LabVIEW™ Core 2 Course Manual

Course Software Version 2012
August 2012 Edition
Part Number 325292D-01

Worldwide Technical Support and Product Information
ni.com

Worldwide Offices
Visit ni.com/niglobal to access the branch office Web sites, 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 Web site at ni.com/info and enter the Info Code feedback.

Contents

Student Guide

A. NI Certification v
B. Course Description v
C. What You Need to Get Started vi
D. Installing the Course Software vii
E. Course Goals vii
F. Course Conventions vii

Lesson 1
Moving Beyond Dataflow

A. Asynchronous Communication 1-2
B. Queues 1-2
C. Event-Driven Programming 1-5

Lesson 2
Implementing Design Patterns

A. Design Patterns 2-2
B. Simple Design Patterns 2-2
C. Multiple Loop Design Patterns 2-8
D. Error Handlers 2-14
E. Generating Error Codes and Messages 2-17
F. Timing a Design Pattern 2-19
G. Functional Global Variable Design Pattern 2-24

Lesson 3
Controlling the User Interface

A. VI Server Architecture 3-2
B. Property Nodes 3-3
C. Invoke Nodes 3-4
D. Control References 3-5

Lesson 4
File I/O Techniques

A. Compare File Formats 4-2
B. Create File and Folder Paths 4-4
C. Write and Read Binary Files 4-5
D. Work with Multichannel Text Files and Headers 4-11
E. Access TDMS Files in LabVIEW and Excel 4-16

Lesson 5
Improving an Existing VI
A. Refactoring Inherited Code....5-2
B. Typical Refactoring Issues....5-4

Lesson 6
Deploying an Application
A. Preparing the Files....6-2
B. Build Specifications....6-3
C. Create and Debug an Application....6-4

Appendix A
Additional Information and Resources

Glossary

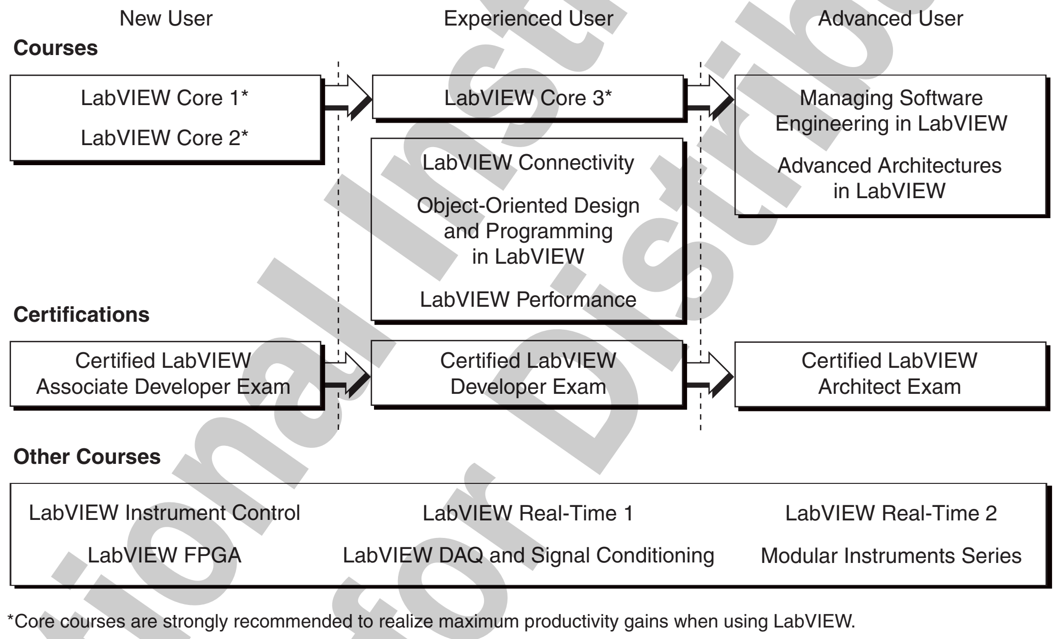
Student Guide

Thank you for purchasing the *LabVIEW Core 2* course kit. You can begin developing an application soon after you complete this course. This course manual and the accompanying software are used in the two-day, hands-on *LabVIEW Core 2* course.

You can apply the full purchase of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit `ni.com/training` for online course schedules, syllabi, training centers, and class registration.

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. 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.



New User
Experienced User
Advanced User
Courses
LabVIEW Core 1*
LabVIEW Core 2*
LabVIEW Core 3*
LabVIEW Connectivity
Object-Oriented Design and Programming in LabVIEW
LabVIEW Performance
Managing Software Engineering in LabVIEW
Advanced Architectures in LabVIEW
Certifications
Certified LabVIEW Associate Developer Exam
Certified LabVIEW Developer Exam
Certified LabVIEW Architect Exam
Other Courses
LabVIEW Instrument Control
LabVIEW Real-Time 1
LabVIEW Real-Time 2
LabVIEW FPGA
LabVIEW DAQ and Signal Conditioning
Modular Instruments Series
*Core courses are strongly recommended to realize maximum productivity gains when using LabVIEW.

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, that you have experience writing algorithms in the form of flowcharts or block diagrams, and that you have taken the *LabVIEW Core 1* course or have equivalent experience. The course and exercise manuals are divided into lessons, described as follows.

In the course manual, each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A description of the topics in the lesson
- A summary or quiz that tests and reinforces important concepts and skills taught in the lesson

In the exercise manual, each lesson consists of the following:

- A set of exercises to reinforce those topics
- Some lessons include optional and challenge exercise sections or a set of additional exercises to complete if time permits

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 2012 or later

☐ DAQmx 9.5.5 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

F. Course Conventions

The following conventions appear in this course manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **Tools»Instrumentation»Find Instrument Drivers** directs you to drop down the **Tools** menu, select the **Instrumentation** item, and finally select the **Find Instrument Drivers** option.
💡	This icon denotes a tip, which alerts you to advisory information.

📝	This icon denotes a note, which alerts you to important information.
⚠	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes sections of dialog boxes and hardware labels.
italic	Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

1

Moving Beyond Dataflow

As you learned in *LabVIEW Core 1*, LabVIEW is a dataflow language where the flow of data determines the execution order of block diagram elements. A block diagram node executes when it receives all required inputs. When a node executes, it produces output data and passes the data to the next node in the dataflow path. The movement of data through wires and nodes determines the execution order of the VIs and functions on the block diagram. This type of communication between nodes is referred to as synchronous communication.

Topics

A. Asynchronous Communication

B. Queues

C. Event-Driven Programming

A. Asynchronous Communication

Although LabVIEW is a dataflow language that uses wires to transfer data between functions, there are situations where communicating asynchronously, or without wires, is desirable. In this lesson you learn two important techniques for communicating asynchronously—queues for communicating between parallel loops and events for communicating between the user interface and the block diagram.

B. Queues

Use queues to communicate data between parallel loops in LabVIEW. A queue can hold data of any type and can store multiple pieces of data. By default, queues work in a first in, first out (FIFO) manner. Therefore, the first piece of data inserted into the queue is the first piece of data that is removed from the queue. Use a queue when you want to process all data placed in the queue.

Variables are useful in LabVIEW for passing data between parallel processes. However, when using variables it is often difficult to synchronize data transfers, which may cause you to read duplicate data or to miss data. Further more, you must take care to avoid race conditions. This lesson introduces queues as alternative methods for passing data between parallel processes. Queues have advantages over using variables because of the ability to synchronize the transfer of data.

Queue Operations

Use the queue operations functions to create and use queues for communicating data between different sections of a VI and different VIs.

Table 1-1 describes the queue operations functions you use in this course.

Table 1-1. Queue Operations Functions

Function	Description
Dequeue Element	Removes an element from the front of a queue and returns the element.
Enqueue Element	Adds an element to the back of a queue.
Enqueue Element at Opposite End	Adds an element to the front of a queue.

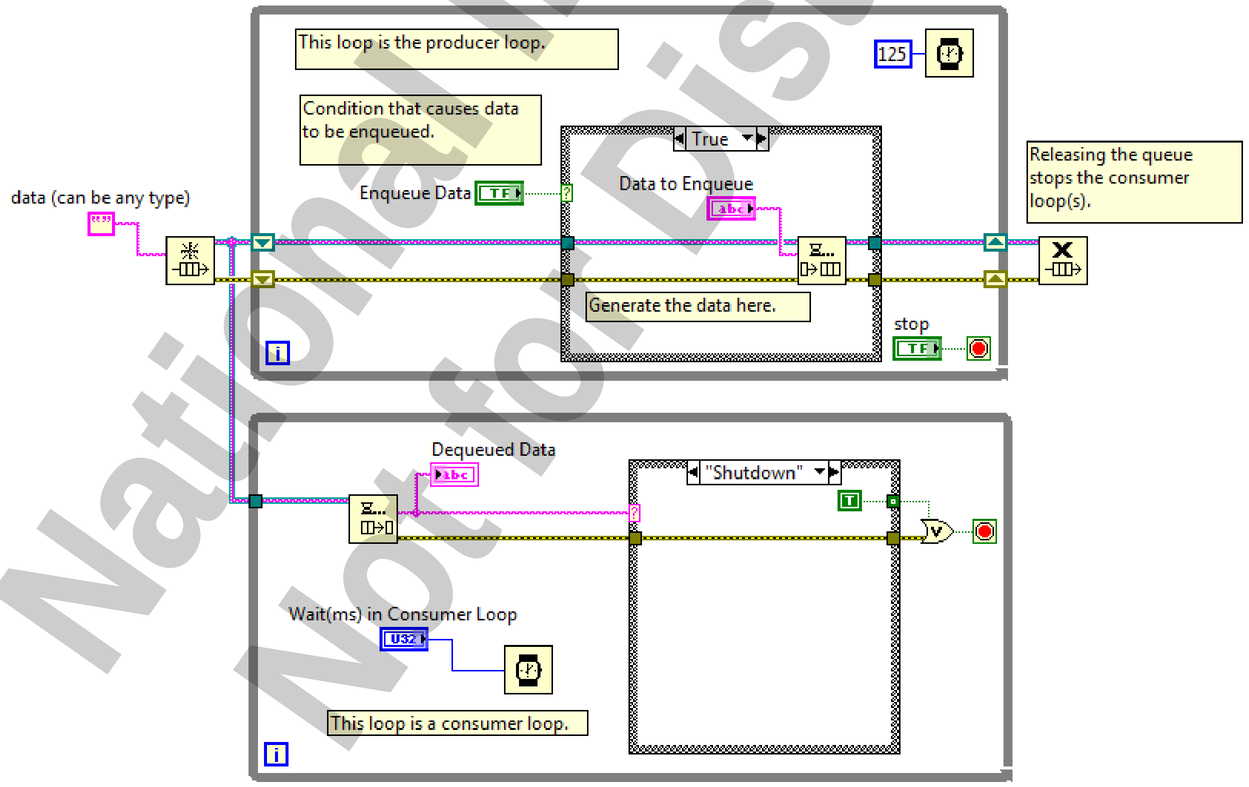
Table 1-1. Queue Operations Functions (Continued)

Function	Description
Get Queue Status	Returns information about the current state of a queue, such as the number of elements currently in the queue.
Obtain Queue	Returns a reference to a queue.
Release Queue	Releases a reference to a queue.

Refer to the *Queue Operations Functions* topic of the *LabVIEW Help* for a complete list and description of queue operations.

When used with the producer/consumer design pattern, queues pass data and synchronize the loops as shown in Figure 1-1.

Figure 1-1. Producer/Consumer Design Pattern (Data) Using Queues


This loop is the producer loop.
125
Condition that causes data to be enqueued.
True
Data to Enqueue
Enqueue Data
data (can be any type)
Releasing the queue stops the consumer loop(s).
Generate the data here.
stop
Dequeued Data
"Shutdown"
Wait(ms) in Consumer Loop
This loop is a consumer loop.

The queue is created before the loops begin using the Obtain Queue function. The producer loop uses the Enqueue Element function to add data to the queue. The consumer loop removes data from the queue using the Dequeue Element function. The consumer loop does not execute until data is available in the queue. After the VI has finished using the queues, the Release Queue function releases the queues. When the queue releases, the Dequeue Element function generates an error, effectively stopping the consumer loop. This eliminates the need to use a variable to stop the loops.

The following benefits result from using queues in the producer/consumer design pattern:

- Both loops are synchronized to the producer loop. The consumer loop only executes when data is available in the queue.
- You can use queues to create globally available data that is queued, removing the possibility of losing the data in the queue when new data is added to the queue.
- Using queues creates efficient code. You need not use polling to determine when data is available from the producer loop.

Queues are also useful for holding state requests in a state machine. In the implementation of a state machine that you have learned, if two states are requested simultaneously, you might lose one of the state requests. A queue stores the second state request and executes it when the first has finished.

Case Study: Weather Station Project

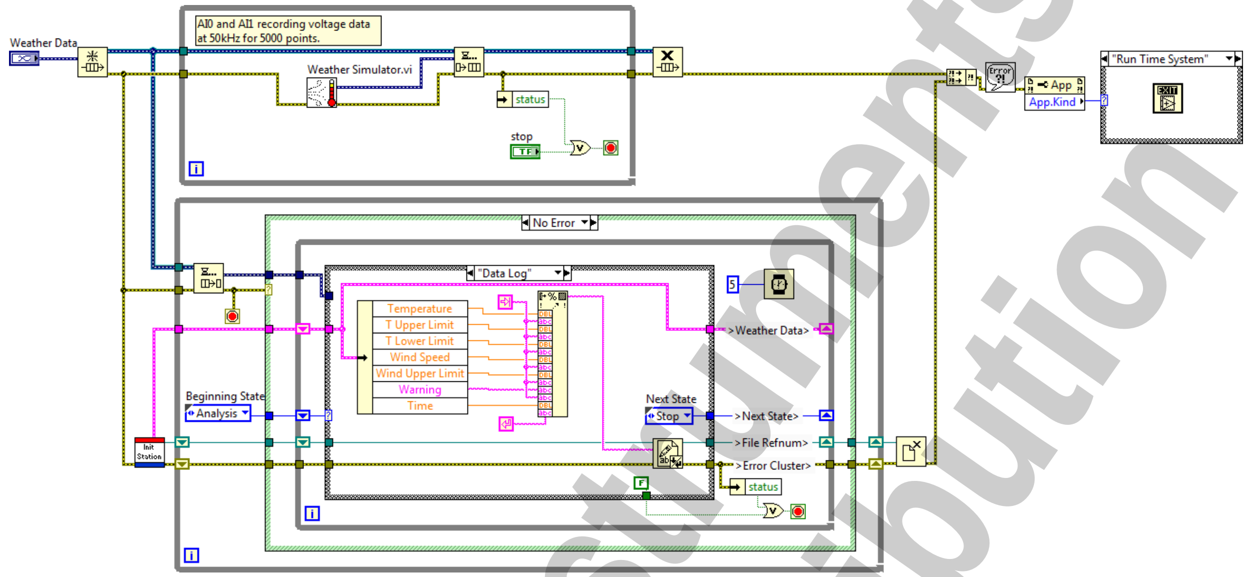
The weather station project acquires temperature and wind speed data, and analyzes it to determine if the situation requires a warning. If the temperature is too high or too low, it alerts the user to a danger of heatstroke or freezing. It also monitors the wind speed to generate a high wind warning when appropriate.

The block diagram consists of two parallel loops, which are synchronized using queues. One loop acquires data for temperature and wind speed and the other loop analyzes the data. The loops in the block diagram use the producer/consumer design pattern and pass the data through the queue. Queues help process every reading acquired from the DAQ Assistant.

Code for acquiring temperature and wind speed is placed in the producer loop. Code containing the state machine for analysis of temperature-weather conditions is within the no error case of the consumer loop. The code using a queue is more readable and efficient than the code using only state machine architecture. The Obtain Queue function creates the queue reference. The producer loop uses the Enqueue Element function to add data obtained from the DAQ Assistant to the queue. The consumer loop uses the Dequeue Element function to get the data from the queue and provide it to the state machine for analysis. The Release Queue function marks the end of queue by destroying it. The use of queues also eliminates the need for a shared variable to stop the loops because the Dequeue Element function stops the consumer loop when the queue is released.

Figure 1-2 shows the block diagram consisting of a producer and a consumer loop. Data transfer and synchronization between the loops is achieved by the queue functions.

Figure 1-2. Data Transfer and Synchronization of Parallel Loops Using Queues



C. Event-Driven Programming

Event-driven programming is a method of programming where the program waits on an event to occur before executing one or more functions. The features of event-driven programming extend the LabVIEW dataflow environment to allow the user's direct interaction with the front panel and other asynchronous activity to further influence block diagram execution.

Events

What Are Events?

An event is an asynchronous notification that something has occurred. Events can originate from the user interface, external I/O, or other parts of the program. User interface events include mouse clicks, key presses, and so on. External I/O events include hardware timers or triggers that signal when data acquisition completes or when an error condition occurs. Other types of events can be generated programmatically and used to communicate with different parts of the program. LabVIEW supports user interface and programmatically generated events. LabVIEW also supports ActiveX and .NET generated events, which are external I/O events.

In an event-driven program, events that occur in the system directly influence the execution flow. In contrast, a procedural program executes in a pre-determined, sequential order. Event-driven programs usually include a loop that waits for an event to occur, executes code to respond to the event, and reiterates to wait for the next event. How the program responds to each event depends on the code written for that specific event. The order in which an event-driven program executes depends on which events occur and on the order in which they occur. Some sections of the program might execute frequently because the events they handle occur frequently, and other sections of the program might not execute at all because the events never occur.

Polling vs Event Structures

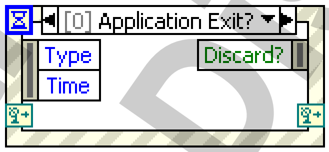
Use user interface events in LabVIEW to synchronize user actions on the front panel with block diagram execution. Events allow you to execute a specific event-handling case each time a user performs a specific action. Without events, the block diagram must poll the state of front panel objects in a loop, checking to see if any change has occurred. Polling the front panel requires a significant amount of CPU time and can fail to detect changes if they occur too quickly.

By using events to respond to specific user actions, you eliminate the need to poll the front panel to determine which actions the user performed. Instead, LabVIEW actively notifies the block diagram each time an interaction you specified occurs. Using events reduces the CPU requirements of the program, simplifies the block diagram code, and guarantees that the block diagram can respond to all interactions the user makes.

Use programmatically generated events to communicate among different parts of the program that have no dataflow dependency. Programmatically generated events have many of the same advantages as user interface events and can share the same event-handling code, making it easy to implement advanced architectures, such as queued state machines using events.

Event Structure Components

Use the Event structure, shown as follows, to handle events in a VI.



The Event structure works like a Case structure with a built-in Wait on Notification function. The Event structure can have multiple cases, each of which is a separate event-handling routine. You can configure each case to handle one or more events, but only one of these events can occur at a time. When the Event structure executes, it waits until one of the configured events occur, then executes the corresponding case for that event. The Event structure completes execution after handling exactly one event. It does not implicitly loop to handle multiple events. Like a Wait on Notification function, the Event structure can time out while waiting for notification of an event. When this occurs, a specific Timeout case executes.

The event selector label at the top of the Event structure indicates which events cause the currently displayed case to execute.

View other event cases by clicking the down arrow next to the case name and selecting another case from the shortcut menu.

The Timeout terminal at the top left corner of the Event structure specifies the number of milliseconds to wait for an event before timing out.

The default is –1, which specifies to wait indefinitely for an event to occur. If you wire a value to the Timeout terminal, you must provide a Timeout case.

The Event Data Node behaves similarly to the Unbundle By Name function.

This node is attached to the inside left border of each event case. The node identifies the data LabVIEW provides when an event occurs. You can resize this node vertically to add more data items, and you can set each data item in the node to access any event data element. The node provides different data elements in each case of the Event structure depending on which event(s) you configure that case to handle. If you configure a single case to handle multiple events, the Event Data Node provides only the event data elements that are common to all the events configured for that case.

The Event Filter Node is similar to the Event Data Node.

This node is attached to the inside right border of filter event cases. The node identifies the subset of data available in the Event Data Node that the event case can modify. The node displays different data depending on which event(s) you configure that case to handle. By default, these items are inplace to the corresponding data items in the Event Data Node. If you do not wire a value to a data item of an Event Filter Node, that data item remains unchanged.

Refer to the *Notify and Filter Events* section of this lesson for more information about filter events.

The dynamic event terminals are available by right-clicking the Event structure and selecting **Show Dynamic Event Terminals** from the shortcut menu.

These terminals are used only for dynamic event registration.

Refer to the *Using Events in LabVIEW* topic of the *LabVIEW Help* for more information about using these terminals.

Note Like a Case structure, the Event structure supports tunnels. However, by default you do not have to wire Event structure output tunnels in every case. All unwired tunnels use the default value for the tunnel data type. Right-click a tunnel and deselect **Use Default If Unwired** from the shortcut menu to revert to the default Case structure behavior where tunnels must be wired in all cases. You also can configure the tunnels to wire the input and output tunnels automatically in unwired cases.

Refer to the *LabVIEW Help* for information about the default values for data types.

Using Events in LabVIEW

LabVIEW can generate many different events. To avoid generating unwanted events, use event registration to specify which events you want LabVIEW to notify you about. LabVIEW supports two models for event registration—static and dynamic.

Static registration allows you to specify which events on the front panel of a VI you want to handle in each Event structure case on the block diagram of that VI. LabVIEW registers these events automatically when the VI runs, so the Event structure begins waiting for events as soon as the VI begins running. Each event is associated with a control on the front panel of the VI, the front panel window of the VI as a whole, or the LabVIEW application. You cannot statically configure an Event structure to handle events for the front panel of a different VI. Configuration is static because you cannot change at run time which events the Event structure handles.

Dynamic event registration avoids the limitations of static registration by integrating event registration with the VI Server, which allows you to use Application, VI, and control references to specify at run time the objects for which you want to generate events. Dynamic registration provides more flexibility in controlling what events LabVIEW generates and when it generates them. However, dynamic registration is more complex than static registration because it requires using VI Server references with block diagram functions to explicitly register and unregister for events rather than handling registration automatically using the information you configured in the Event structure.

Note In general, LabVIEW generates user interface events only as a result of direct user interaction with the active front panel. LabVIEW does not generate events, such as Value Change, when you use shared variables, global variables, local variables, and so on. However, you can use the Value (Signaling) property to generate a Value Change event programmatically. In many cases, you can use programmatically generated events instead of queues.

The event data provided by a LabVIEW event always include a time stamp, an enumeration that indicates which event occurred, and a VI Server reference to the object that triggered the event. The time stamp is a millisecond counter you can use to compute the time elapsed between two events or to determine the order of occurrence. The reference to the object that generated the event is strictly typed to the VI Server class of that object. Events are grouped into classes according to what type of object generates the event, such as Application, VI, or Control. If a single case handles multiple events for objects of different VI Server classes, the reference type is the common parent class of all objects. For example, if you configure a single case in the Event structure to handle events for a numeric control and a color ramp control, the type of the control reference of the event source is Numeric because the numeric and color ramp controls are in the

Numeric class. If you register for the same event on both the VI and Control class, LabVIEW generates the VI event first.

