardware

This appendix contains instructions on setting up your hardware.

Note Complete this setup only if you have hardware installed and have not already completed hardware setup on your system as part of a *LabVIEW Core 1* course exercise.

1. Locate the BNC-2120 hardware and visually confirm that it is connected to the DAQ device in your computer.
2. Using a BNC cable, connect the Analog In Channel 1 to the Sine Function Generator, as shown in Figure A-1.
3. Set the **Frequency Selection** switch to its middle level and the **Frequency Adjust** knob to its lowest level.

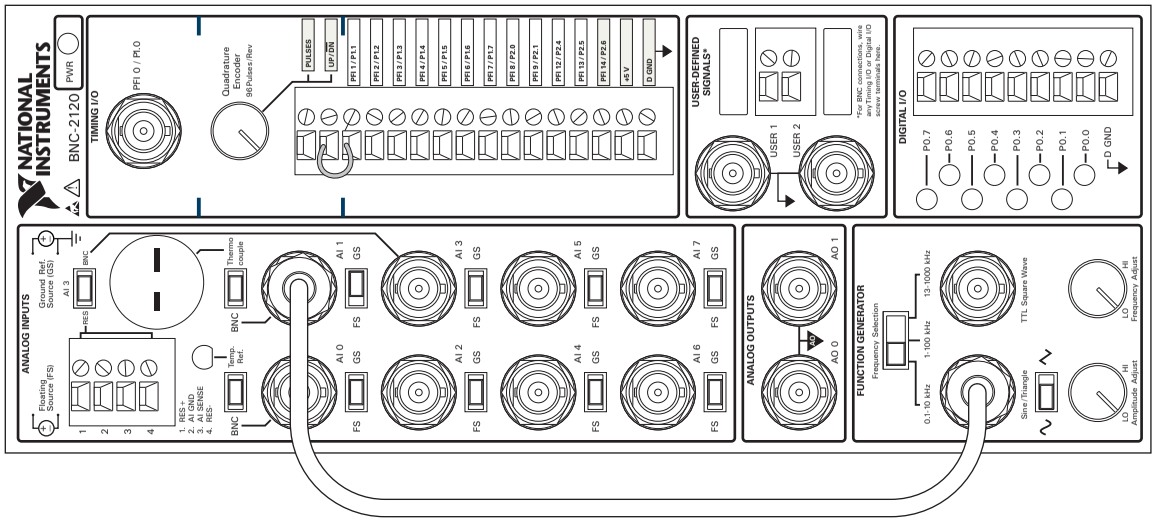
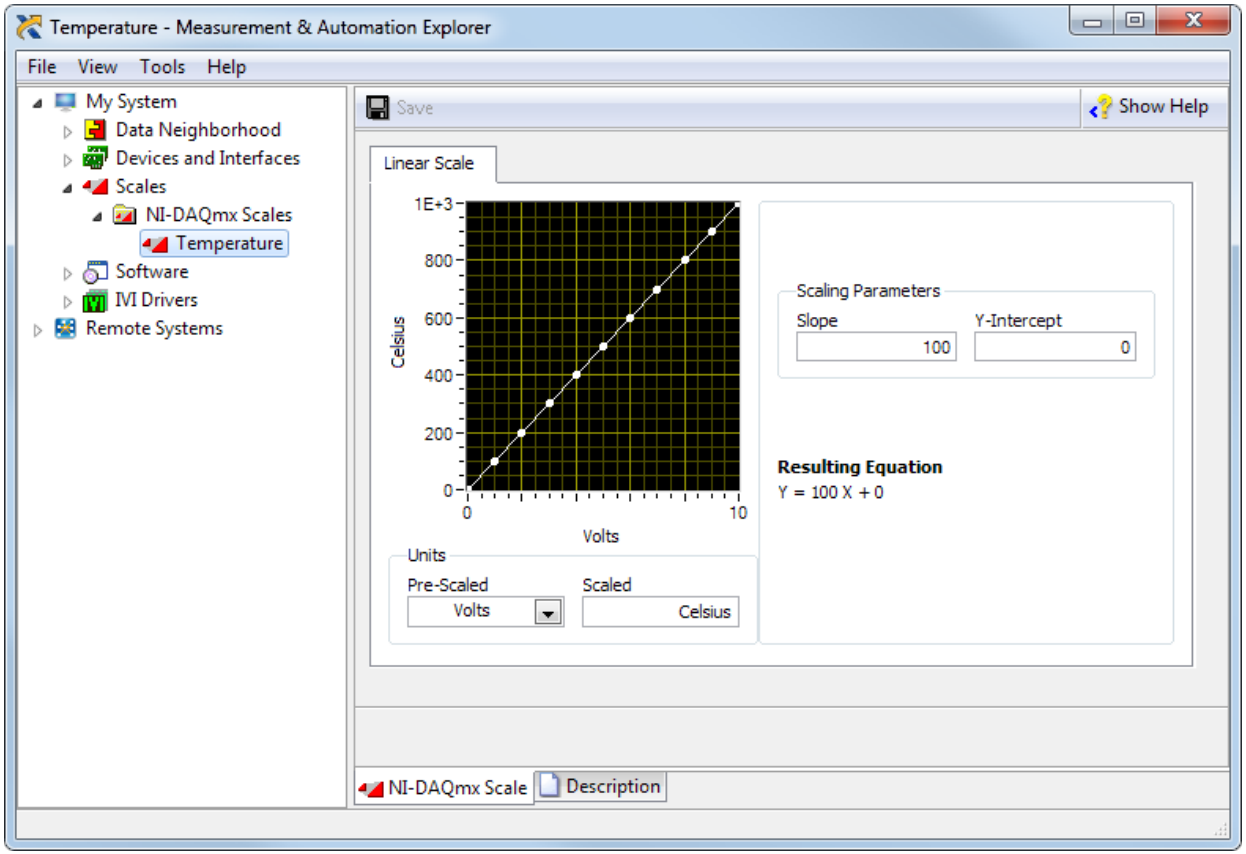