Note Clusters are the only container objects for which you can generate events. LabVIEW generates Control events for clusters, before it generates events for the objects they contain, except in the case of the Value Change event. The Value Change event generates the event on an element in the cluster, then on the cluster itself. If the Event structure case for a VI event or for a Control event on a container object discards the event, LabVIEW does not generate further events.

Each Event structure and Register For Events function on the block diagram owns a queue that LabVIEW uses to store events. When an event occurs, LabVIEW places a copy of the event into each queue registered for that event. An Event structure handles all events in its queue and the events in the queues of any Register For Events functions that you wired to the dynamic event terminals of the Event structure. LabVIEW uses these queues to ensure that events are reliably delivered to each registered Event structure in the order the events occur.

By default, when an event enters a queue, LabVIEW locks the front panel that contains the object that generated that event. LabVIEW keeps the front panel locked until all Event structures finish handling the event. While the front panel is locked, LabVIEW does not process front panel activity but places those interactions in a buffer and handles them when the front panel is unlocked.

For example, a user might anticipate that an event case launches an application that requires text entry. Since the user already knows text entry is needed, he might begin typing before the application appears on the front panel. If the **Lock front panel (defer processing of user action) until this event case completes** option is enabled, once the application launches and appears on the front panel, it processes the key presses in the order in which they occurred. If the **Lock front panel (defer processing of user action) until this event case completes** option is disabled, the key presses might be processed elsewhere on the front panel, since LabVIEW does not queue their execution to depend on the completion of the event case.

Front panel locking does not affect certain actions, such as moving the window, interacting with the scroll bars, and clicking the **Abort** button.

LabVIEW can generate events even when no Event structure is waiting to handle them. Because the Event structure handles only one event each time it executes, place the Event structure in a While Loop to ensure that an Event structure can handle all events that occur.

Caution If no Event structure executes to handle an event and front panel locking is enabled, the user interface of the VI becomes unresponsive. If this occurs, click the **Abort** button to stop the VI. You can disable front panel locking by right-clicking the Event structure and removing the checkmark from the **Lock front panel (defer processing of user action) until this event case completes** checkbox in the **Edit Events** dialog box. You cannot turn off front panel locking for filter events.

Static Event Registration

Static event registration is available only for user interface events. Use the Edit Events dialog box to configure an Event structure to handle a statically registered event. Select the event source, which can be the application, the VI, or an individual control. Select a specific event the event source can generate, such as Panel Resize, Value Change, and so on. Edit the case to handle the event data according to the application requirements.

LabVIEW statically registers events automatically and transparently when you run a VI that contains an Event structure. LabVIEW generates events for a VI only while that VI is running or when another running VI calls the VI as a subVI.

When you run a VI, LabVIEW sets that top-level VI and the hierarchy of subVIs the VI calls on its block diagram to an execution state called reserved. You cannot edit a VI or click the Run button while the VI is in the reserved state because the VI can be called as a subVI at any time while its parent VI runs. When LabVIEW sets a VI to the reserved state, it automatically registers the events you statically configured in all Event structures on the block diagram of that VI. When the top-level VI finishes running, LabVIEW sets it and its subVI hierarchy to the idle execution state and automatically unregisters the events.

Configuring Events

Before you configure events for the Event structure to handle, refer to the *Caveats and Recommendations when Using Events in LabVIEW* topic of the *LabVIEW Help*.

Complete the following steps to configure an Event structure case to handle an event.

1. (Optional) If you want to configure the Event structure to handle a user event, a Boolean control within a radio buttons control, or a user interface event that is generated based on a reference to an application, VI, or control, you first must dynamically register that event. Refer to the *Dynamically Registering Events* topic of the *LabVIEW Help* for more information about using dynamic events.
2. Right-click the border of the Event structure and select **Edit Events Handled by This Case** from the shortcut menu to display the **Edit Events** dialog box to edit the current case. You also can select **Add Event Case** from the shortcut menu to create a new case.
3. Specify an event source in the **Event Sources** pane.
4. Select the event you want to configure for the event source, such as **Key Down**, **Timeout**, or **Value Change** from the **Events** list. When you select a dynamic event source from the **Event Sources** list, the **Events** list displays that event. This is the same event you selected when you registered the event. If you have registered for events dynamically and wired **event reg refnum out** to the dynamic event terminal, the sources appear in the **Dynamic** section.
5. If you want to add additional events for the current case to handle, click the + button and repeat steps 3 and 4 to specify each additional event. The **Event Specifiers** section at the top of the dialog box lists all the events for the case to handle. When you click an item in this list, the **Event Sources** section updates to highlight the event source you selected. You can repeat steps 3 and 4 to redefine each event or click the **X** button to remove the selected event.

6. Click the **OK** button to save the configuration and close the dialog box. The event cases you configured appear as selection options in the event selector label at the top of the Event structure and the Event Data node displays the data common to all events handled in that case.
7. (Optional) You can use a Timeout event to configure an Event structure to wait a specified amount of time for an event to occur. Wire a value to the Timeout terminal at the top left of the Event structure to specify the number of milliseconds the Event structure should wait for an event to occur before generating a Timeout event. The default value for the Timeout terminal is –1, which specifies to wait indefinitely for an event to occur.
8. Repeat steps 1 through 6 for each event case you want to configure.

Notify and Filter Events

Notify events are an indication that a user action has already occurred, such as when the user has changed the value of a control. Use notify events to respond to an event after it has occurred and LabVIEW has processed it. You can configure any number of Event structures to respond to the same notify event on a specific object. When the event occurs, LabVIEW sends a copy of the event to each Event structure configured to handle the event in parallel.

Filter events inform you that the user has performed an action before LabVIEW processes it, which allows you to customize how the program responds to interactions with the user interface. Use filter events to participate in the handling of the event, possibly overriding the default behavior for the event. In an Event structure case for a filter event, you can validate or change the event data before LabVIEW finishes processing it, or you can discard the event entirely to prevent the change from affecting the VI. For example, you can configure an Event structure to discard the Panel Close? event, which prevents the user from interactively closing the front panel of the VI.

Filter events have names that end with a question mark, such as Panel Close?, to help you distinguish them from notify events. Most filter events have an associated notify event of the same name, but without the question mark, which LabVIEW generates after the filter event if no event case discarded the event.

For example, you can use the Mouse Down? and Shortcut Menu Activation? filter events to display a context menu when you left-click a control. To perform this action, modify the data returned by the **Button** event data field of the Mouse Down? filter event. The value of the left mouse button is 1, and the value of the right mouse button is 2. In order to display the context menu when you left-click a control, change the **Button** event data field to 2 so that LabVIEW treats a left-click like a right-click.

As with notify events, you can configure any number of Event structures to respond to the same filter event on a specific object. However, LabVIEW sends filter events sequentially to each Event structure configured for the event. The order in which LabVIEW sends the event to each Event structure depends on the order in which the events were registered. Each Event structure must complete its event case for the event before LabVIEW can notify the next Event structure. If an Event structure case changes any of the event data, LabVIEW passes the changed data to subsequent Event structures in the chain. If an Event structure in the chain discards the event, LabVIEW does not pass the event to any Event structures remaining in the chain. LabVIEW completes processing the user action which triggered the event only after all configured Event structures handle the event without discarding it.

Note National Instruments recommends you use filter events only when you want to take part in the handling of the user action, either by discarding the event or by modifying the event data. If you only want to know that the user performed a particular action, use notify events.

Event structure cases that handle filter events have an Event Filter Node. You can change the event data by wiring new values to these terminals. If you do not wire a value to the data item of the Event Filter Node, the default value equals the value that the corresponding item in the Event Data Node returns. You can completely discard an event by wiring a TRUE value to the Discard? terminal.

Note A single case in the Event structure cannot handle both notify and filter events. A case can handle multiple notify events but can handle multiple filter events only if the event data items are identical for all events.

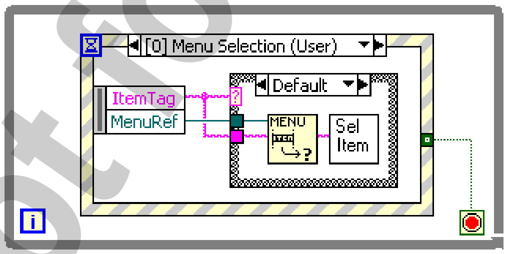
Refer to the *Using Events in LabVIEW* section of this lesson for more information about event registration.

Tip In the Edit Events dialog box, notify events are signified by a green arrow, and filter events are signified by a red arrow.

Event Example

Figure 1-3 shows an Event structure configured with the Menu Selection (User) event. This VI uses the Event structure to capture menu selections made using the user-defined menu named `sample.rtm`. The ItemTag returns the menu item that was selected and the MenuRef returns the refnum to the menubar. This information is passed to the Get Menu Item Info function. Refer to `examples\general\uievents.llb` for more examples of using events.

Figure 1-3. Menu Selection (User) Event



Note If you use the Get Menu Selection function with an Event structure configured to handle the same menu item, the Event structure takes precedence, and LabVIEW ignores the Get Menu Selection function. In any given VI, use the Event structure or the Get Menu Selection function to handle menu events, not both.

Caveats and Recommendations

The following list describes some of the caveats and recommendations to consider when incorporating events into LabVIEW applications.

- Avoid using an Event structure outside a loop.

  LabVIEW can generate events even when no Event structure is waiting to handle them. Because the Event structure handles only one event each time it executes, place the Event structure in a While Loop that terminates when the VI is no longer interested in events to ensure that an Event structure handles all events that occur.
- Remember to read the terminal of a latched Boolean control in its Value Change event case.

  When you trigger an event on a Boolean control configured with a latching mechanical action, the Boolean control does not reset to its default value until the block diagram reads the terminal on the Boolean control. You must read the terminal inside the event case for the mechanical action to work correctly.
- Avoid placing two Event structures in one loop.

  National Instruments recommends that you place only one Event structure in a loop. When an event occurs in this configuration, the Event structure handles the event, the loop iterates, and the Event structure waits for the next event to occur. If you place two Event structures in a single loop, the loop cannot iterate until both Event structures handle an event. If you have enabled front panel locking for the Event structures, the user interface of the VI can become unresponsive depending on how the user interacts with the front panel.

Refer to the *Caveats and Recommendations when Using Events in LabVIEW* topic of the *LabVIEW Help* for more caveats and recommendations when you use events in LabVIEW.

Self-Review: Quiz

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

2. Match the following:

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

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

4. The Event structure handles only one event each time it executes.
   a. True
   b. False

Self-Review: Quiz Answers

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

2. Match the following:

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

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

4. The Event structure handles only one event each time it executes.
   - **a. True**
   - b. False

Notes

2

Implementing Design Patterns

You can develop better programs in LabVIEW and in other programming languages if you follow consistent programming techniques. Design patterns represent techniques that have proved themselves useful time and time again. To facilitate development, LabVIEW provides templates for several common design patterns. This lesson discusses two different categories of programming design patterns—single loop and multiple loops.

Single loop design patterns include the simple VI, the general VI, and the state machine.

Multiple loop design patterns include the parallel loop VI, the master/slave, and the producer/consumer.

Understanding the appropriate use of each design pattern helps you create more efficient LabVIEW VIs.

Topics

A. Design Patterns

B. Simple Design Patterns

C. Multiple Loop Design Patterns

D. Error Handlers

E. Generating Error Codes and Messages

F. Timing a Design Pattern

G. Functional Global Variable Design Pattern

A. Design Patterns

Application design patterns represent LabVIEW code implementations and techniques that are solutions to specific problems in software design. Design patterns typically evolve through the efforts of many developers and are fine-tuned for simplicity, maintainability, and readability. Design patterns represent the techniques that have proved themselves useful over time. Furthermore, as a pattern gains acceptance, it becomes easier to recognize—this recognition alone helps you to read and make changes to your code.

B. Simple Design Patterns

You learned to design three different types of design patterns in the *LabVIEW Core 1* course—the simple architecture, the general architecture, and the state machine.

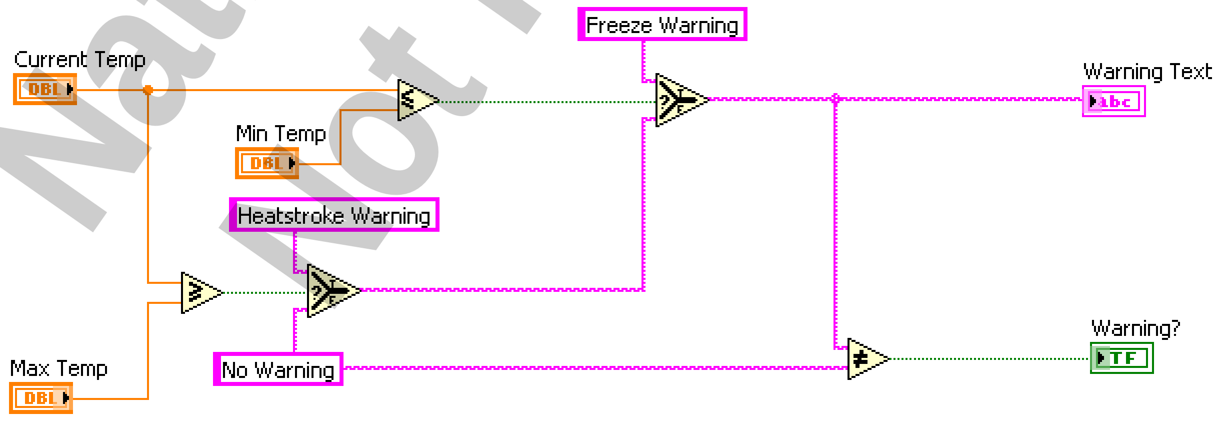
Simple VI Design Pattern

When performing calculations or making quick lab measurements, you do not need a complicated architecture. Your program might consist of a single VI that takes a measurement, performs calculations, and either displays the results or records them to disk. The simple VI design pattern usually does not require a specific start or stop action from the user. The user just clicks the **Run** button. Use this architecture for simple applications or for functional components within larger applications. You can convert these simple VIs into subVIs that you use as building blocks for larger applications.

Figure 2-1 displays the block diagram of the Determine Warnings VI that was the course project in the *LabVIEW Core 1* course. This VI performs a single task—it determines what warning to output dependent on a set of inputs. You can use this VI as a subVI whenever you must determine the warning level.

Notice that the VI in Figure 2-1 contains no start or stop actions from the user. In this VI all block diagram objects are connected through data flow. You can determine the overall order of operations by following the flow of data. For example, the Not Equal function cannot execute until the Greater Than or Equal function, the Less Than or Equal function, and both Select functions have executed.

Figure 2-1. Simple VI Architecture



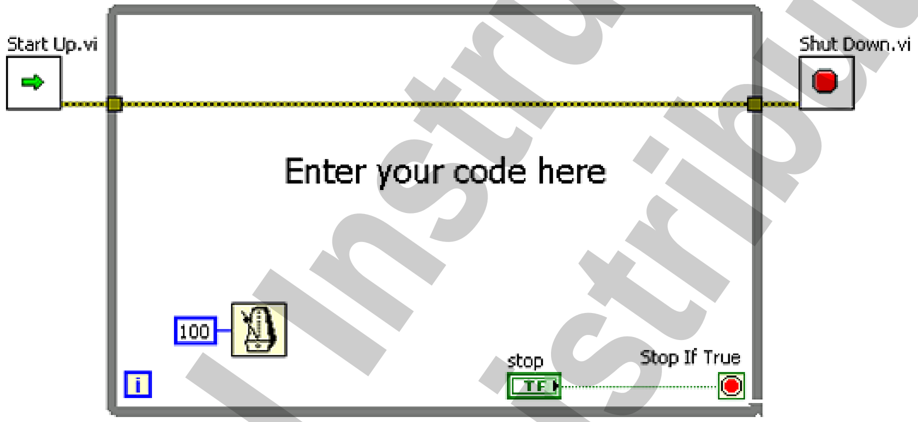
General VI Design Pattern

A general VI design pattern has three main phases—startup, main application, and shutdown. Each of the following phases may contain code that uses another type of design pattern.

- **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.

Figure 2-2 shows the general VI design pattern.

Figure 2-2. General VI Design Pattern Framework



In Figure 2-2, the error cluster wires control the execution order of the three sections. The While Loop does not execute until the Start Up VI finishes running and returns the error cluster data. Consequently, the Shut Down VI cannot run until the main application in the While Loop finishes and the error cluster data leaves the loop.

Tip Most loops require a Wait function, especially if that loop monitors user input on the front panel. Without the Wait function, the loop might run continuously and use all of the computer system resources. The Wait function forces the loop to run asynchronously even if you specify 0 milliseconds as the wait period. If the operations inside the main loop react to user inputs, you can increase the wait period to a level acceptable for reaction times. A wait of 100 to 200 ms is usually good because most users cannot detect that amount of delay between clicking a button on the front panel and the subsequent event execution.

For simple applications, the main application loop is obvious and contains code that uses the simple VI design pattern. When the application incudes complicated user interfaces or multiple tasks such as user actions, I/O triggers, and so on, the main application phase gets more complicated.

State Machine Design Pattern (Polling)

The state machine design pattern is a modification of the general design pattern. It usually has a start up and shut down phase. However, the main application phase consists of a Case structure embedded in the loop. This architecture allows you to run different code each time the loop executes, depending upon some condition. Each case defines a state of the machine, hence the name, state machine. Use this design pattern for VIs that are easily divided into several simpler tasks, such as VIs that act as a user interface.

A state machine in LabVIEW consists of a While Loop, a Case structure, and a shift register. Each state of the state machine is a separate case in the Case structure. You place VIs and other code that the state should execute within the appropriate case. A shift register stores the state that should execute upon the next iteration of the loop. The block diagram of a state machine VI with five states appears in Figure 2-3. Figure 2-4 shows the other cases, or states, of the state machine.

Figure 2-3. State Machine with Startup State

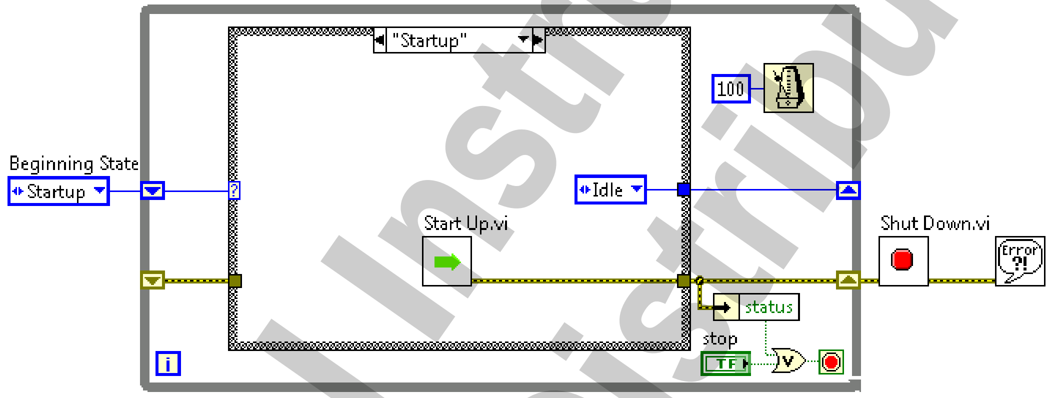
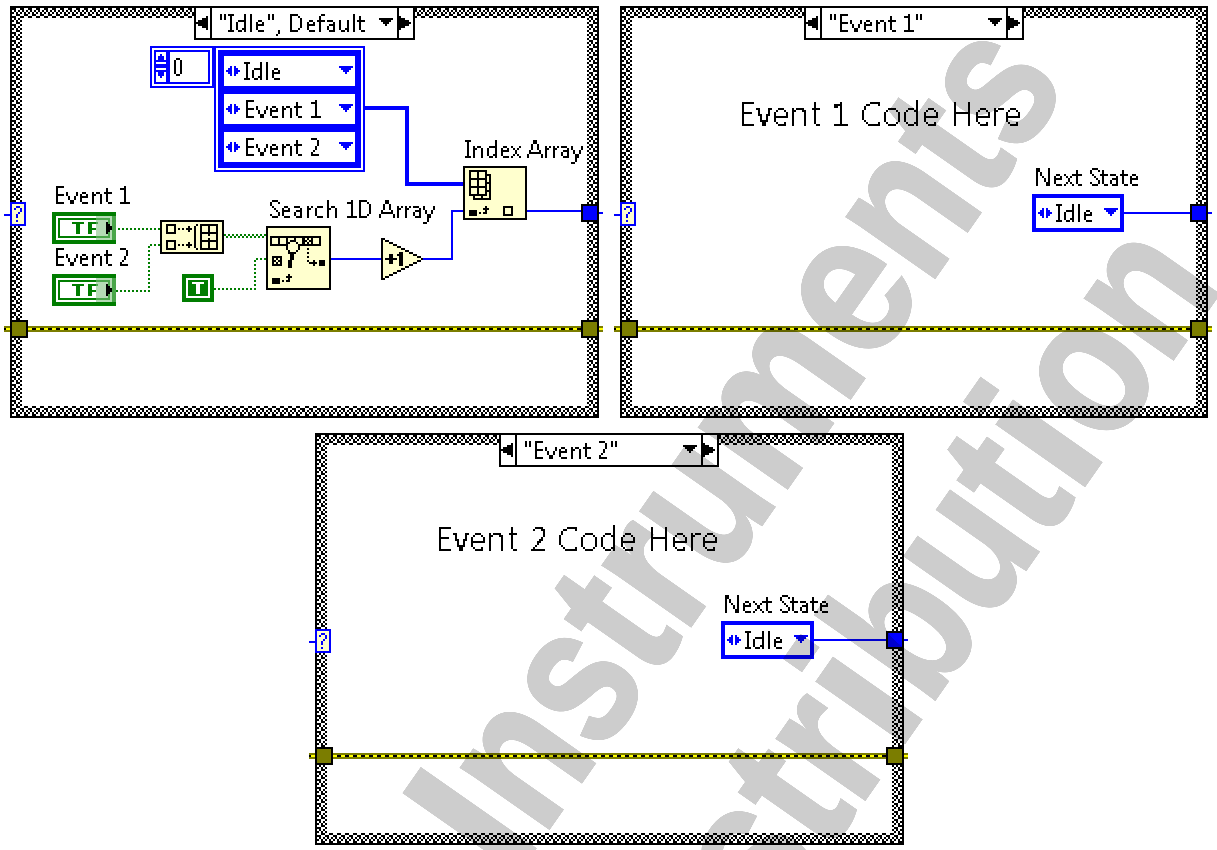


Figure 2-4. Idle (Default), Event 1, and Event 2 States



In the state machine design pattern, you design the list of possible tasks, or states, and then map them to each case. For the VI in the previous example, the possible states are Startup, Idle, Event 1, and Event 2. An enumerated constant stores the states. Each state has its own case in the Case structure. The outcome of one case determines which case to execute next. The shift register stores the value that determines which case to execute next.

The state machine design pattern can make the block diagram much smaller, and therefore, easier to read and debug. Another advantage of the state machine architecture is that each case determines the next state, unlike Sequence structures that must execute every frame in sequence.

A disadvantage of the state machine design pattern is that with the approach in the previous example, it is possible to skip states. If two states in the structure are called at the same time, this model handles only one state, and the other state does not execute. Skipping states can lead to errors that are difficult to debug because they are difficult to reproduce. More complex versions of the state machine design pattern contain extra code that creates a queue of events, or states, so that you do not miss a state. Refer to Lesson 1, *Moving Beyond Dataflow*, for more information about queue-based state machines.

State Machine Design Pattern (Event-Based)

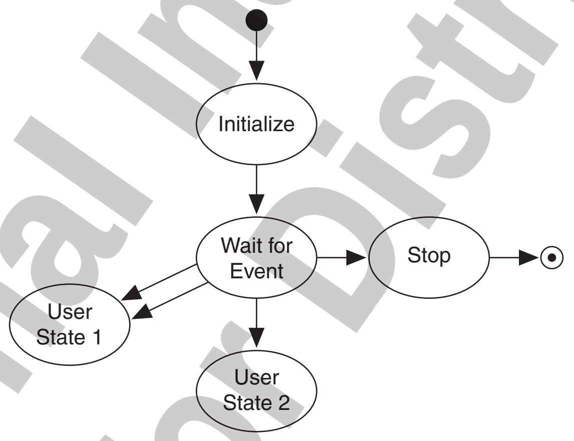
The polling-based state machine design pattern monitors front panel activity using standard dataflow techniques. An event-based state machine combines the powerful user interaction of the user interface event handler with transition flexibility of the state machine. Because this combination is so useful for a wide range of applications, LabVIEW provides a project-based template, the Simple State Machine, to simplify the process of creating an application based on this design pattern.

The Simple State Machine template is a customizable application which is in the form of a `.lvproj` file with supporting VIs and type definition controls. The application is based on the event-based state machine design pattern. The template includes ample documentation on how to modify the code to build a customized state machine application.

Use the Create Project dialog box to create a project from a template or sample project. Templates provide common architectures that you can modify to accomplish specific goals. Sample projects demonstrate how a template can be modified to accomplish specific goals.

Refer to the Single Shot Measurement sample project, available from the Create Project dialog box, for an example of adapting the Simple State Machine template to a measurement application.

Figure 2-5. Simple State Machine State Transition Diagram



After initialization, the state machine transitions to the Wait for Event state. This state contains an Event structure that waits for front panel changes. When a user clicks a button, LabVIEW recognizes the event and switches to the appropriate subdiagram of the Event structure. This subdiagram initiates a transition to the appropriate state.

Only one state executes at a time, and the single While Loop means all tasks execute at a single rate. If you need multi-rate or parallel tasks, consider a multi-loop design pattern. In this lesson you learn about the Producer/Consumer design pattern. In later courses you learn about other design patterns for which LabVIEW has templates, such as the Queued Message Handler or Actor Framework templates.

The Wait for Event state is the only one that recognizes user input. The state machine must be in this state for any user input to be accepted. Therefore, each state should be quick so the code can return to the Wait for Event state.

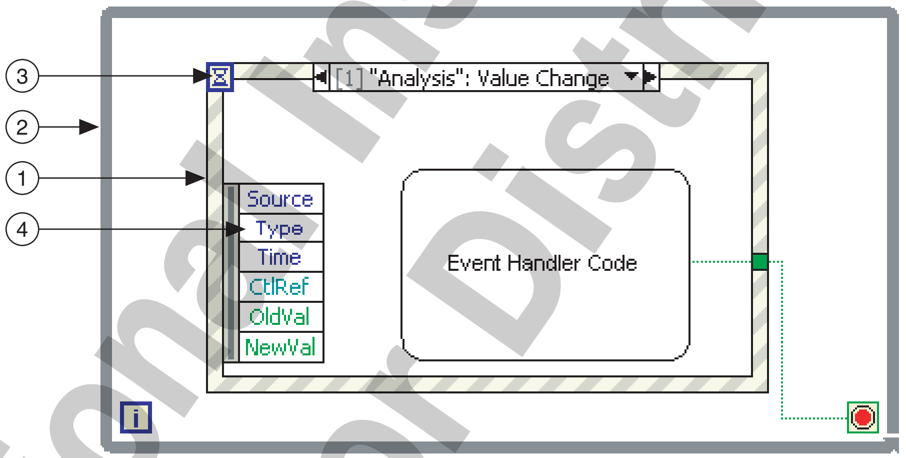
User Interface Event Handler Design Pattern

The user interface event handler design pattern provides a powerful and efficient architecture for handling user interaction with LabVIEW. Use the user interface event handler for detecting when a user changes the value of a control, moves or clicks the mouse, or presses a key.

The standard user interface event handler template consists of an Event structure contained in a While Loop, as shown in Figure 2-6. Configure the Event structure to have one case for each category of event you want to detect. Each event case contains the handling code that executes immediately after an event occurs.

Because the event handler loop wakes up precisely when an event occurs and sleeps in between events, you do not have to poll or read control values repeatedly in order to detect when a user clicks a button. The user interface event handler allows you to minimize processor use without sacrificing interactivity.

Figure 2-6. User Interface Event Handler Design Pattern



1	Event Structure	3	Timeout Terminal
2	While Loop	4	Event Data Node

A common problem when using the user interface event handler is that it computes the While Loop termination before the Event structure executes. This can cause the While Loop to iterate one more time than you expected. To avoid this situation, compute the While Loop termination within all your event handling code.

The event handler code must execute quickly, generally within 200 ms. Anything slower can make it feel as if the user interface is locked up. Also, if the event handler code takes a long time to execute, the Event structure might lock. By default, the front panel locks while an event is handled. You can disable front panel locking for each event case to make the user interface more responsive.

However, any new events that are generated while an event is being handled will not be handled immediately. So, the user interface will still seem unresponsive.

Any code that is in an event case cannot be shared with another Event structure case. You must use good code design when using the Event structure. Modularize code that will be shared between multiple Event structure cases.

The Event structure includes a Timeout event, which allows you to control when the Timeout event executes. For example, if you set a Timeout of 200 ms, the Timeout event case executes every 200 ms in the absence of other events. You can use the Timeout event to perform critical timing in your code.

C. Multiple Loop Design Patterns

There are several multiple loop design patterns, some of which are beyond the scope of this course. This course focuses on the producer/consumer design pattern because of its flexibility and versatility.

Producer/Consumer Design Pattern

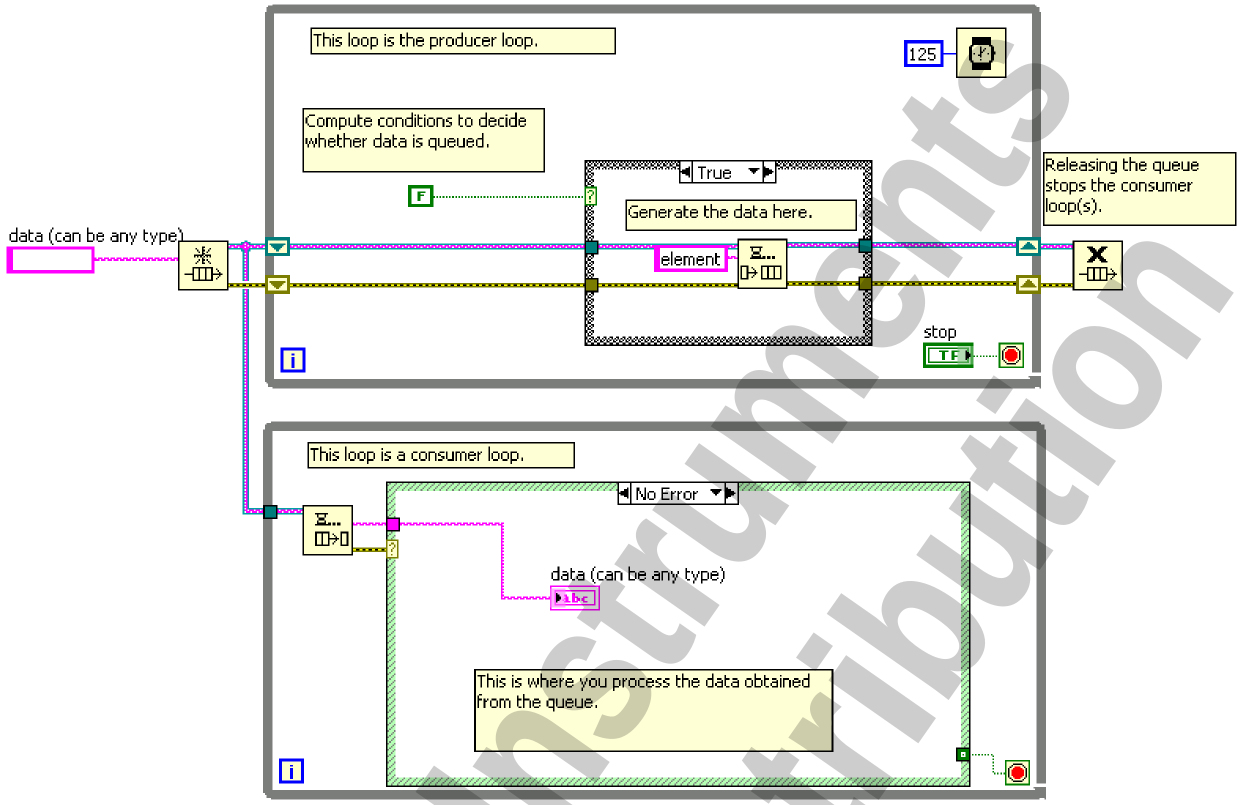
The producer/consumer design pattern is based on the master/slave design pattern and improves data sharing among multiple loops running at different rates. Similar to the master/slave design pattern, the producer/consumer design pattern separates tasks that produce and consume data at different rates. The parallel loops in the producer/consumer design pattern are separated into two categories—those that produce data and those that consume the data produced. Data queues communicate data among the loops. The data queues also buffer data among the producer and consumer loops.

Tip A buffer is a memory device that stores temporary data among two devices, or in this case, multiple loops.

Use the producer/consumer design pattern when you must acquire multiple sets of data that must be processed in order. Suppose you want to create a VI that accepts data while processing the data sets in the order they were received. The producer/consumer pattern is ideal for this type of VI because queuing (producing) the data occurs much faster than the data can be processed (consumed). You could put the producer and consumer in the same loop for this application, but the processing queue could not receive additional data until the first piece of data was completely processed. The producer/consumer approach to this VI queues the data in the producer loop and processes the data in the consumer loop, as shown in Figure 2-7.

Tip Queue functions allow you to store a set of data that can be passed among multiple loops running simultaneously or among VIs. Refer to Lesson 1, *Moving Beyond Dataflow*, for more information about queues and implementing applications using the producer/consumer design pattern.

Figure 2-7. Producer/Consumer Design Pattern


This loop is the producer loop.
125
Compute conditions to decide whether data is queued.
F
True
Generate the data here.
Releasing the queue stops the consumer loop(s).
data (can be any type)
element
stop
This loop is a consumer loop.
No Error
data (can be any type)
This is where you process the data obtained from the queue.

This design pattern allows the consumer loop to process the data at its own pace, while the producer loop continues to queue additional data.

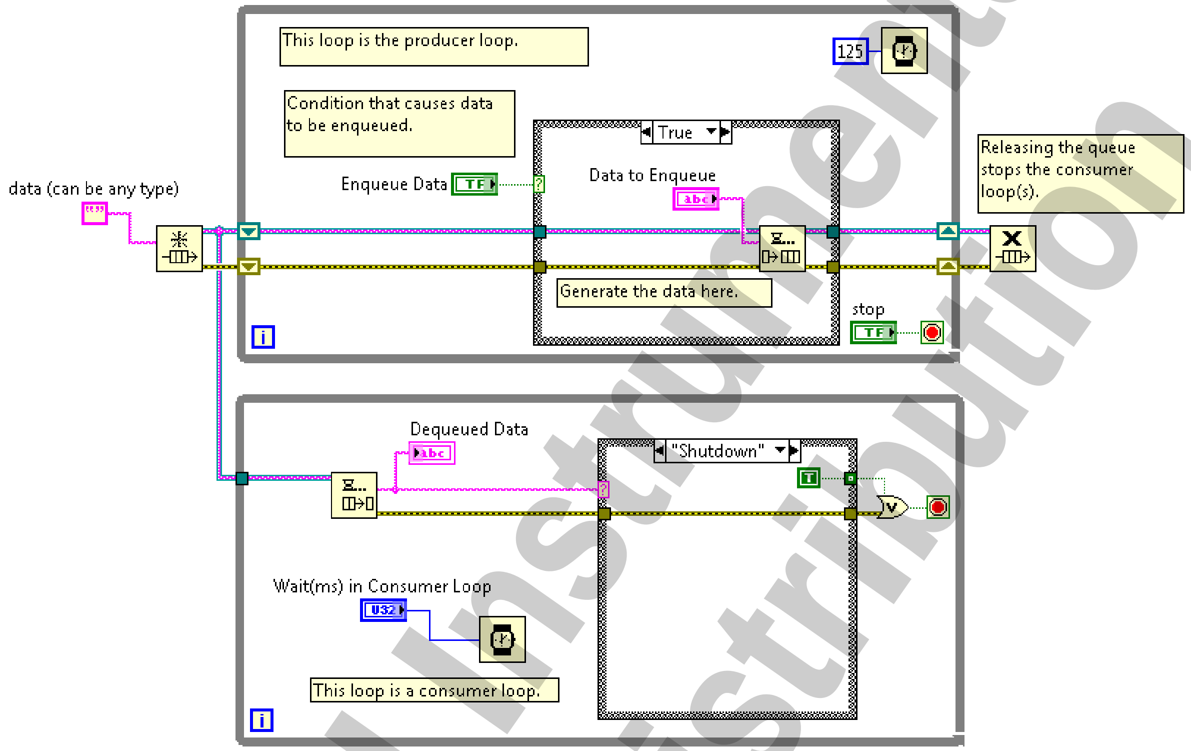
You also can use the producer/consumer design pattern to create a VI that analyzes network communication. This type of VI requires two processes to operate at the same time and at different speeds. The first process constantly polls the network line and retrieves packets. The second process analyzes the packets retrieved by the first process.

In this example, the first process acts as the producer because it supplies data to the second process, which acts as the consumer. The producer/consumer design pattern is an effective architecture for this VI. The parallel producer and consumer loops handle the retrieval and analysis of data off the network, and the queued communication between the two loops allows buffering of the network packets retrieved. Buffering can become important when network communication is busy. With buffering, packets can be retrieved and communicated faster than they can be analyzed.

Producer/Consumer Design Pattern (Data)

The producer/consumer design pattern (data) enhances data sharing among multiple loops running at different rates. There are two categories of parallel loops in the producer/consumer design pattern (data)—those that produce data and those that consume the data. Data queues communicate data among the loops. The data queues also buffer data among the producer and consumer loops. Use the producer/consumer design pattern (data) to acquire multiple sets of data that must be processed in order, for example, a VI that accepts data while processing the data sets in the order they are received. Queuing (producing) the data occurs much faster than the data can be processed (consumed). The producer/consumer design pattern (data) queues the data in the producer loop and processes the data in the consumer loop as shown in Figure 2-8.

Figure 2-8. Producer/Consumer Design Pattern (Data)


This loop is the producer loop.
Condition that causes data to be enqueued.
Enqueue Data
True
Data to Enqueue
125
Releasing the queue stops the consumer loop(s).
data (can be any type)
Generate the data here.
stop
Dequeued Data
"Shutdown"
Wait(ms) in Consumer Loop
U32
This loop is a consumer loop.

The consumer loop processes the data at its own pace, while the producer loop continues to queue additional data. You also can use the producer/consumer design pattern (data) to create a VI to analyze network communication where two processes operate at the same time and at different speeds. The first process constantly polls the network line and retrieves packets. The second process analyzes the packets retrieved by the first process. The first process acts as the producer because it supplies data to the second process, which acts as the consumer. The parallel producer and consumer loops handle the retrieval and analysis of data off the network, and the queued communication between the two loops allows buffering of the network packets retrieved.

Figure 2-8 shows how you can use Synchronization VIs and functions to add functionality to the design pattern. Queues have the ability to transfer any data type. The data type transferred in Figure 2-8 is a string. A string is not the most efficient data type for passing data in design patterns. A more efficient data type for passing data in design patterns is a cluster consisting of state and data elements.

Producer/Consumer (Data) Demonstration

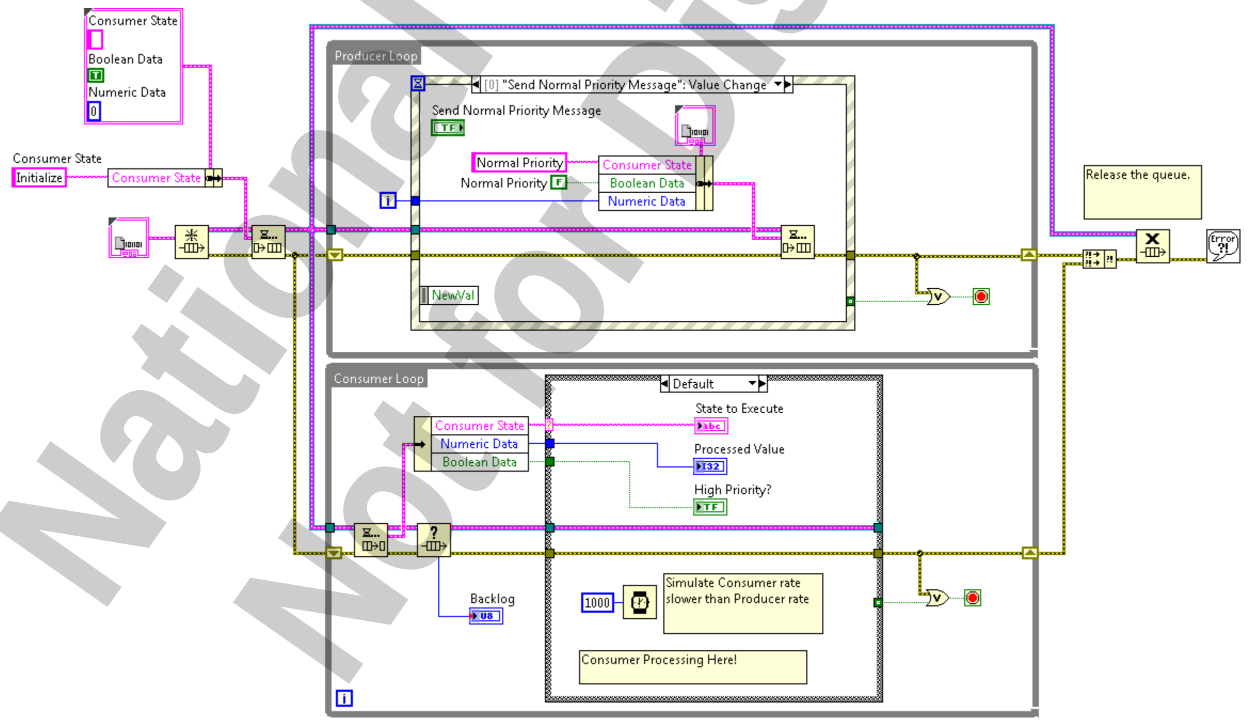
The Producer Consumer Data project, located in the `<Exercises>\LabVIEW Core 2\Demonstrations\Producer Consumer - Data` directory, demonstrates the design pattern. The front panel includes a button that controls when the Data to Enqueue sting value is added to the queue. The Wait (ms) in Consumer Loop slider simulates intensive processing time in the consumer loop. This demonstrates the effect on the overall program execution. While the consumer loop executes, the user can still enqueue more elements. The queue retains each element and executes them in order after the simulated processing of the element in the consumer loop. With this implementation, when the consumer loop is busy, the user interface remains responsive and user commands are recorded.

Producer/Consumer Design Pattern (Events)

One of the most versatile and flexible design patterns combines the producer/consumer and user interface event handler design patterns. A VI built using the producer/consumer design pattern (events) responds to the user interface asynchronously, allowing the user interface to continuously respond to the user. The consumer loop of this pattern responds as events occur, similar to the consumer loop of the producer/consumer design pattern (data).

The producer/consumer design pattern (events) uses the same implementation as the producer/consumer design pattern (data) except the producer loop uses an Event structure to respond to user interface events, as shown in Figure 2-9. The Event structure enables continuous response to user interaction.

Figure 2-9. Producer/Consumer Design Pattern (Events)



Producer/Consumer (Events) Demonstration

The Producer Consumer project, located in the `<Exercises>\LabVIEW Core 2\Producer Consumer - Event` directory, demonstrates the design pattern. The front panel includes buttons to enqueue either a normal priority message or a high priority message. When a Normal Priority Message is enqueued, the Consumer State is set to Normal Priority, the Boolean Data is set to False, and the Numeric Data value set to the While Loop iteration count. When a High Priority Message is enqueued, the Consumer State is set to High Priority, the Boolean Data is set to True, and the Numeric Data value set to `1000`. The State to Execute indicator shows the current state being executed in the Consumer Loop.

The consumer loop is implemented with a 1 second wait to demonstrate the effect of intensive processing time. While the consumer loop executes, the user can click other buttons. While the consumer loop is busy, the user interface remains responsive and user commands are recorded. Because the producer loop uses an Event structure, less processing occurs because the controls inside the Event structure are read only when their values change. The user interface is completely responsible for determining the actions taken in the consumer loop. Even if there is a backlog of normal priority queue elements, a high priority queue element is added to the front of the queue and is processed before normal priority messages.

When the user clicks on the Stop button, a Shutdown message is sent to the Consumer Loop which terminates the Consumer Loop by sending a True to the loop condition terminal.

Job Aid

Use Table 2-1 to determine the best uses for the design patterns described in this lesson.

Table 2-1. Design Pattern Comparisons

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

Table 2-1. Design Pattern Comparisons (Continued)

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 Not suitable for parallelism
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	Not suitable for parallelism
User Interface Event Handler	Processes messages from the user interface	Handles user interface messages	Does not allow for intensive processing Not suitable for sequencing
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

Table 2-1. Design Pattern Comparisons (Continued)

Design Pattern	Use	Advantage	Disadvantage
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

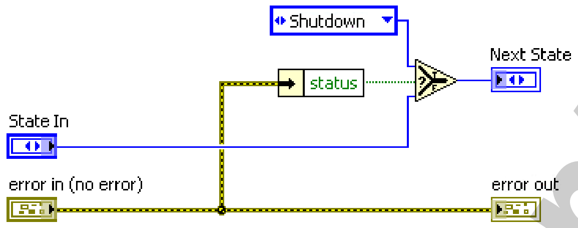
D. Error Handlers

By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box. Automatic error handling is convenient for quick prototypes and proof-of-concept development, but not recommended for professional application development. If you rely on automatic error handling your application might stop in a critical section of your code because of an error dialog box. The user might be unable to continue running the application or fix the problem.