Figure A-1. Connections for the BNC-2120

4. Launch MAX by selecting **Start»Programs»NI MAX** or by double-clicking the NI MAX icon on your desktop. The NI Measurement & Automation Explorer (MAX) searches the computer for installed National Instruments hardware and displays the information.

5. Create a custom scale for the temperature sensor on the DAQ Signal Accessory. The sensor conversion is linear and uses the following the formula `Voltage x 100 = Celsius`.

Figure A-2. Temperature Scale



☐ Right-click the **Scales** section and select **Create New** from the shortcut menu.

☐ Select **NI-DAQmx Scale**.

☐ Click **Next**.

☐ Select **Linear**.

☐ Name the scale `Temperature`.

☐ Click **Finish**.

☐ Change the Scaling Parameter **Slope** to `100`.

☐ Under **Units**, enter `Celsius` in the **Scaled** field.

☐ Click the **Save** button on the toolbar to save the scale.

☐ Close MAX by selecting **File»Exit**.

B Additional Information and Resources

National Instruments provides global services and support as part of our commitment to your success. Take advantage of product services in addition to training and certification programs that meet your needs during each phase of the application life cycle; from planning and development through deployment and ongoing maintenance.

A. NI Services

To get started, register your product at ni.com/myproducts.

As a registered NI product user, you are entitled to the following benefits:

- Access to applicable product services.
- Easier product management with an online account.
- Receive critical part notifications, software updates, and service expirations.

Log in to your National Instruments ni.com User Profile to get personalized access to your services.

B. Services and Resources

- **Maintenance and Hardware Services**—NI helps you identify your systems' accuracy and reliability requirements and provides warranty, sparing, and calibration services to help you maintain accuracy and minimize downtime over the life of your system. Visit ni.com/services for more information.
  - **Warranty and Repair**—All NI hardware features a one-year standard warranty that is extendable up to five years. NI offers repair services performed in a timely manner by highly trained factory technicians using only original parts at a National Instruments service center.
  - **Calibration**—Through regular calibration, you can quantify and improve the measurement performance of an instrument. NI provides state-of-the-art calibration services. If your product supports calibration, you can obtain the calibration certificate for your product at ni.com/calibration.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.
- **Training and Certification**—The NI training and certification program is the most effective way to increase application development proficiency and productivity. Visit ni.com/training for more information.
  - The Skills Guide assists you in identifying the proficiency requirements of your current application and gives you options for obtaining those skills consistent with your time and budget constraints and personal learning preferences. Visit ni.com/skills-guide to see these custom paths.
  - NI offers courses in several languages and formats including instructor-led classes at facilities worldwide, courses on-site at your facility, and online courses to serve your individual needs.
- **Technical Support**—Support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—Visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Software Support Service Membership**—The Standard Service Program (SSP) is a renewable one-year subscription included with almost every NI software product, including NI Developer Suite. This program entitles members to direct access to NI Applications Engineers through phone and email for one-to-one technical support, as well as exclusive access to online training modules at ni.com/self-paced-training. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.

- **Declaration of Conformity (DoC)**—A DoC is our claim of compliance with the Council of the European Communities using the manufacturer's declaration of conformity. This system affords the user protection for electromagnetic compatibility (EMC) and product safety. You can obtain the DoC for your product by visiting `ni.com/certification`.

For information about other technical support options in your area, visit `ni.com/services`, or contact your local office at `ni.com/contact`.

You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

C. Other National Instruments Training Courses

National Instruments offers several training courses for LabVIEW users. These courses continue the training you received here and expand it to other areas. Visit `ni.com/training` to purchase course materials or sign up for instructor-led, hands-on courses at locations around the world.

D. National Instruments Certification

Earning an NI certification acknowledges your expertise in working with NI products and technologies. The measurement and automation industry, your employer, clients, and peers recognize your NI certification credential as a symbol of the skills and knowledge you have gained through experience. Visit `ni.com/training` for more information about the NI certification program.