By manually implementing error handling, you control when popup dialogs occur. If you plan to create a stand-alone application, you must incorporate manual error handling because LabVIEW does not display automatic error handling dialog boxes in the LabVIEW Run-Time Engine.

An error handler is a VI or code that changes the normal flow of the program when an error occurs. The Simple Error Handler VI is an example of a built-in error handler that is used in LabVIEW. You can implement other error handlers that are customized for your application. For example, you might choose to log error information to a file. Another common error handler is a VI that redirects code to a cleanup or shutdown routine when an error occurs so that your application exits gracefully. Figure 2-10 shows a state machine error handler that sets the next state to be the Shutdown state when an error in status is TRUE.

Figure 2-10. State Machine Error Handler



Handling Errors in the Producer/Consumer Design Pattern (Events)

The Producer/Consumer diagram shown in Figure 2-9 is useful for understanding the fundamental architecture of the design pattern. In addition to passing commands and data between loops, the design pattern shuts down both loops when the user clicks on the Stop button. However, if an error occurs in either loop, there is no mechanism for communicating this information to the other loop. Therefore, it is possible for one of the loops to execute indefinitely after the other stops.

Therefore, you need to add error handling code to the design pattern so that both the producer and consumer loops gracefully stop when an error occurs. There are many techniques for communicating error information between the two loops, some of which are covered in later courses. Some techniques involve adding an additional communication channel—such as an additional queue. Another technique is to use the existing queue cluster to execute a Shutdown case in the Consumer loop. This approach is demonstrated using the Error Handler VI shown in Figure 2-11 and the Producer/Consumer calling the Error Handler VI shown in Figure 2-12.

Figure 2-11. Error Handler VI

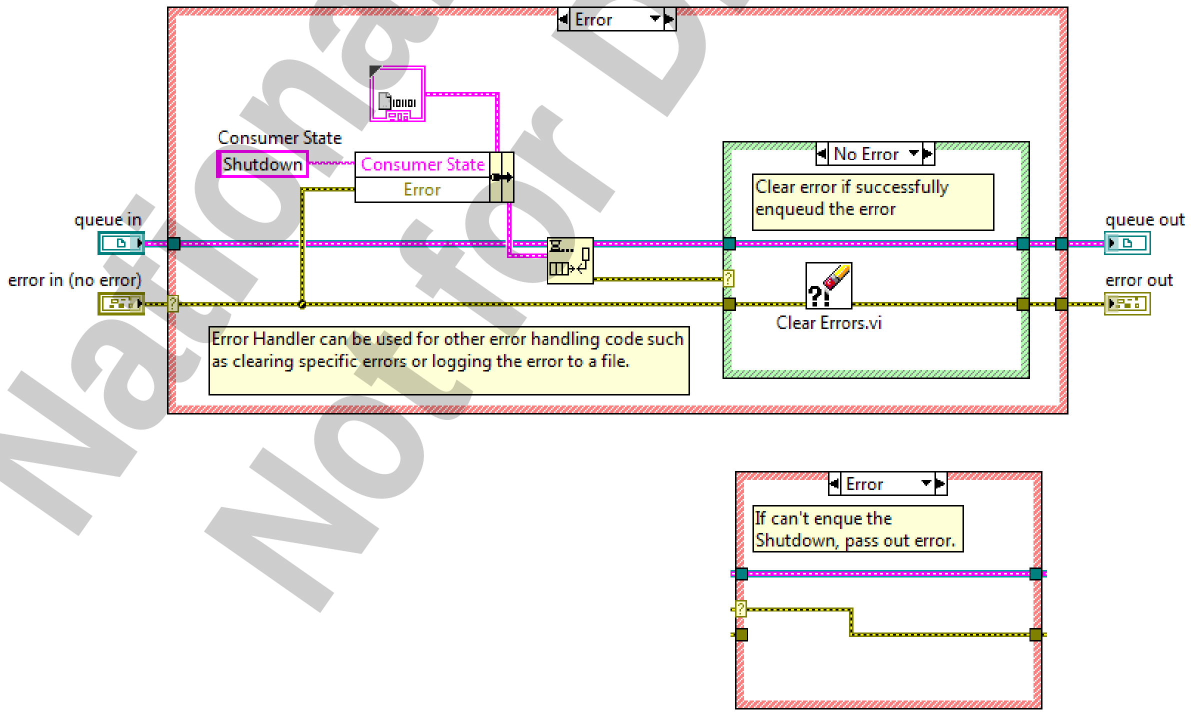
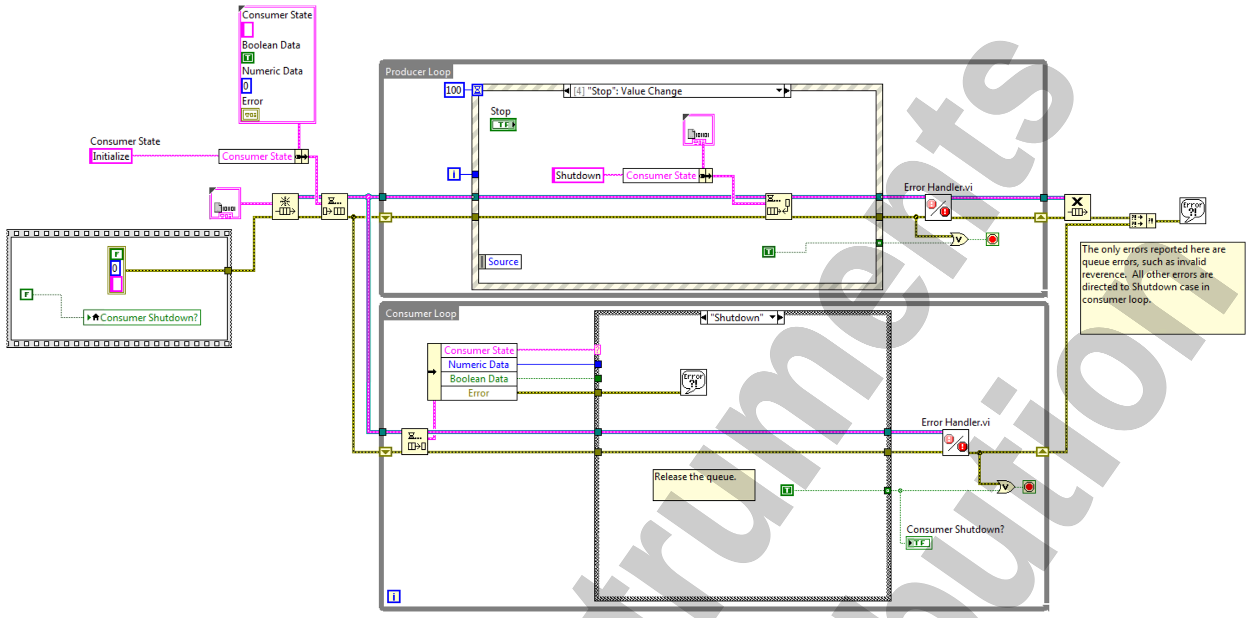

Error
Consumer State
Shutdown
Consumer State
Error
No Error
Clear error if successfully enqueud the error
queue in
error in (no error)
queue out
error out
Clear Errors.vi
Error Handler can be used for other error handling code such as clearing specific errors or logging the error to a file.
Error
If can't enque the Shutdown, pass out error.

Figure 2-12. Producer/Consumer Calling the Error Handler VI

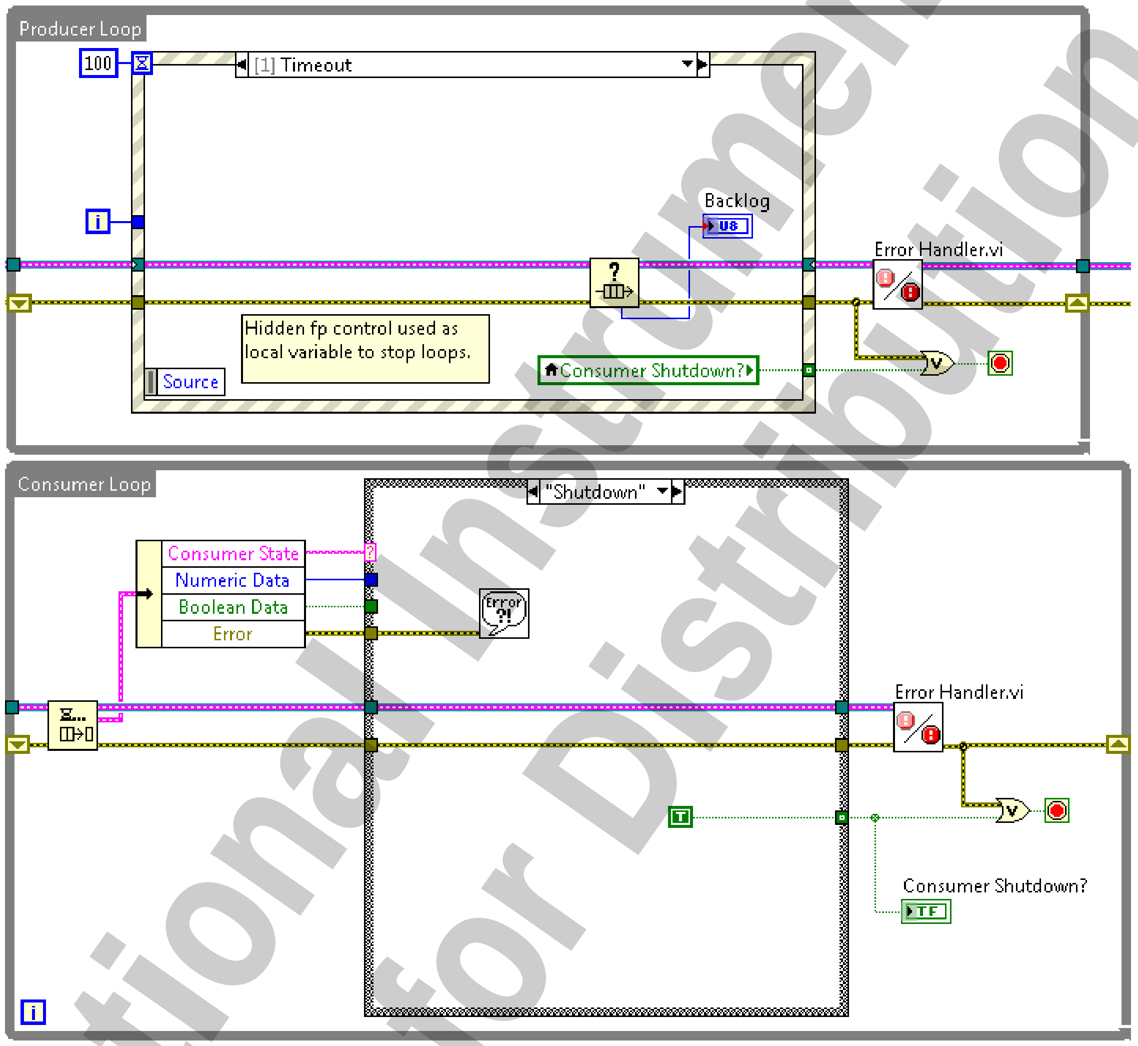


The only errors reported here are queue errors, such as invalid reverence. All other errors are directed to Shutdown case in consumer loop.

If an error occurs in the producer loop, the Error Handler VI enqueues the Shutdown state along with the error cluster information. Since you want to re-use this same Error Handler VI in the consumer loop, you need to clear the error in the error wire after enqueuing the information. Otherwise, the consumer loop would terminate before it had a chance to execute the Shutdown case. If the Enqueue function fails in the Error Handler VI, communication between the loops stops so the only reasonable course of action is to terminate immediately.

Because state processing, including shutdown processing, occurs in the consumer loop and not in the producer loop, you still need some mechanism for the consumer loop to tell the producer loop to stop executing. One approach, as shown in Figure 2-13, is to use a local variable that is polled in the Event timeout case. If an error occurs in the consumer loop, the Shutdown case executes and stops the consumer loop. The Consumer Shutdown? local variable then stops the producer loop.

Figure 2-13. Producer/Consumer Using Local Variable



E. Generating Error Codes and Messages

Error handling in LabVIEW follows the dataflow model. Just as data values flow through a VI, so can error information. As the VI runs, LabVIEW tests for errors at each execution node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing that part of the code. The next node does the same thing, and so on. At the end of the execution flow, the error is typically reported via the Simple Error Handler.vi or through an Error Out cluster.

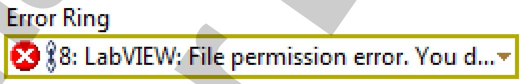
Setting an error when a node or VI fails should not be limited to errors that LabVIEW functions and VIs report. As a VI developer, you should also be detecting error conditions and reporting errors in your subVIs. There are many situations where you might want to report an error condition that you detected with your code. Below are a few examples for setting or overriding error codes and messages:

- Check for invalid inputs to subVIs or algorithms. Check for empty arrays or empty strings before processing. If an input is invalid, set an appropriate error code and message. An example might be checking to see if an input is a positive value before attempting to take the square root of the number. If the value is negative, generate an error with an appropriate error message as to how the user might fix the input value.
- Check for invalid outputs of various algorithms. For example, the Search 1D Array function returns an index value of -1 if the search element is not found in the array. In this situation, you might want to report an error with a message indicating the element value that wasn't found in the array.
- Overwrite LabVIEW error messages with more specific details. The Open/Create/Replace File function returns error code 7 when attempting to open a file that doesn't exist. The message associated with error code 7 is a generic "file not found" error message. Instead of the generic error message, you might want the error message to be more specific noting the specific filepath that failed and how the user might rectify the problem and retry the operation.

Error Ring

Use the Error Ring to quickly select and pass NI or custom error codes throughout your VI. You can configure the ring to return a built-in error message or you can create a custom error message for a one-time use. By default, the source string of the error cluster contains the call chain from the top-level VI to the current VI. Figure 2-14 shows a configured error ring.

Figure 2-14. Error Ring



After you select the error, you can change the type (Error or Warning) and whether to include the call chain by clicking the icons on the ring. You also can toggle the error type and call chain options by right-clicking the Error Ring and selecting Generate Error, Generate Warning, Include Call Chain, or Exclude Call Chain from the shortcut menu.

Defining a Custom Error Code

Creating a custom error code is useful if you want to define a single error code or overwrite a single built-in error code. If you have multiple custom errors you want to use in the Error Ring, use the Error Code Editor dialog box.

F. Timing a Design Pattern

This section discusses two forms of timing—execution timing and software control timing. Execution timing uses timing functions to give the processor time to complete other tasks. Software control timing involves timing a real-world operation to perform within a set time period.

Execution Timing

Execution timing involves timing a design pattern explicitly or based on events that occur within the VI. Explicit timing uses a function that specifically allows the processor time to complete other tasks, such as the Wait Until Next ms Multiple function. When timing is based on events, the design pattern waits for some action to occur before continuing and allows the processor to complete other tasks while it waits.

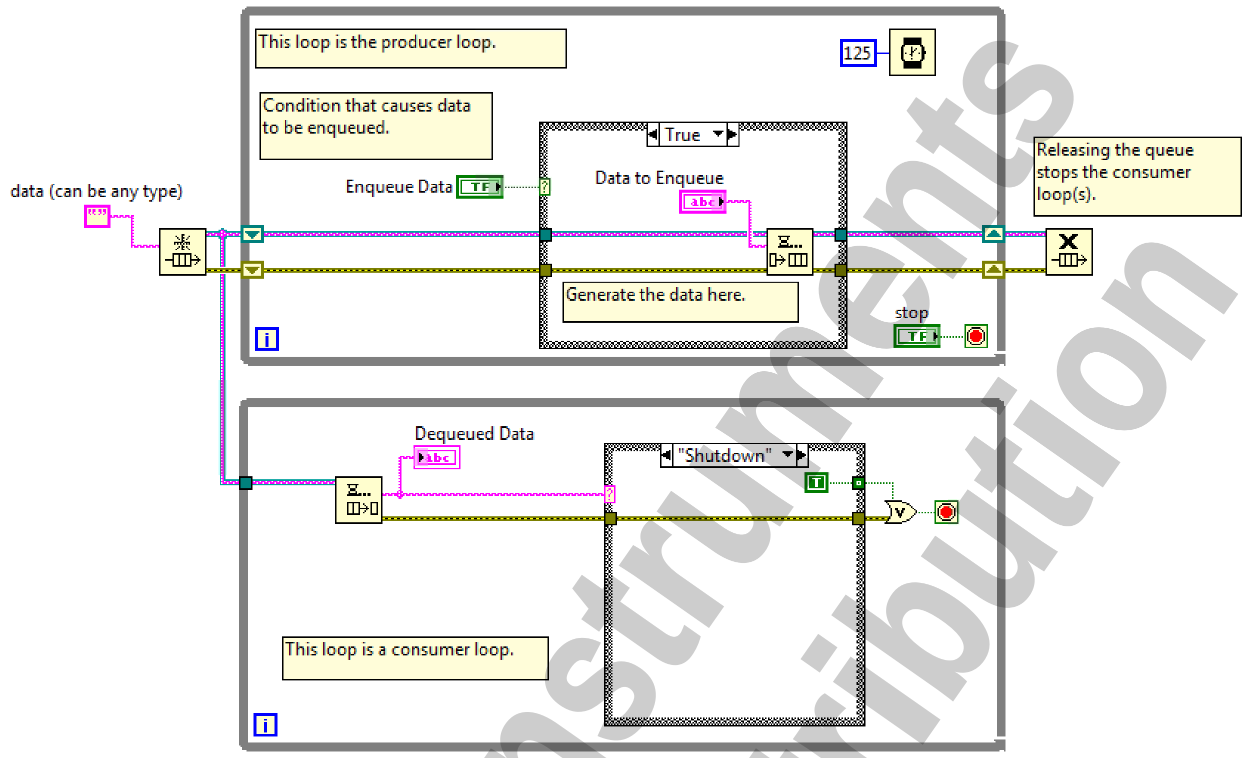
Use explicit timing for polling-based design patterns such as the producer/consumer design pattern (data) or the polling-based state machine.

Tip Polling is the process of making continuous requests for data from another device. In LabVIEW, this generally means that the block diagram continuously asks if there is data available, usually from the user interface.

For example, the producer/consumer design pattern (data) shown in Figure 2-15 uses a While Loop and a Case structure to implement the producer loop. The producer executes continuously and polls for an event of some type, such as the user clicking a button. When the event occurs, the producer enqueues a message to the consumer. You need to time the producer so it does not take over the execution of the processor. In this case, you typically use the Wait (ms) function to regulate how frequently the master polls.

Tip Always use a timing function such as the Wait (ms) function or the Wait Until Next ms Multiple function in any design pattern that continually executes and needs to be regulated. If you do not use a timing function in a continuously executing structure, LabVIEW uses all the processor time, and background processes may not run.

Figure 2-15. Producer/Consumer Design Pattern

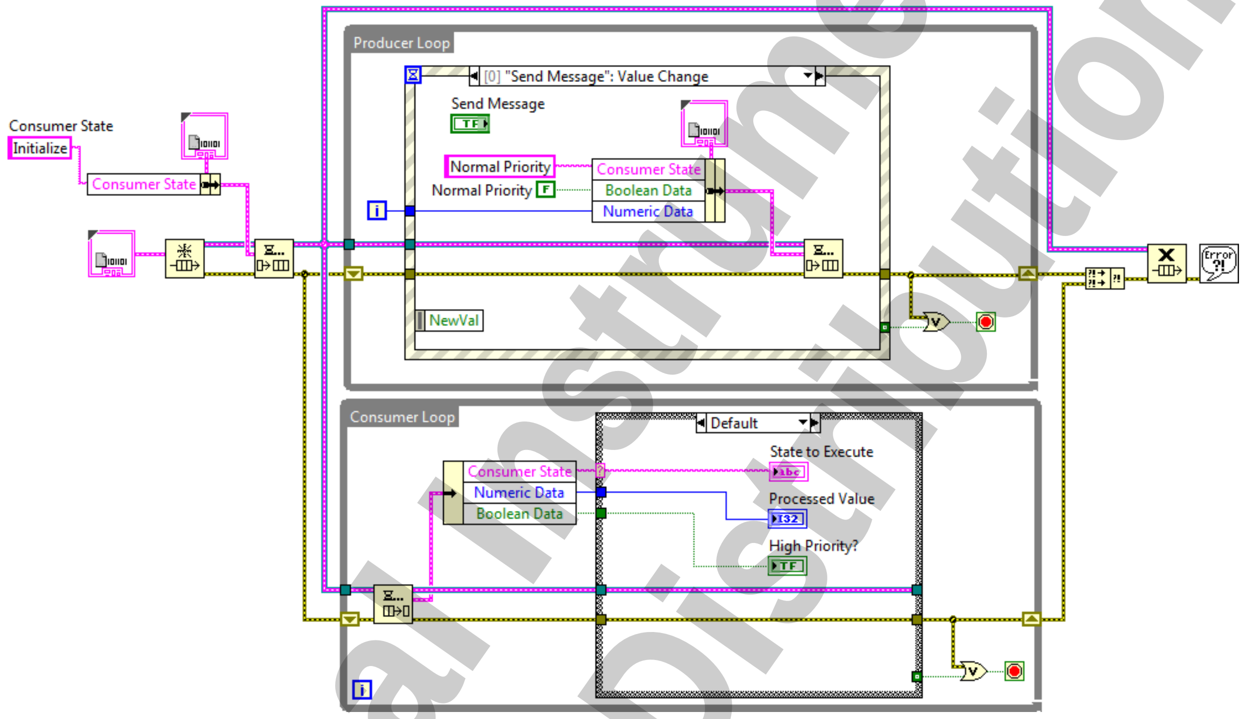

This loop is the producer loop.
Condition that causes data to be enqueued.
125
True
Data to Enqueue
Enqueue Data
data (can be any type)
Releasing the queue stops the consumer loop(s).
Generate the data here.
stop
Dequeued Data
"Shutdown"
This loop is a consumer loop.

Notice that the consumer loop does not contain any form of timing. The use of queues to pass data and messages provides an inherent form of timing in the consumer loop because the consumer loop waits for the Queue function to receive an enqueued element. After the Queue function receives a queued element, the consumer loop executes on the data or message. This creates an efficient block diagram that does not waste processor cycles by polling for messages. This is an example of execution timing by waiting for an event.

When you implement design patterns where the timing is based on the occurrence of events, you can use event mechanisms for both execution timing and software control timing. Synchronization functions and the Event structure both include a Timeout feature. By default, the timeout value is –1 which indicates an infinite wait. With an infinite timeout, execution of the design pattern only occurs when an event occurs.

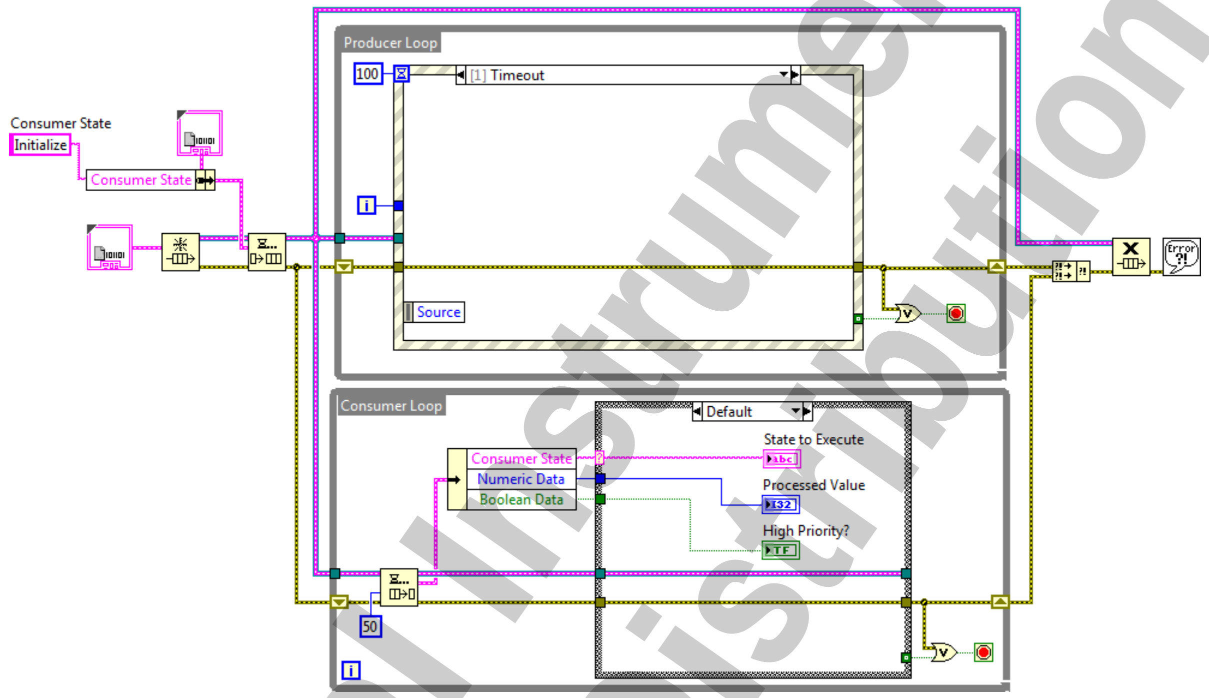
The Producer/Consumer Design Pattern (Events) VI shown in Figure 2-16 does not require any timing functions because the timing is inherent in the implementation of the producer loop and the consumer loop. The Event structure in the producer loop controls when the producer loop executes. The Dequeue Element function in the consumer loop waits until an item is placed in the queue, thus controlling the execution of the consumer loop. Design patterns such as the producer/consumer design pattern (events) do not require any timing because their timing is controlled by external events.

Figure 2-16. Producer/Consumer Design Pattern (Events) with Indefinite Wait



Specifying a timeout value allows functionality to be executed at regular intervals. For example, the producer/consumer design pattern (events) shown in Figure 2-17 demonstrates how you can execute code at regular intervals in both the producer loop and the consumer loop. In Figure 2-17, the Producer loop executes every 100 ms even if no other events occur. Wiring a millisecond value to the timeout terminal of an Event structure wakes the Event structure and executes the code in the Timeout case. The consumer loop executes every 50 ms even if the queue is empty.

Figure 2-17. Timeout Event Case in Producer/Consumer Design Pattern with Timed Execution

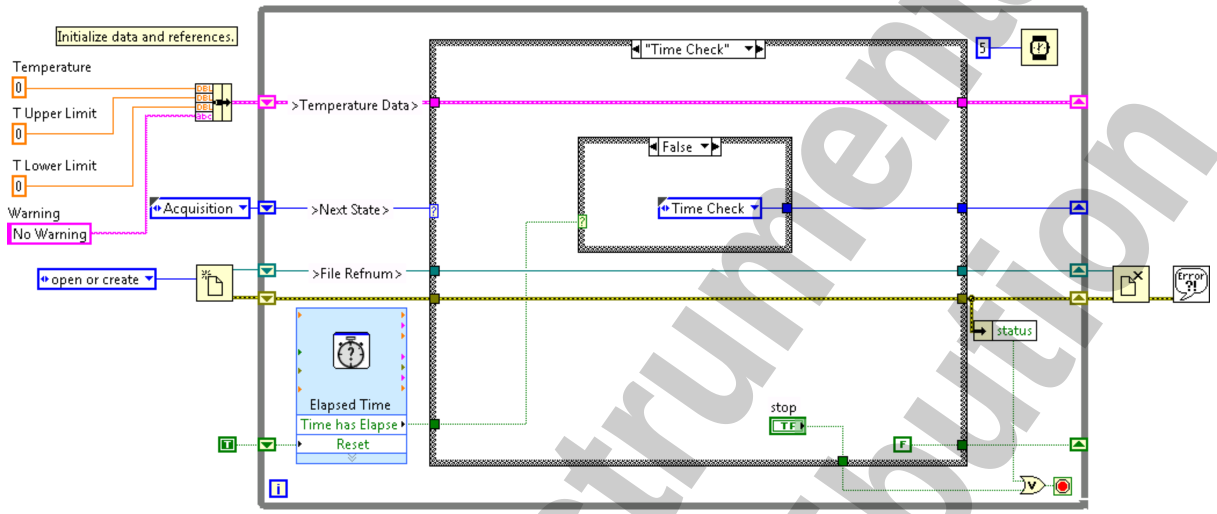


Software Control Timing

Many applications that you create must execute an operation for a specified amount of time. Consider implementing a state machine design pattern for a temperature data acquisition system. If the specifications require that the system acquire temperature data for 5 minutes, you could remain in the acquisition state for 5 minutes. However, during that time you cannot process any user interface actions such as stopping the VI. To process user interface actions, you must implement timing so that the VI continually executes for specified time. Implementing this type of timing involves keeping the application executing while monitoring a real-time clock.

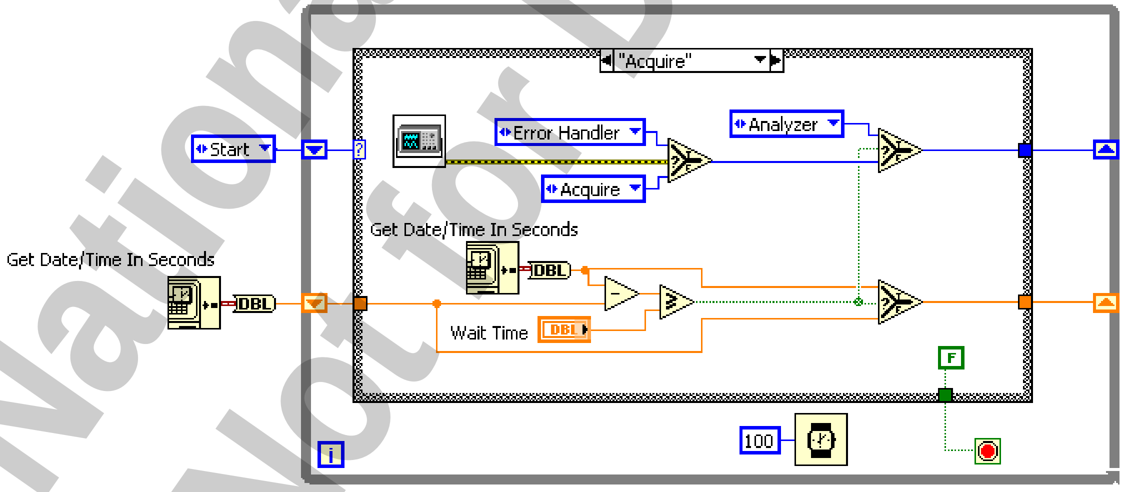
In the *LabVIEW Core 1* course, you implemented software control timing to monitor the time until the VI should acquire the next piece of data, as shown in Figure 2-18. Notice the use of the Elapsed Time Express VI to keep track of a clock.

Figure 2-18. Use of the Elapsed Time Express VI



If you use the Wait (ms) function or the Wait Until Next ms Multiple function to perform software timing, the execution of the function you are timing does not occur until the wait function finishes. These timing functions are not the preferred method for performing software control timing, especially for VIs where the system must continually execute. A better method for software control timing utilizes the Get Date/Time In Seconds function to get the current time and track it using shift registers.

Figure 2-19. Software Timing Using the Get Date/Time In Seconds Function



The Get Date/Time In Seconds function, connected to the left terminal of the shift register, initializes the shift register with the current system time. Each state uses another Get Date/Time In Seconds function and compares the current time to the start time. If the difference in these two times is greater or equal to the wait time, the state finishes executing and the rest of the application executes.

Tip Always use the Get Date/Time In Seconds function instead of the Tick Count (ms) function for this type of comparison because the value of the Tick Count (ms) function can rollover to 0 during execution.

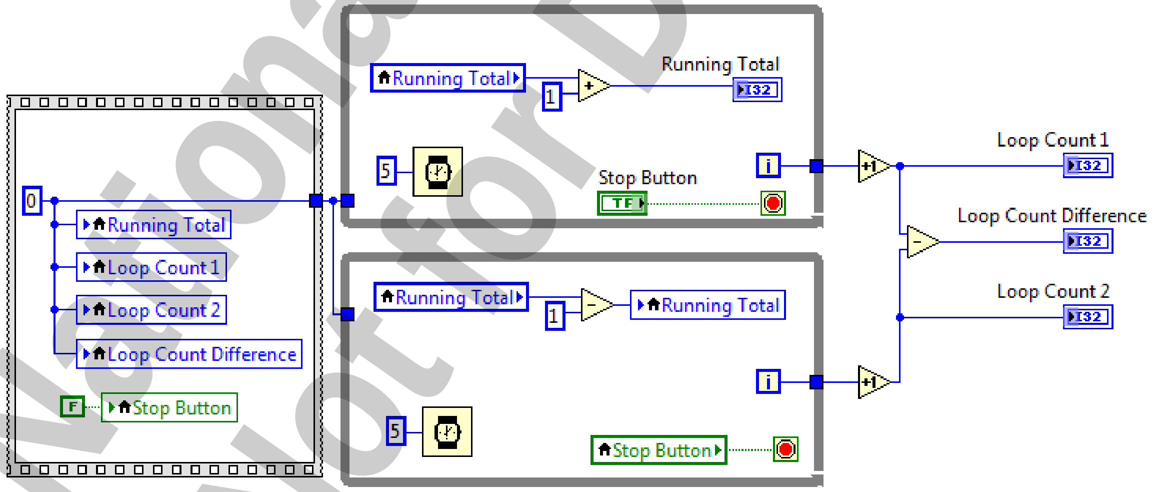
G. Functional Global Variable Design Pattern

As you continue to develop your LabVIEW skills and interact with other developers, you will learn about other design patterns with names like Master/Slave and Queued Message Handler. Some design patterns, like the Actor Framework, are quite advanced and require additional knowledge of LabVIEW's object-oriented design features. Some other design patterns are useful components in larger applications. The Functional Global Variable is one such design pattern. Before you can understand this new design pattern, it is good to have an in depth understanding of a "read-modify-write" type of race condition.

Read-Modify-Write Race Conditions

A race condition occurs when the timing of events or the scheduling of tasks unintentionally affects an output or data value. Race conditions are a common problem for programs that execute multiple tasks in parallel and share data between them. Consider the example in Figure 2-20.

Figure 2-20. Race Condition Example



Both loops modify a local variable during each loop iteration. One loop increments the counter while the other loop decrements the counter. If you run this VI, the expected result after clicking the **Stop** button is that the **Running Total** is equal to **Loop Count Difference**. Since both loops are running at the same speed, the value should be zero or close to zero. Depending on when you stop the VI, it is possible you will see the expected result. However, it is also likely that the **Running Total** will not equal **Loop Count Difference** because this VI contains a race condition.

On a single processor computer, actions in a multi-tasking program like this example actually happen sequentially, but LabVIEW and the operating system rapidly switch tasks so that the tasks effectively execute at the same time. The race condition in this example occurs when the switch from one task to the other occurs at a certain time. Notice that both of the loops perform the following operations:

- Read the local variable
- Increment or decrement the value read
- Write the modified value to the local variable

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

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

In this example, the increment of the first loop is effectively overwritten by Loop 2. If the local variable started out with a value zero, the resulting value would be minus one. This generates a race condition, which can cause serious problems if you intend the program to calculate an exact difference.

In this particular example, there are few instructions between when the local variable is read and when it is written. Therefore, the VI is less likely to switch between the loops at the wrong time. This explains why this VI might run accurately for short periods and only loses a few counts for longer periods.

Race conditions are difficult to identify and debug, because the outcome depends upon the order in which the operating system executes scheduled tasks and the timing of external events. The way tasks interact with each other and the operating system, as well as the arbitrary timing of external events, make this order essentially random. Often, code with a race condition can return the same result thousands of times in testing, but still can return a different result, which can appear when the code is in use.

The best way to avoid race conditions is by using the following techniques:

- Controlling and limiting shared resources.
- Identifying and protecting critical sections within your code.
- Specifying execution order.

Controlling and Limiting Shared Resources

Race conditions are most common when two tasks have both read and write access to a resource, as is the case in the previous example. A resource is any entity that is shared between the processes. When dealing with race conditions, the most common shared resources are data storage, such as variables. Other examples of resources include files and references to hardware resources.

Allowing a resource to be altered from multiple locations often introduces the possibility for a race condition. Therefore, an ideal way to avoid race conditions is to minimize shared resources and the number of writers to the remaining shared resources. In general, it is not harmful to have multiple readers or monitors for a shared resource. However, try to use only one writer or controller for a shared resource. Most race conditions only occur when a resource has multiple writers.

In the previous example, you can reduce the dependency upon shared resources by having each loop maintain its count locally. Then, share the final counts after clicking the **Stop** button. This involves only a single read and a single write to a shared resource and eliminates the possibility of a race condition. If all shared resources have only a single writer or controller, and the VI has a well sequenced instruction order, then race conditions do not occur.

Protecting Critical Sections

A critical section of code is code that must behave consistently in all circumstances. When you use multi-tasking programs, one task may interrupt another task as it is running. In nearly all modern operating systems, this happens constantly. Normally, this does not have any effect upon running code, however, when the interrupting task alters a shared resource that the interrupted task assumes is constant, then a race condition occurs.

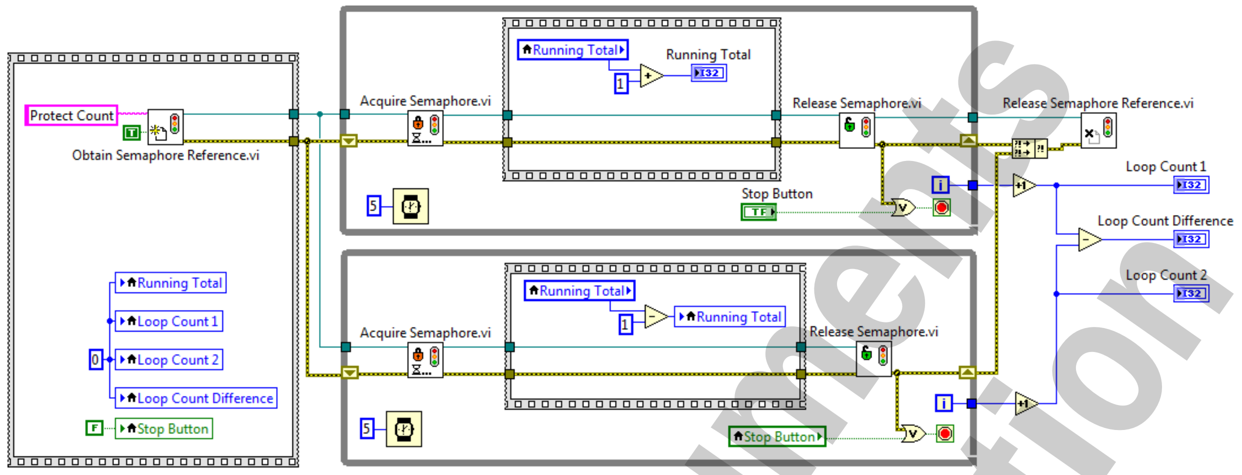
Each loop in Figure 2-20 contains a critical code section. If one of the loops interrupts the other loop while it is executing the code in its critical section, then a race condition can occur. One way to eliminate race conditions is to identify and protect the critical sections in your code. There are many techniques for protecting critical sections. Two of the most effective are functional global variables and semaphores.

Semaphores

Semaphores are synchronization mechanisms specifically designed to protect resources and critical sections of code. You can prevent critical sections of code from interrupting each other by enclosing each between an Acquire Semaphore and Release Semaphore VI. By default, a semaphore only allows one task to acquire it at a time. Therefore, after one of the tasks enters a critical section, the other tasks cannot enter their critical sections until the first task completes. When done properly, this eliminates the possibility of a race condition.

You can use semaphores to protect critical sections of code in Figure 2-20. To remove the race condition, open the semaphore before starting each loop. Within each loop, use the Acquire Semaphore VI just before the critical section and use the Release Semaphore VI after the critical section. Figure 2-21 shows a solution to the race condition using semaphores.

Figure 2-21. Protecting Critical Sections with a Semaphore



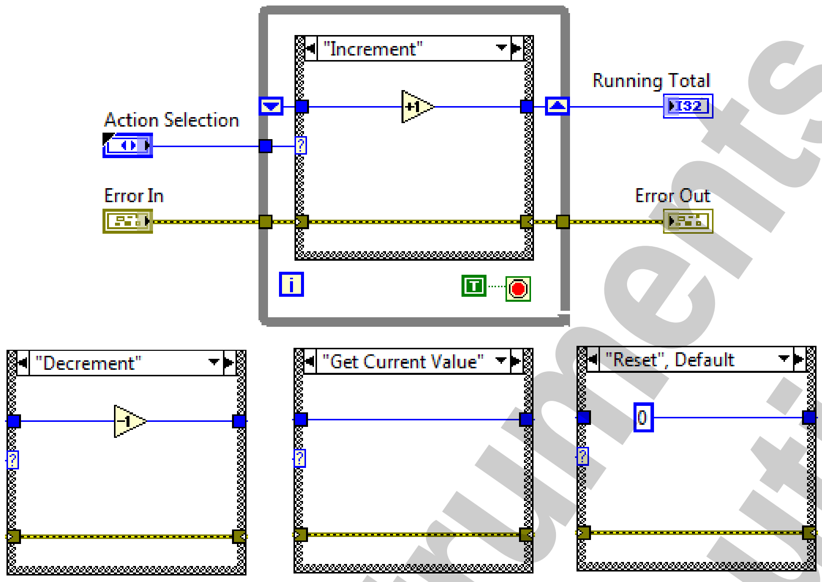
Functional Global Variables

One way to protect critical sections is to place them in subVIs. You can only call a non-reentrant subVI from one location at a time. Therefore, placing critical code in a non-reentrant subVI keeps the code from being interrupted by other processes calling the subVI. Using the functional global variable architecture to protect critical sections is particularly effective, because shift registers can replace less protected storage methods like global or single-process shared variables. Functional global variables also encourage the creation of multi-functional subVIs that handle all tasks associated with a particular resource.

After you identify each section of critical code in your VI, group the sections by the resources they access, and create one functional global variable for each resource. Critical sections performing different operations each can become a command for the functional global variable, and you can group critical sections that perform the same operation into one command, thereby re-using code.

You can use functional global variables to protect critical sections of code in Figure 2-20. To remove the race condition, replace the local variables with a functional global variable and place the code to modify the counter within the functional global variable, as shown in Figure 2-22.

Figure 2-22. Using Functional Global Variables to Protect Sections of Code

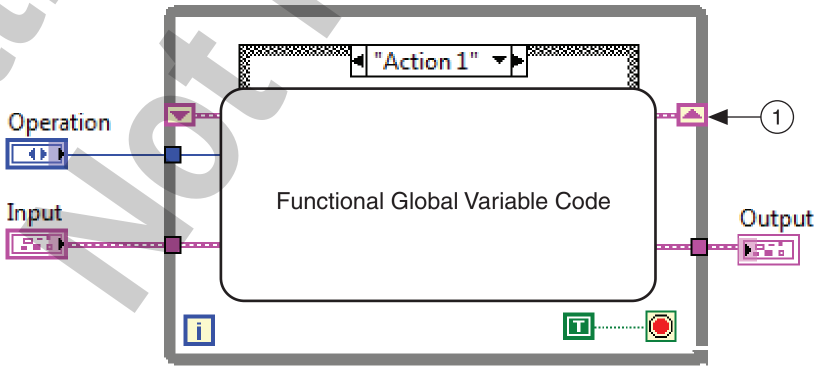


Functional Global Variables

A functional global variable is a non-reentrant VI that uses unitialized shift registers to hold global data. The VI often allows for actions to be performed on the data. When a VI is non-reentrant, there is one data space for the VI. Therefore, only one caller can be running the VI at any given time. Other callers have to “wait their turn” to use the VI.

You can use uninitialized shift registers in For or While Loops to hold data as long as the VI never goes out of memory. The shift register holds the last state of the shift register. A loop with an uninitialized shift register is known as a functional global variable. The advantage of a functional global variable over a global variable is that you can control access to the data in the shift register. Also, the functional global variable eliminates the possibility of race conditions because only one instance of a functional global variable can be loaded into memory at a time. The general form of a functional global variable VI includes a Case structure and an uninitialized shift register with a single iteration While Loop, as shown in Figure 2-23.

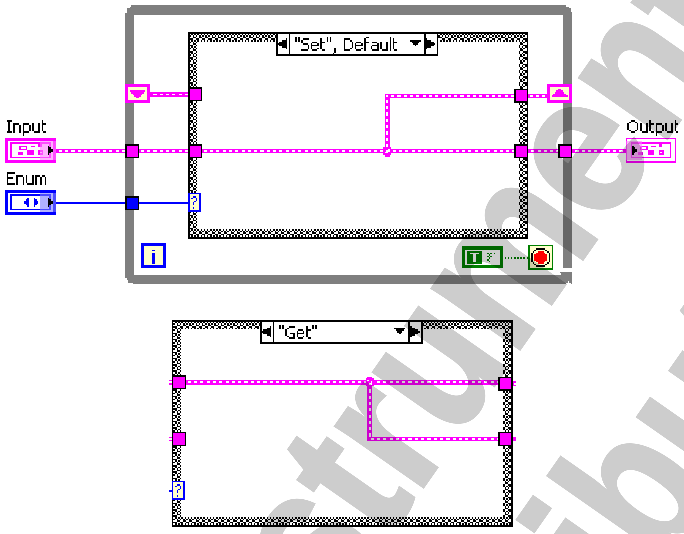
Figure 2-23. Functional Global Variable Framework



1 Uninitialized Shift Register

A functional global variable usually has an **action** input parameter that specifies which task the VI performs. The VI uses an uninitialized shift register in a While Loop to hold the result of the operation. Figure 2-24 shows a simple functional global variable with set and get functionality.

Figure 2-24. Functional Global Variable with Set and Get Functionality



In this example, data passes into the VI and is stored in the shift register if the enumerated data type control is configured to `Set`. Data is retrieved from the shift register if the enumerated data type control is configured to `Get`.

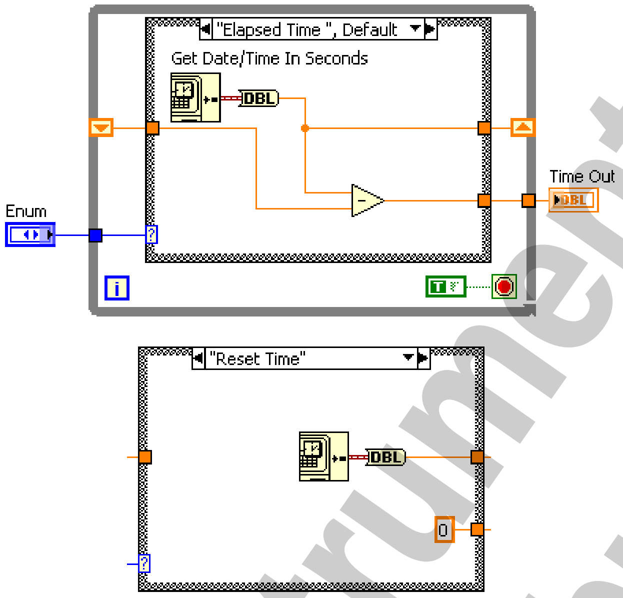
Tip Before you use a local or global variable, make sure a functional global variable would not have worked instead.

Although you can use functional global variables to implement simple global variables, as shown in the previous example, they are especially useful when implementing more complex data structures, such as a stack or a queue buffer. You also can use functional global variables to protect access to global resources, such as files, instruments, and data acquisition devices, that you cannot represent with a global variable.

Using Functional Global Variables for Timing

One powerful application of functional global variables is to perform timing in your VI. Many VIs that perform measurement and automation require some form of timing. Often an instrument or hardware device needs time to initialize, and you must build explicit timing into your VI to take into account the physical time required to initialize a system. You can create a functional global variable that measures the elapsed time between each time the VI is called, as shown in Figure 2-25.

Figure 2-25. Elapsed Time Functional Global Variable



The Elapsed Time case gets the current date and time in seconds and subtracts it from the time that is stored in the shift register. The Reset Time case initializes the functional global variable with a known time value.

Self-Review: Quiz

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

3. What is the default timeout value of an Event structure?
   a. 0
   b. 100 ms
   c. Never time out
   d. The input value of the Wait (ms) function that exists in the same loop as the Event structure

Self-Review: Quiz 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**

3. What is the default timeout value of an Event structure?
   - a. 0
   - b. 100 ms
   - **c. Never time out**
   - d. The input value of the Wait (ms) function that exists in the same loop as the Event structure

Notes

3

Controlling the User Interface

When writing programs, often you must change the attributes of front panel objects programmatically. For example, you may want to make an object invisible until a certain point in the execution of the program. In LabVIEW, you can use VI Server to access the properties and methods of front panel objects. This lesson explains the Property Nodes, Invoke Nodes, VI Server, and control references.

Topics

A. VI Server Architecture
B. Property Nodes
C. Invoke Nodes
D. Control References

A. VI Server Architecture

The VI Server is an object-oriented, platform-independent technology that provides programmatic access to LabVIEW and LabVIEW applications. VI Server performs many functions; however, this lesson concentrates on using the VI Server to control front panel objects and edit the properties of a VI and LabVIEW. To understand how to use VI Server, it is useful to understand the terminology associated with it.

Object-Oriented Terminology

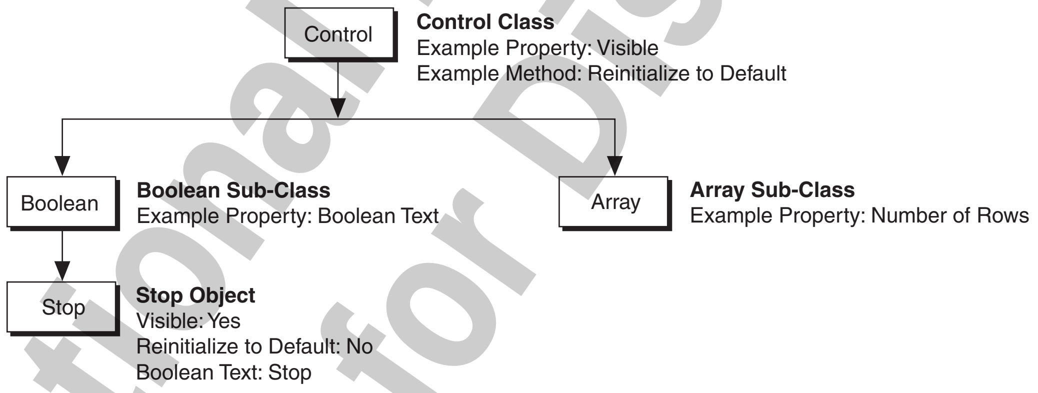
Object-oriented programming is based on objects. An *object* is a member of a class. A *class* defines what an object is able to do, what operations it can perform (methods), and what properties it has, such as color, size, and so on.

Objects can have methods and properties. *Methods* perform an operation, such as reinitializing the object to its default value. *Properties* are the attributes of an object. The properties of an object could be its size, color, visibility, and so on.

Control Classes

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, as shown in Figure 3-1.

Figure 3-1. Boolean Class Example


Control
Control Class
Example Property: Visible
Example Method: Reinitialize to Default
Boolean
Boolean Sub-Class
Example Property: Boolean Text
Array
Array Sub-Class
Example Property: Number of Rows
Stop
Stop Object
Visible: Yes
Reinitialize to Default: No
Boolean Text: Stop

VI Class

Controls are not the only objects in LabVIEW to belong to a class. A VI belongs to the VI Class and has its own properties and methods associated with it. For instance, you can use VI class methods to abort a VI, to adjust the position of the front panel window, and to get an image of the block diagram. You can use VI class properties to change the title of a front panel window, to retrieve the size of the block diagram, and to hide the **Abort** button.

B. Property Nodes

Property Nodes access the properties of an object. In some applications, you might want to programmatically modify the appearance of front panel objects in response to certain inputs. For example, if a user enters an invalid password, you might want a red LED to start blinking. Another example is changing the color of a trace on a chart. When data points are above a certain value, you might want to show a red trace instead of a green one. Property Nodes allow you to make these modifications programmatically. You also can use Property Nodes to resize front panel objects, hide parts of the front panel, add cursors to graphs, and so on.

Property Nodes in LabVIEW are very powerful and have many uses. Refer to the *LabVIEW Help* for more information about Property Nodes.

Creating Property Nodes

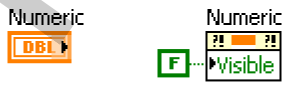
When you create a property from a front panel object by right-clicking the object, selecting **Create»Property Node**, and selecting a property from the shortcut menu, LabVIEW creates a Property Node on the block diagram that is implicitly linked to the front panel object. If the object has a label, the Property Node has the same label. You can change the label after you create the node. You can create multiple Property Nodes for the same front panel object.

Using Property Nodes

When you create a Property Node, it initially has one terminal representing a property you can modify for the corresponding front panel object. Using this terminal on the Property Node, you can either set (write) the property or get (read) the current state of that property.

For example, if you create a Property Node for a digital numeric control using the Visible property, a small arrow appears on the right side of the Property Node terminal, indicating that you are reading that property value. You can change the action to write by right-clicking the terminal and selecting **Change To Write** from the shortcut menu. Wiring a False Boolean value to the Visible property terminal causes the numeric control to vanish from the front panel when the Property Node receives the data. Wiring a True Boolean value causes the control to reappear.

Figure 3-2. Using Property Nodes



To get property information, right-click the node and select **Change All to Read** from the shortcut menu. To set property information, right-click the node and select **Change All to Write** from the shortcut menu. If a property is read only, **Change to Write** is dimmed in the shortcut menu. If the small direction arrow on the Property Node is on the right, you are getting the property value. If the small direction arrow on a Property Node is on the left, you are setting the property value. If the Property Node in Figure 3-2 is set to Read, when it executes it outputs a True value if the control is visible or a False value if it is invisible.

Tip Some properties are read-only, such as the Label property, or write only, such as the Value (Signaling) property.

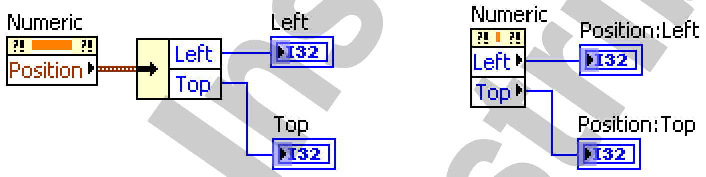
To add terminals to the node, right-click the white area of the node and select **Add Element** from the shortcut menu or use the Positioning tool to resize the node. Then, you can associate each Property Node terminal with a different property from its shortcut menu.

Tip Property Nodes execute each terminal in order from top to bottom.

Some properties use clusters. These clusters contain several properties that you can access using the cluster functions. Writing to these properties as a group requires the Bundle function and reading from these properties requires the Unbundle function. To access bundled properties, select **All Elements** from the shortcut menu. For example, you can access all the elements in the Position property by selecting **Properties»Position»All Elements** from the shortcut menu.

However, you also can access the elements of the cluster as individual properties, as shown in Figure 3-3.

Figure 3-3. Properties Using Clusters



C. Invoke Nodes

Invoke Nodes access the methods of an object.

Use the Invoke Node to perform actions, or methods, on an application or VI. Unlike the Property Node, a single Invoke Node executes only a single method on an application or VI. Select a method by using the Operating tool to click the method terminal or by right-clicking the white area of the node and selecting **Methods** from the shortcut menu. You also can create an implicitly linked Invoke Node by right-clicking a front panel object, selecting **Create»Invoke Node**, and selecting a method from the shortcut menu.

The name of the method is always the first terminal in the list of parameters in the Invoke Node. If the method returns a value, the method terminal displays the return value. Otherwise, the method terminal has no value.

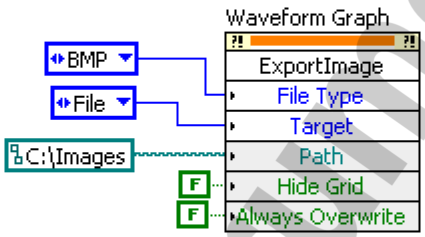
The Invoke Node lists the parameters from top to bottom with the name of the method at the top and the optional parameters, which are dimmed, at the bottom.

Example Methods

An example of a method common to all controls is the Reinitialize to Default method. Use this method to reinitialize a control to its default value at some point in your VI. The VI class has a similar method called Reinitialize All to Default.

Figure 3-4 is an example of a method associated with the Waveform Graph class. This method exports the waveform graph image to the clipboard or to a file.

Figure 3-4. Invoke Node for the Export Image Method



D. Control References

A Property Node created from the front panel object or block diagram terminal is an implicitly linked Property Node. This means that the Property Node is linked to the front panel object. What if you must place your Property Nodes in a subVI? Then the objects are no longer located on the front panel of the VI that contains the Property Nodes. In this case, you need an explicitly linked Property Node. You create an explicitly linked Property Node by wiring a reference to a generic Property Node.

If you are building a VI that contains several Property Nodes or if you are accessing the same property for several different controls and indicators, you can place the Property Node in a subVI and use control references to access that node. A control reference is a reference to a specific front panel object.

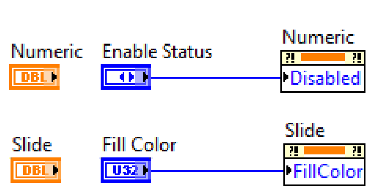
This section shows one way to use control references. Refer to the *Controlling Front Panel Objects* topic of the *LabVIEW Help* for more information about control references.

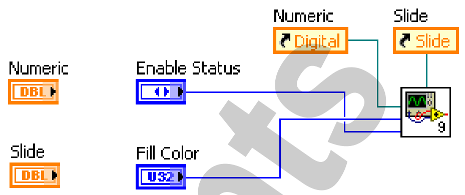
Creating a SubVI with Property Nodes

As shown in Figure 3-5, the simplest way to create explicitly linked Property Nodes is to complete the following steps:

1. Create your VI.
2. Select the portion of the block diagram that is in the subVI, as shown in the first part of Figure 3-5.
3. Select **Edit»Create SubVI**. LabVIEW automatically creates the control references needed for the subVI.
4. Customize and save the subVI. As you can see in the second part of Figure 3-5, the subVI uses the default icon.

Figure 3-5. Using Edit»Create SubVI to Create Control References

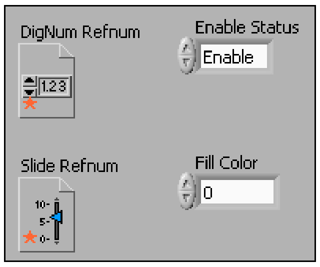


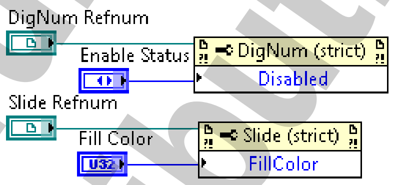


Objects Selected to Create SubVI | Edit»Create SubVI used

Figure 3-6 shows the subVI created. Notice that the front panel Control Refnum controls have been created and connected to a Property Node on the block diagram.

Figure 3-6. Sub VI Created Using Edit»Create SubVI





Created SubVI Front Panel | Created SubVI Block Diagram

Note A red star on the Control Reference control indicates that the refnum is strictly typed. Refer to the *Strictly Typed and Weakly Typed Control Refnums* section of the *Controlling Front Panel Objects* topic of the *LabVIEW Help* for more information about weakly and strictly typed control references.

Creating Control References

To create a control reference for a front panel object, right-click the object or its block diagram terminal and select **Create»Reference** from the shortcut menu.

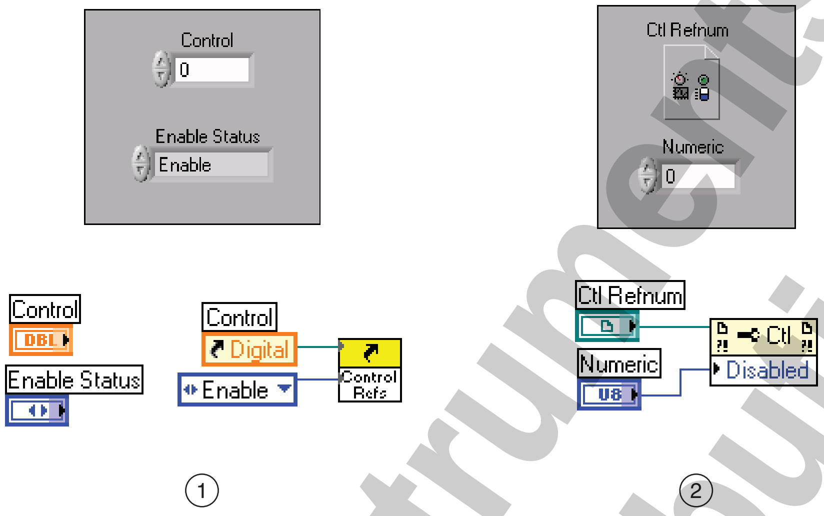
You can wire this control reference to a generic Property Node. You can pass the control reference to a subVI using a control refnum terminal.

Using Control References

Setting properties with a control reference is useful for setting the same property for multiple controls. Some properties apply to all classes of controls, such as the Disabled property. Some properties are only applicable to certain control classes, such as the Lock Boolean Text in Center property.

The following example shows how to construct a VI that uses a control reference on the subVI to set the Enable/Disable state of a control on the main VI front panel.

Figure 3-7. Control References



1 Main VI
2 SubVI

The main VI sends a reference for the digital numeric control to the subVI along with a value of zero, one, or two from the enumerated control. The subVI receives the reference by means of the **Ctl Refnum** on its front panel window. Then, the reference is passed to the Property Node. Because the Property Node now links to the numeric control in the main VI, the Property Node can change the properties of that control. In this case, the Property Node manipulates the enabled/disabled state.

Notice the appearance of the Property Node in the block diagram. You cannot select a property in a generic Property Node until the class is chosen. The class is chosen by wiring a reference to the Property Node. This is an example of an explicitly linked Property Node. It is not linked to a control until the VI is running and a reference is passed to the Property Node. The advantage of this type of Property Node is its generic nature. Because it has no explicit link to any one control, it may be reused for many different controls. This generic Property Node is available on the **Functions** palette.

Selecting the VI Server Class

When you add a Control Refnum to the front panel of a subVI, you next need to specify the VI Server Class of the control. This specifies the type of control references that the subVI will accept. In the previous example, Control was selected as the VI Server Class type, as shown in Figure 3-7. This allows the VI to accept a reference to any type of front panel control.

However, you can specify a more specific class for the refnum to make the subVI more restrictive. For example, you can select Digital as the class, and the subVI only can accept references to numeric controls of the class Digital. Selecting a more generic class for a control refnum allows it to accept a wider range of objects, but limits the available properties to those that apply to all objects which the Property Node can accept.

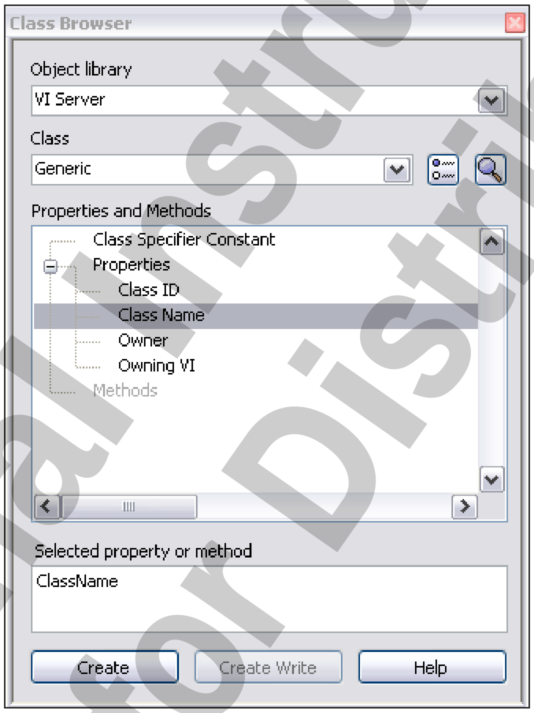
To select a specific control class, right-click the control and select **Select VI Server Class»Generic»GObject»Control** from the shortcut menu. Then, select the specific control class.

Creating Properties and Methods with the Class Browser Window

You can use the Class Browser window to select an object library and create a new property or method.

Complete the following steps to create a new property or method using the Class Browser window.

1. Select **View»Class Browser** to display the Class Browser window.



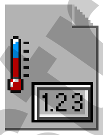
2. From the Object library pull-down menu, select a library.

3. Select a class from the Class pull-down menu. Use the following buttons to navigate the classes.
   - Click the **Select View** button to toggle between an alphabetical view and a hierarchical view of the items in the Class pull-down menu and the Properties and Methods list.
   - Click the **Search** button to launch the Class Browser Search dialog box.
4. From the Properties and Methods list in the Class Browser window, select a property or method. The property or method you select appears in the Selected property or method box.
5. Click the **Create** button or the **Create Write** button to attach a node with the selected property or method to your mouse cursor and add the node to the block diagram. The Create button creates a property for reading or a method. This button dims when you select a write-only property. To create a property for writing, click the **Create Write** button. The Create Write button dims when you select a method or read-only property. You also can drag a property or method from the Properties and Methods list directly to the block diagram.
6. Repeat steps 2 through 5 for any other properties and methods you want to create and add to the block diagram.

Self-Review: Quiz

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

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

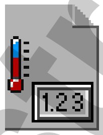


   a. Control reference of a knob
   b. Control reference of a numeric array
   c. Control reference of a thermometer indicator
   d. Control reference of an LED

Self-Review: Quiz Answers

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

2. You have a Numeric control refnum, shown at left, 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

Notes

4 File I/O Techniques

Frequently, the decision to separate the production of data and the consumption of data into separate processes occurs because you must write the data to a file as it is acquired. In such cases, you must choose a file format. This lesson explains ASCII, Binary, and Technical Data Management Streaming (TDMS) file formats and when each is a good choice for your application.

Topics

A. Compare File Formats

B. Create File and Folder Paths

C. Write and Read Binary Files

D. Work with Multichannel Text Files and Headers

E. Access TDMS Files in LabVIEW and Excel

A. Compare File Formats

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. In LabVIEW, three of the most common techniques for storing data are the ASCII file format, direct binary storage, and the TDMS file format. Each of these formats has advantages and some formats work better for storing certain data types than others.

When to Use Text (ASCII) Files

Use text format files for your data to make it available to other users or applications if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important.

Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. A variety of text-based programs can read text-based files.

Store data in text files when you want to access it from another application, such as a word processing or spreadsheet application. To store data in text format, use the String functions to convert all data to text strings. Text files can contain information of different data types.

Text files typically take up more memory than binary files if the data is not originally in text form, such as graph or chart data, because the ASCII representation of data usually is larger than the data itself. For example, you can store the number –123.4567 in 4 bytes as a single-precision, floating-point number. However, its ASCII representation takes 9 bytes, one for each character.

In addition, it is difficult to randomly access numeric data in text files. Although each character in a string takes up exactly 1 byte of space, the space required to express a number as text typically is not fixed. To find the ninth number in a text file, LabVIEW must first read and convert the preceding eight numbers.

You might lose precision if you store numeric data in text files. Computers store numeric data as binary data, and typically you write numeric data to a text file in decimal notation. Loss of precision is not an issue with binary files.

When to Use Binary Files

Storing binary data, such as an integer, uses a fixed number of bytes on disk. For example, storing any number from 0 to 4 billion in binary format, such as 1, 1,000, or 1,000,000, takes up 4 bytes for each number.

Use binary files to save numeric data and to access specific numbers from a file or randomly access numbers from a file. Binary files are machine readable only, unlike text files, which are human readable. Binary files are the most compact and fastest format for storing data. You can use multiple data types in binary files, but it is uncommon.

Binary files are more efficient because they use less disk space and because you do not need to convert data to and from a text representation when you store and retrieve data. A binary file can represent 256 values in 1 byte of disk space. Often, binary files contain a byte-for-byte image of the data as it was stored in memory, except for cases like extended and complex numeric values. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary.

When to Use TDMS Files

To reduce the need to design and maintain your own data file format, National Instruments has created a flexible data model called Technical Data Management Streaming, which is natively accessible through LabVIEW, LabWindows™/CVI™, and DIAdem, and is portable to other applications such as Microsoft Excel. The TDMS data model offers several unique benefits such as the ability to scale your project requirements and easily attach descriptive information to your measurements while streaming your data to disk.

The TDMS file format consists of two files—a `.tdms` file and a `.tdms_index` file. The `.tdms` file is a binary file that contains data and stores properties about that data. The `.tdms_index` file is a binary index file that speeds up access while reading and provides consolidated information on all the attributes and pointers in the TDMS file. All the information in the `.tdms_index` file is also contained in the `.tdms` file. For this reason, the `.tdms_index` file can be automatically regenerated from the `.tdms` file. Therefore, when you distribute TDMS files, you only need to distribute the `.tdms` file. The internal structure of the TDMS file format is publicly documented, so it is possible to create third-party programs to write and read TDMS files. In addition, there is a TDM Excel Add-in Tool available on `ni.com` that you can install to load `.tdms` files into Microsoft Excel.

Use TDMS files to store test or measurement data, especially when the data consists of one or more arrays. TDMS files are most useful when storing arrays of simple data types such as numbers, strings, or Boolean data. TDMS files cannot store arrays of clusters directly. If your data is stored in arrays of clusters, use another file format, such as binary, or break the cluster up into channels and use the structure of the TDMS file to organize them logically.

Use TDMS files to create a structure for your data. Data within a file is organized into channels. You can also organize channels into channel groups. A file can contain multiple channel groups. Well-grouped data simplifies viewing and analysis and can reduce the time required to search for a particular piece of data.

Use TDMS files when you want to store additional information about your data. For example, you might want to record the following information:

- Type of tests or measurements
- Operator or tester name
- Serial numbers
- Unit Under Test (UUT) numbers for the device tested

- Time of the test
- Conditions under which the test or measurement was conducted

Table compares ASCII, TDMS, and Direct Binary file formats.

Table 4-1. File Format Comparison

	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 meta data. High-speed streaming without loss of precision.	Store numeric data compactly with ability to access randomly.

B. Create File and Folder Paths

Regardless of your desired file format, you often need to select a method for creating filenames and destination directories.

To this point, you have used the File Dialog Express VI to specify a file name and location.

The File Dialog Express VI allows user to specify the path to a file or directory. You can also customize the file dialog with prompts and options to limit file types with specific file extensions, such as **`*.txt`** for text files.

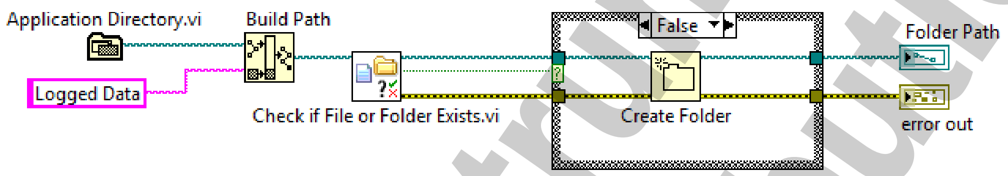
However, it is often desirable to programmatically create folders and filepaths. Hard coded paths are useful for quick prototypes but not recommended for applications as directory structures might differ from one machine to another.

Typically you want to specify a directory location that is relative to a known location, such the project directory or the user's documents directory. Use the Application Directory VI in conjunction with the Build Path function to specify a path relative to the project directory. If you call this VI from the development environment and the VI is loaded in a LabVIEW project file (`.lvproj`), this VI returns the path to the folder containing the project file. If you call this VI from a stand-alone application, this VI returns the path to the folder containing the stand-alone application.

Use the Get System Directory VI in conjunction with the Build Path function to specify a path relative to a system directory such as the User Documents folder or the User Desktop.

Use the Create Folder function to create a new folder. However, if the folder already exists at the specified location, the Create Folder function returns an error instead of overwriting the folder. To avoid the error condition, check if the folder exists to conditionally create the folder. Figure 4-1 shows how to conditionally create a folder named "Logged Data" that is relative to the project file path. The Check if File or Folder Exists.vi checks whether the "Logged Data" folder exists on disk at a specified path and returns True if the folder exists or False if the folder does not exist. This Boolean output is then wired to the select input of a Case structure. Only if the folder doesn't already exist will the Create Folder function be called.

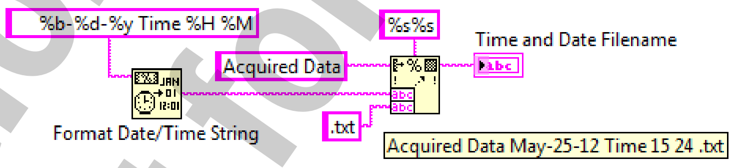
Figure 4-1. Create Folder Path



In addition to specifying a consistent folder directory for your files, it is often desirable to create a series of files with unique names. To help distinguish between multiple files, you might want to include date and time information in the filename.

Figure 4-2 shows how you can use the Format Date/Time String function in conjunction with the Format into String function to format a filename string that includes the current date and time. You can customize the date and time format using a variety of different time format codes. For example, the format codes used in Figure 4-2 include month, day and year information along with hour and minute information. Dates can be specified in long format or abbreviated format. Times can be specified in 24-hour clock format or 12-hour clock format. Refer to the help on the Format Date/Time String function for all the possible time format codes that can be used with this function.

Figure 4-2. Create Time and Data File Name



C. Write and Read Binary Files

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 following list describes the common functions that interact with binary files.

Open/Create/Replace File—Opens a reference to a new or existing file for binary files as it does for ASCII Files.

Write to Binary File—Writes binary data to a file. The function works much like the Write to Text File function, but can accept most data types.

Read from Binary File—Reads binary data starting at its current file position. You must specify to the function the data type to read. Use this function to access a single data element or wire a value to the count input. This causes the function to return an array of the specified data type.

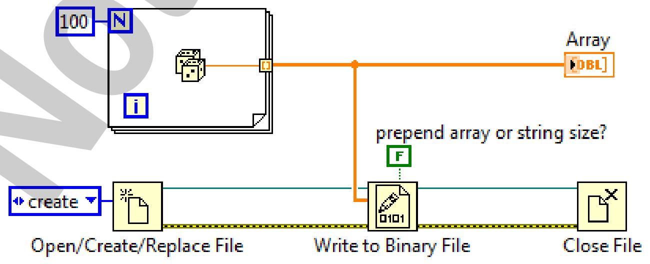
Get File Size—Returns the size of the file in bytes. Use this function in combination with the Read from Binary File function when you want to read all of a binary file. Remember that if you are reading data elements that are larger than a byte you must adjust the count to read.

Get/Set File Position—These functions get and set the location in the file where reads and writes occur. Use these functions for random file access.

Close File—Closes an open reference to a file.

Figure 4-3 shows an example that writes an array of doubles to a binary file. Refer to the *Arrays* section of this lesson for more information about the **Prepend array or string size?** option.

Figure 4-3. Writing a Binary File



Binary Representation

Each LabVIEW data type is represented in a specified way when written to a binary file. This section discusses the representation of each type and important issues when dealing with the binary representation of each type.

Tip A bit is a single binary value. Represented by a 1 or a 0, each bit is either on or off. A byte is a series of 8 bits.

Boolean Values

LabVIEW represents Boolean values as 8-bit values in a binary file. A value of all zeroes represents False. Any other value represents True. This divides files into byte-sized chunks and simplifies reading and processing files. To efficiently store Boolean values, convert a series of Boolean values into an integer using the Boolean Array To Number function. Figure 4-4 shows two methods for writing six Boolean values to a binary file.

Figure 4-4. Writing Boolean Values to a Binary File

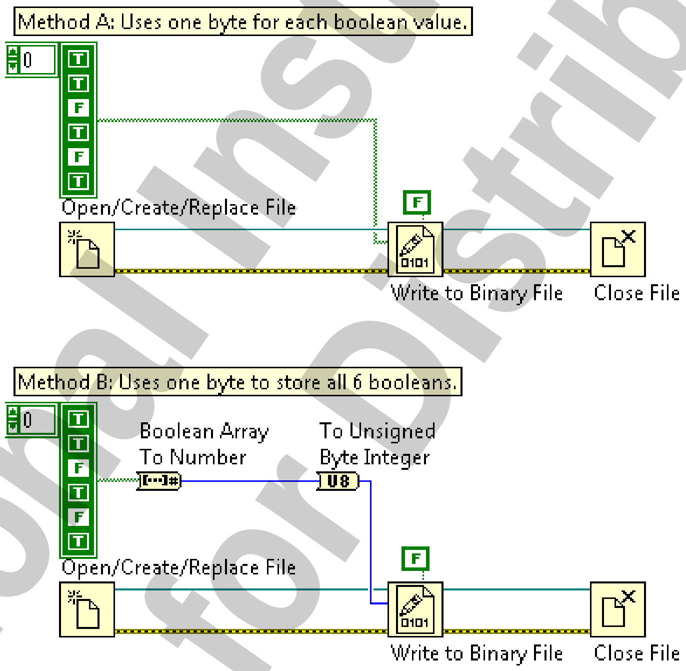


Table 4-2 displays a binary representation of the file contents resulting from running the programs in Figure 4-4. Notice that Method B is a more efficient storage method.

Table 4-2. Results of Figure 4-4

Method A	0000001 00000001 00000000 00000001 00000000 00000001
Method B	00101011

8-bit Integers

Unsigned 8-bit integers (U8) directly correspond to bytes written to the file. When you must write values of various types to a binary file, convert each type into an array of U8s using the Boolean Array To Number, String to Byte Array, Split Number, and Type Cast functions. Then, you can concatenate the various arrays of U8s and write the resulting array to a file. This process is unnecessary when you write a binary file that contains only one type of data.

Table 4-3. U8 Representation

Binary Value	U8 Value
00000000	0
00000001	1
00000010	2
11111111	255

Other Integers

Multi-byte integers are broken into separate bytes and are stored in files in either little-endian or big-endian byte order. Using the Write to Binary File function, you can choose whether you store your data in little-endian or big-endian format.

Little-endian byte order stores the least significant byte first, and the most significant byte last. Big-endian order stores the most significant byte first, and the least significant byte last.

From a hardware point of view, Intel x86 processors use the little-endian byte order while Motorola, PowerPC and most RISC processors use the big-endian byte order. From a software point of view, LabVIEW uses the big-Endian byte order when handling and storing data to disk, regardless of the platform. However, the operating system usually reflects the byte order format of the platform it's running on. For example, Windows running on an Intel platform usually stores data to file using the little-endian byte order. Be aware of this when storing binary data to disk. The binary file functions of LabVIEW have a byte order input that sets the endian form of the data.

Table 4-4. Integer Representations

U32 Value	Little-endian Value	Big-endian Value
0	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000
1	00000001 00000000 00000000 00000000	00000000 00000000 00000000 00000001
255	11111111 00000000 00000000 00000000	00000000 00000000 00000000 11111111

Table 4-4. Integer Representations (Continued)

U32 Value	Little-endian Value	Big-endian Value
65535	11111111 11111111 00000000 00000000	00000000 00000000 11111111 11111111
4,294,967,295	11111111 11111111 11111111 11111111	11111111 11111111 11111111 11111111

Floating-Point Numbers

Floating point numbers are stored as described by the IEEE 754 Standard for Binary Floating-Point Arithmetic. Single-precision numerics use 32-bits each and double-precision numerics use 64-bits each. The length of extended-precision numerics depends on the operating system.

Strings

Strings are stored as a series of unsigned 8-bit integers, each of which is a value in the ASCII Character Code Equivalents Table. This means that there is no difference between writing strings with the Binary File functions and writing them with the Text File functions.

Arrays

Arrays are represented as a sequential list of each of their elements. The actual representation of each element depends on the element type. When you store an array to a file you have the option of preceding the array with a header. A header contains a 4-byte integer representing the size of each dimension. Therefore, a 2D array with a header contains two integers, followed by the data for the array. Figure 4-5 shows an example of writing a 2D array of 8-bit integers to a file with a header. The **prepend array or string size?** input of the Write to Binary File function enables the header. Notice that the default value of this terminal is True. Therefore, headers are added to all binary files by default.

Figure 4-5. Writing a 2D Array of Unsigned Integers to a File with a Header

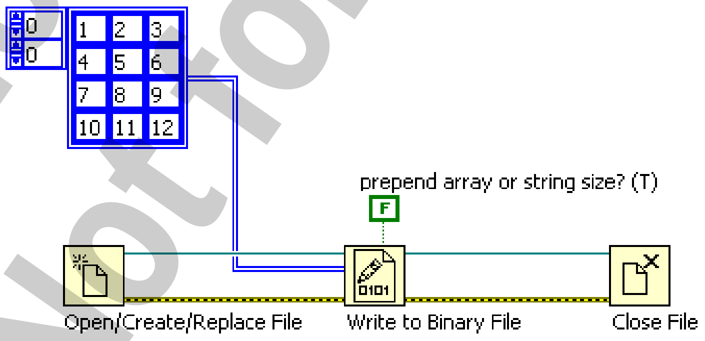


Table 4-5 shows the layout of the file that the code in Figure 4-5 generates. Notice that the headers are represented as 32-bit integers even though the data is 8-bit integers.

Table 4-5. Example Array Representation In Binary File

4	3	1	2	3	4	5	6	7	8	9	10	11	12

Sequential vs. Random Access

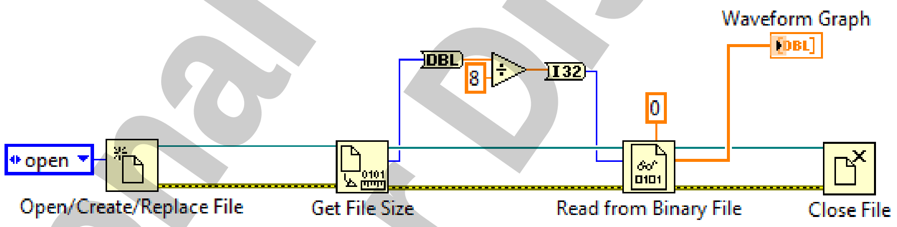
When reading a binary file, there are two methods of accessing data. The first is to read each item in order, starting at the beginning of a file. This is called sequential access and works similar to reading an ASCII file. The second is to access data at an arbitrary point within the file for random access. For example, if you know that a binary file contains a 1D array of 32-bit integers that was written with a header and you want to access the tenth item in the array, you could calculate the offset in bytes of that element in the file and then read only that element. In this example, the element has an offset of 4 (the header) + 10 (the array index) × 4 (the number of bytes in an I32) = 44.

Sequential Access

To sequentially access all the data in a file, you can call the Get File Size function and use the result to calculate the number of items in the file, based on the size of each item and the layout of the file. You can then wire the number of items to the count terminal of the Read Binary function.

Figure 4-6 shows an example of this method.

Figure 4-6. Sequentially Reading an Entire File

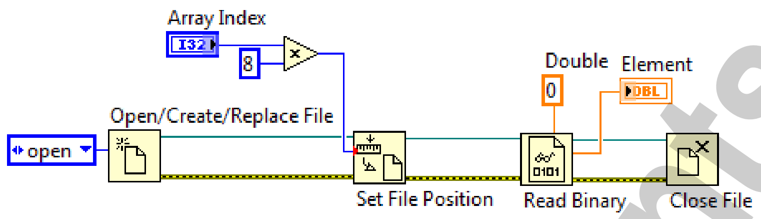


Alternately, you can sequentially access the file one item at a time by repeatedly calling the Read Binary function with the default count of 1. Each read operation updates the position within the file so that you read a new item each time read is called. When using this technique to access data you can check for the **End of File** error after calling the Read Binary function or calculate the number of reads necessary to reach the end of the file by using the Get File Size function.

Random Access

To randomly access a binary file, use the Set File Position function to set the read offset to the point in the file you want to begin reading. Notice that the offset is in bytes. Therefore, you must calculate the offset based on the layout of the file. In Figure 4-7, the VI returns the array item with the index specified, assuming that the file was written as a binary array of double-precision numerics with no header, like the one written by the example in Figure 4-3.

Figure 4-7. Randomly Accessing a Binary File



D. Work with Multichannel Text Files and Headers

Add Headers to the file

When saving measurement data to file, you often want to include information about the measurement, such as the time of test, test fixture, test operator, environment conditions, and measurement units. As you saw earlier, it is possible to programmatically encode some of this information in the filename, such as the date and time. However, the more information you have, the more cumbersome it is to encode the information in the filename. Furthermore, when you open the file, you do not have ready access to the information. Therefore, it is useful to include file and channel headers with the measurement data.

Table 4-6. No Header Data

24.45	⇥	34.54	↵
23.41	⇥	35.32	↵
22.97	⇥	35.98	↵
21.56	⇥	36.76	↵

Table 4-7. 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	↵

Table 4-7. Header Data (Continued)

⇥	⇥	23.41	⇥	35.32	↵
⇥	⇥	22.97	⇥	35.98	↵
⇥	⇥	21.56	⇥	36.76	↵

You create headers to aide understanding of the data.

This section will show students how to create text files that include headers and multiple channels of data using a combination of tab (or comma) and carriage return/line feed characters.

As you see in the exercises and demonstrations, creating headers in text files can become cumbersome, especially if you want to create generic modules for creating the data.

TDMS files may be a better solution for creating files with complex or dynamic headers, and students need to understand the benefits of using them.

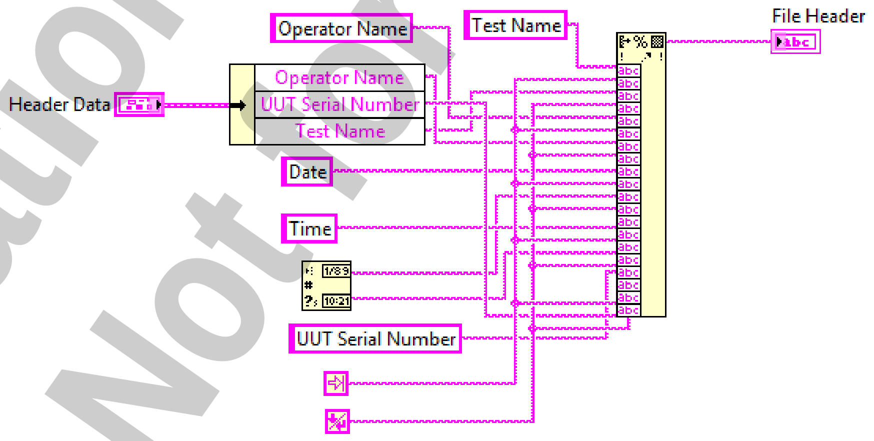
For example, if data is being collected over a long period of time and you want to create entries for the maximum values measured, do you save all the data in memory, calculate the maximum value and write it to file?

Another option would be to write the data to a file, calculate the maximum value when the test is finished and create another file that includes the values.

A TDMS file allows the developer to write data and modify the properties (headers) at any time. National Instruments has done all the hard work of keeping track of the header location in the file.

New LabVIEW developers typically write code similar to that shown in Figure 4-8.

Figure 4-8. Generate File Header



You might use a Format Into String function to create a tab delimited line of text. This method can make it difficult to debug and modify the code for complex strings. In addition, adding new strings to the header requires a lot of work. In other words, the code is not very scalable.

A modular approach, as shown in Figure 4-9 is more scalable and easier to manage.

The use of a subVI, shown in Figure 4-10 to add data to the string and include the column separator (tab or comma) makes the diagram a lot easier to read.

If the column separator does need to be changed, only one VI has to be modified.

Figure 4-9. Write Multiple Channels with Simple Headers

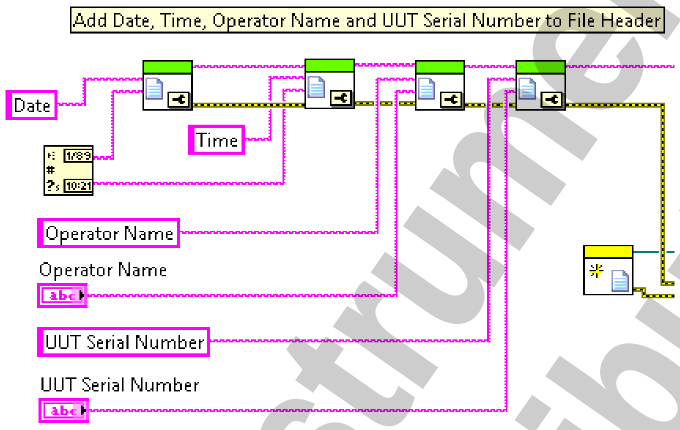
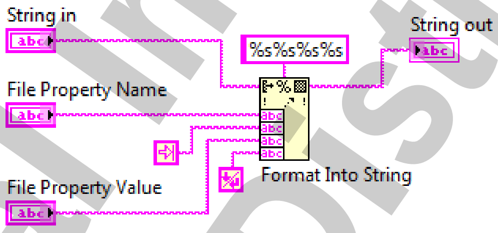


Figure 4-10. Format File Properties SubVI



Write multichannel data

LabVIEW stores multidimensional arrays in row-major order.

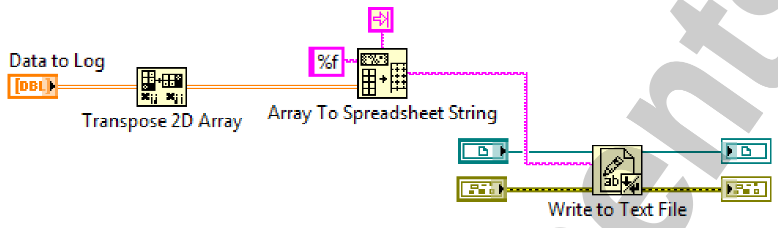
Rows are identified by the first index of a 2D array and columns by the second index.

For example, a 2x3 element 2D array in LabVIEW looks like this:

1	2	3
4	5	6

Representing a 2x50 element array would be wider than the width of this page. Typically, when viewing measurement data in a file, you view the data in column-major format. Transpose data, as shown in Figure 4-11, before writing to file to view channel data in column format. After transposing the array, convert the numeric data to string data using the Array to Spreadsheet String function. This function converts an array to a table in string form. For two-dimensional arrays, the resulting table separates column elements with a user-specified delimiter and separates rows with an EOL character. The default column separator is a tab character.

Figure 4-11.

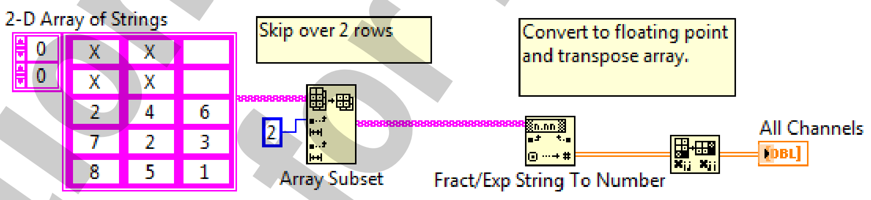


Read data and extract information

Most of the time users want to log data to file so they can share the data with others or view in other applications. It is often useful to read the data back into LabVIEW for future analysis. For example, after logging several test data to file, you might want to read the tests back into LabVIEW for comparison analysis. If the file was created using the Array to Spreadsheet String function or the Write to Spreadsheet File function, you can easily perform the reverse operation using the Spreadsheet String to Array function or Read from Spreadsheet File function. After the data is in a string table, you might want to programmatically extract specific information or select channels of data. This requires that you have consistently formatted files and knowledge of the file format.

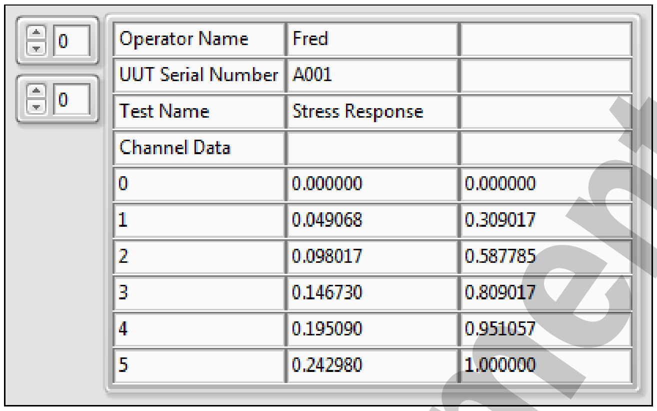
For example, suppose you have files that contain multichannel measurement data with two rows of header information. After reading the file into LabVIEW using the Read from Spreadsheet File function, you want to ignore the header information and read only the measurement data. The simplest approach is to use the Array Subset function to skip over the first two rows of data. Figure 4-12 demonstrates how to extract measurement data from a string table that includes header information (represented by "X" elements).

Figure 4-12.



The problem with this approach is that it isn't very scalable. Over time you decide to store more than two rows of header information. Consider the measurement data shown in Figure 4-13. Consider that you have a variable number of file header properties. How could you programmatically extract a specific property value given the property name? For example, how could you programmatically read the UUT Serial Number value? Or how can you programmatically extract a single channel of data?

Figure 4-13. String Table

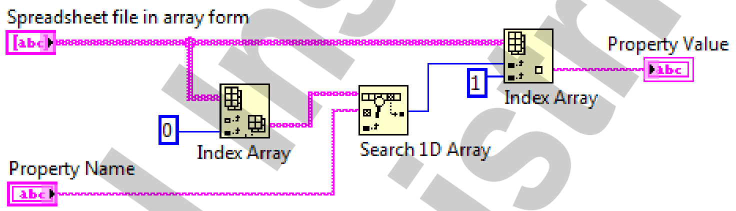


Operator Name	Fred	
UUT Serial Number	A001	
Test Name	Stress Response	
Channel Data		
0	0.000000	0.000000
1	0.049068	0.309017
2	0.098017	0.587785
3	0.146730	0.809017
4	0.195090	0.951057
5	0.242980	1.000000

Given a Property Name, Find the Value

To programmatically extract a file property value, you want to search for a property name match in the first column. Then read the associated value in the second column. The code in Figure 4-14 shows how you can do this in LabVIEW.

Figure 4-14. Programmatically Extract a File Property Value



First you use the Index Array function to put column 0 in a 1D array. You can extract a 1D array from a 2D array by leaving one of the index terminals unwired. In this case, you can extract the first column by specifying 0 in the column index and leaving the row index unwired.

Use the Search 1D Array function to find a Property Name match in the array. The Search 1D Array function searches for an element in a 1D array starting at start index. Because the search is linear, you do not need to sort the array before calling this function. LabVIEW stops searching as soon as the element is found. If the element is found, the function returns the index where the element is found. The index represents the row index in the 2D string array. You can then 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.

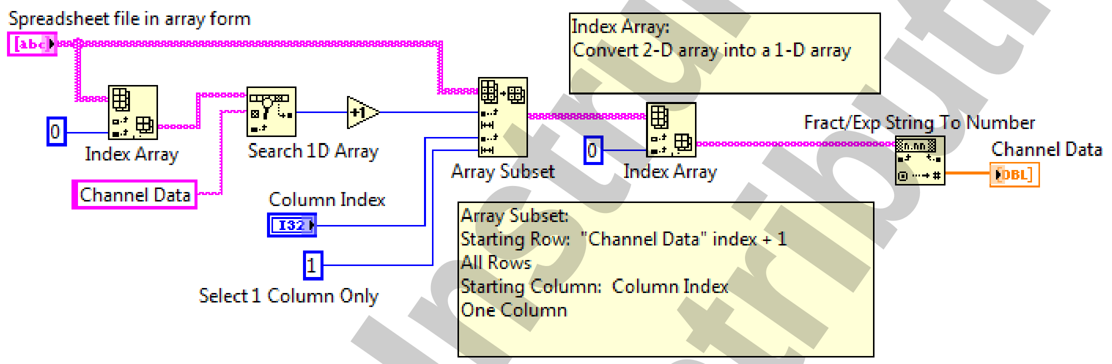
If the Search 1D Array function fails to find a match, the function returns a value of –1. To avoid unexpected errors in your application, you should check for the value of –1 before proceeding with your code. In this case, you might want to force an error if the Search 1D Array function returns a –1.

Extracting a Data Channel

If you have a variable number of file properties preceding measurement data in a file, as shown in Figure 4-15, you cannot rely on hardcoded row index to represent the start of the measurement data. One approach to make the code more scalable is to mark the start of the data. For example, in Figure 4-13, the row directly above the start of the measurement data includes the entry Channel Data. If you can detect the Channel Data entry you know that the measurement data starts on the next row.

You can use the same technique for matching a Property Name to detect the "Channel Data" entry in column 0. You can then use the Array Subset function to extract all the measurement data starting in the next row. Figure 4-15 shows how to extract one column of the measurement data.

Figure 4-15. Extract a Single Data Channel


Spreadsheet file in array form
0
Index Array
Channel Data
Search 1D Array
Column Index
1
Select 1 Column Only
Array Subset
Index Array:
Convert 2-D array into a 1-D array
0
Index Array
Fract/Exp String To Number
Channel Data
Array Subset:
Starting Row: "Channel Data" index + 1
All Rows
Starting Column: Column Index
One Column

E. Access TDMS Files in LabVIEW and Excel

As you learned in the previous section, the more information you include with your measurement data the more complex the file. In addition to file properties, it is often desirable to store channel-specific properties such as channel name, measurement units, test limits, and sensor information. The more information the more complex the file formatting for writing the information and data to file as well as file parsing when reading back the data and information into LabVIEW.

To reduce the need to design and maintain your own data file format, use the TDMS file format. This format was designed specifically for saving measurement data to disk.

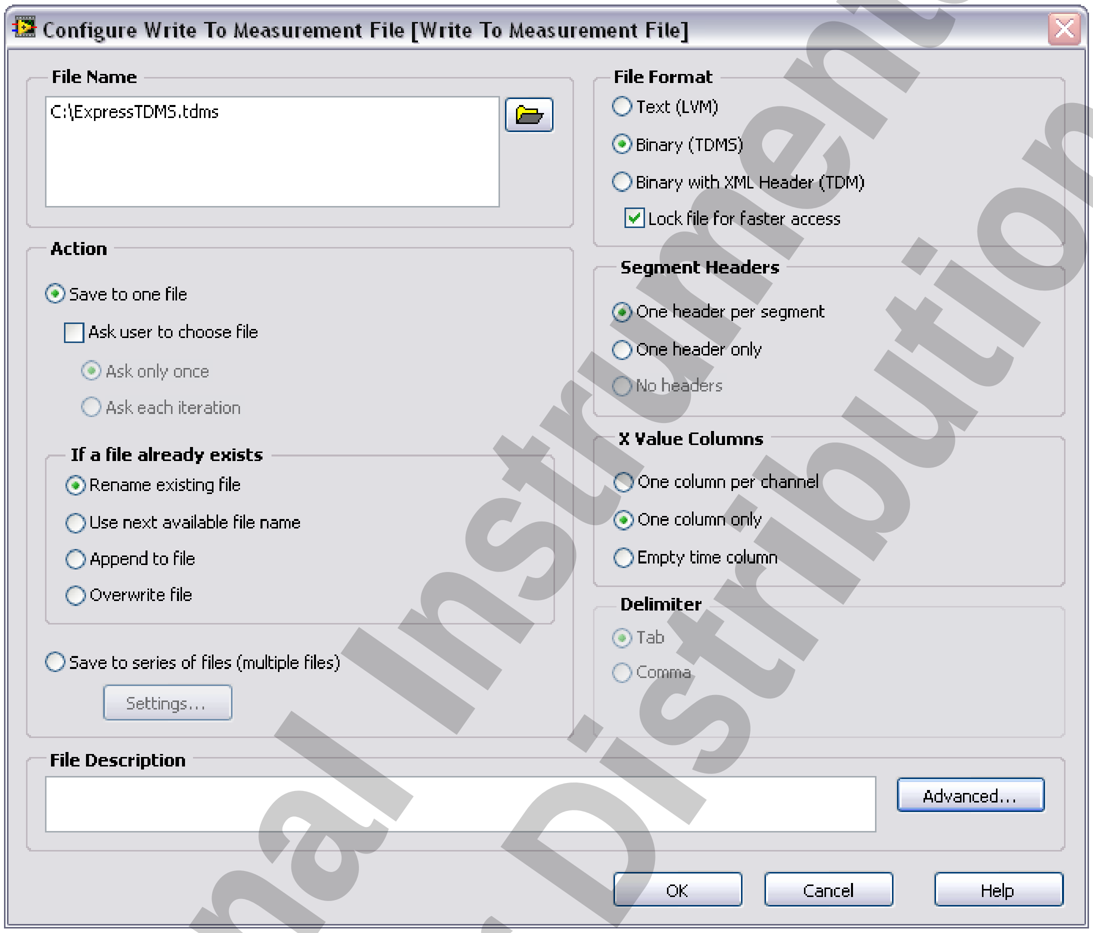
Creating TDMS Files

In LabVIEW, you can create TDMS Files in two ways. Use the Write to Measurement File Express VI and Read from Measurement File Express VI or the TDM Streaming API.

With the Express VIs you can quickly save and retrieve data from the TDMS format. Figure 4-16 shows the configuration dialog box for the Write to Measurement File Express VI. Notice that you can choose to create a LabVIEW measurement data file (LVM) or TDMS file type. However, these

Express VIs give you little control over your data grouping and properties and do not allow you to use some of the features that make TDMS files useful, such as defining channel names and channel group names.

Figure 4-16. Creating a TDMS with Write to Measurement File Express VI



Configure Write To Measurement File [Write To Measurement File]

File Name
C:\ExpressTDMS.tdms

Action
[✓ Save to one file]
Ask user to choose file ☐
[✓ Ask only once]
Ask each iteration

If a file already exists
[✓ Rename existing file]
Use next available file name
Append to file
Overwrite file

Save to series of files (multiple files)
Settings...

File Format
Text (LVM)
[✓ Binary (TDMS)]
Binary with XML Header (TDM)
☑ Lock file for faster access

Segment Headers
[✓ One header per segment]
One header only
No headers

X Value Columns
One column per channel
[✓ One column only]
Empty time column

Delimiter
[✓ Tab]
Comma

File Description

Advanced...

OK Cancel Help

To gain access to the full capabilities of TDMS files, use the TDM Streaming functions. Use the TDM Streaming functions to attach descriptive information to your data and quickly save and retrieve data. Some of the commonly used TDM Streaming functions are described in the *TDMS API* section of this lesson.

Data Hierarchy

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

A channel stores measurement signals or raw data in a TDMS file. The signal is an array of measurement data. Each channel also can have properties that describe the data. The data stored in the signal is stored as binary data on disk to conserve disk space and efficiency.

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

TDMS files each contain as many channel group and channel objects as you want. Each of the objects in a file has properties associated with it, which creates three levels of properties you can use to store data. For example, test conditions are stored at the file level. UUT information is stored at the channel or channel group level. Storing plenty of information about your tests or measurements can make analysis easier.

TDMS API

The following describes some of the most commonly used TDM Streaming VIs and functions.

- **TDMS Open**—Opens a reference to a TDMS file for reading or writing.
- **TDMS Write**—Streams data to the specified TDMS file. It also allows you to create channels and channel groups within your file.
- **TDMS Read**—Reads the specified TDMS file and returns data from the specified channel and/or channel group.
- **TDMS Set Properties**—Sets the properties of the specified TDMS file, channel group, or channel.
- **TDMS Get Properties**—Returns the properties of the specified TDMS file, channel group, or channel.
- **TDMS Close**—Closes a reference to a TDMS File. Notice that you only must close the file reference, any references that you acquire to channels and channel groups close automatically when you close the file reference.

- **TDMS File Viewer**—Opens the specified TDMS file and presents the file data in the TDMS File Viewer dialog box.

- **TDMS List Contents**—Provides a list of group and channel names contained within the specified TDMS file.

- **TDMS Defragment**—Defragments the file data in the specified TDMS data. Use this function to clean up your TDMS data when it becomes cluttered and to increase performance.

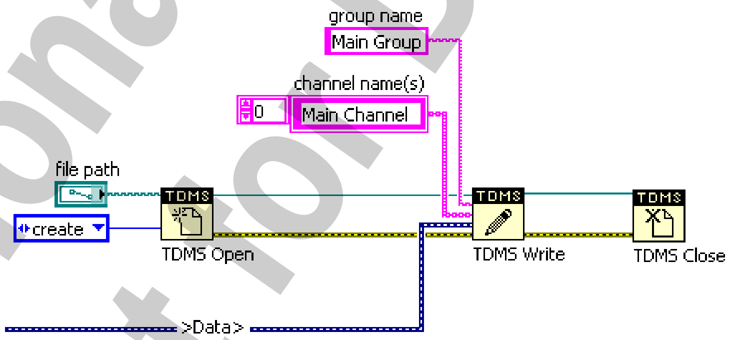
- **TDMS Flush**—Flushes the system memory of all TDMS data to maintain data security.

TDMS Programming

Writing a TDMS File

Figure 4-17 shows the simplest form of writing measurement data with the TDMS API. This example writes data to the channel Main Channel in the channel group Main Group.

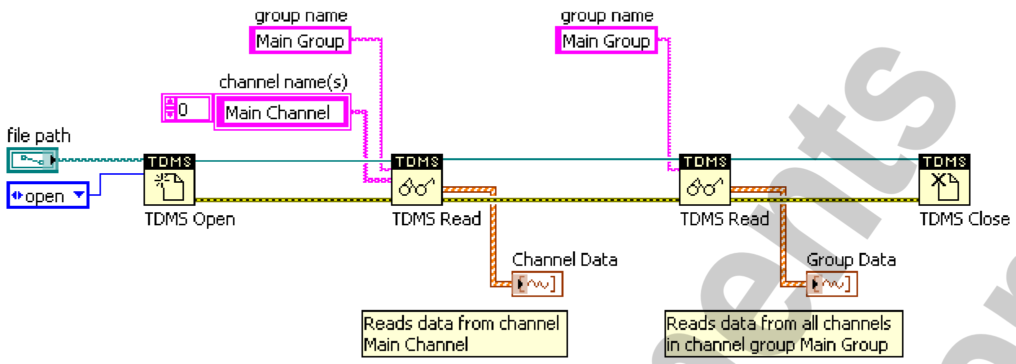
Figure 4-17. Write Data to a TDMS File at the Channel Level



Reading a TDMS File

Figure 4-18 shows the simplest form of reading data using the TDMS API. This example reads all the data in the channel Main Channel from channel group Main Group and displays it in the Channel Data waveform graph. Next, the example reads data from all the channels in the channel group Main Group and displays it in the Group Data waveform graph.

Figure 4-18. Read Data Back from the TDMS File

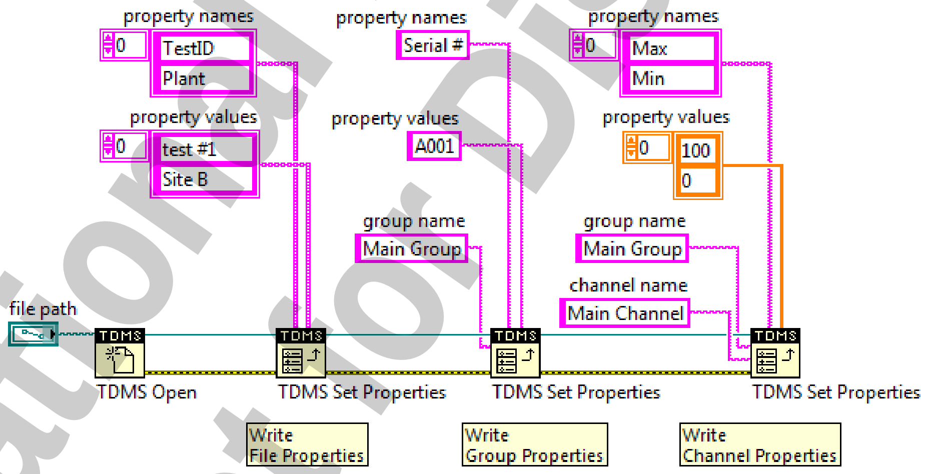


Writing TDMS Custom Properties

The TDMS data model automatically creates certain properties when some data types are written. However, in many cases you may want to create a property specific to your particular needs, such as UUT, serial number, and test temperature. This task can be accomplished using the TDMS Set Properties function with which you can write properties at the file, group, or channel level.

The file level of a property determines which input terminals need to be wired. For example, to write a group property, only the group name input must be wired. To write a channel property, both the group name and channel name inputs must be wired. But, to write a file property, the group name and channel names inputs should be left unwired. Figure 4-19 illustrates these examples.

Figure 4-19. Write Custom Properties at Three Different Levels before Writing Data to the File



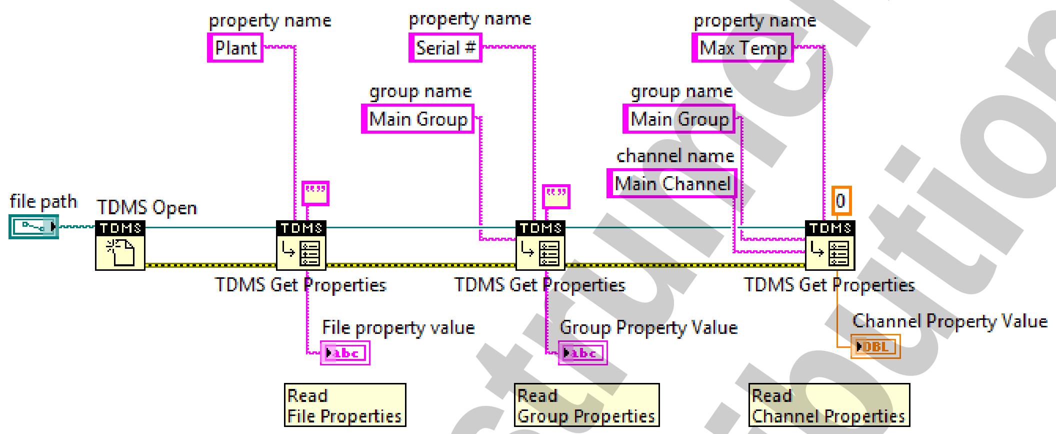
With the TDMS Set Properties function, you can specify an individual property by itself or specify many properties by using arrays. Figure 4-19 shows two properties specified at the file level (TestID and Plant). You could expand this to specify many more properties by increasing the size of the array. Arrays are not necessary if only a single property, such as a serial number, is written.

Property values can also be different data types. In Figure 4-19, string properties are written at the file and group level. But at the channel level, two numeric properties are written to specify the minimum and maximum temperature.

Reading TDMS Custom Properties

When a TDMS file has been written, the properties can be read back into LabVIEW using TDMS Get Properties function. Properties are returned only for the level specified by the wiring of the group name and channel name inputs. This process is similar to writing the properties, as it is shown in Figure 4-20.

Figure 4-20. Read TDMS Properties from Three Different Levels

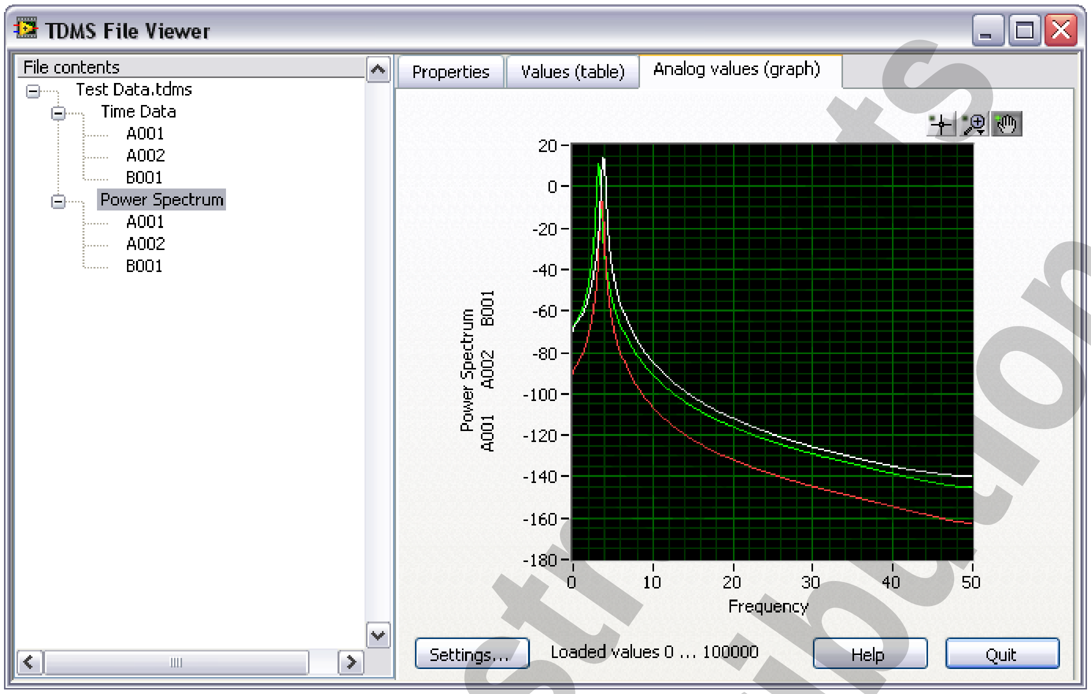


In this configuration, the property values are returned as an array of Variant data because the data could be a string, double, Boolean, or another data type. The data can be displayed on the front panel as a Variant or it can be converted in LabVIEW to the appropriate data type. If a property name and its data type are known, they can be wired as inputs to TDMS Get Properties function and read directly with the correct data type.

TDMS File Viewer

Use the TDMS File Viewer VI when developing a TDMS application to automatically see everything that has been written to a TDMS file by making a single VI call. The TDMS File Viewer VI is flexible and can read complex TDMS files. The TDMS File Viewer VI is included with the TDMS API, so it can be easily placed in a program. Place the TDMS File Viewer VI after the file is closed to use it. The TDMS File Viewer launches another window in which you can view the data and properties inside the TDMS file.

Figure 4-21. TDMS File Viewer

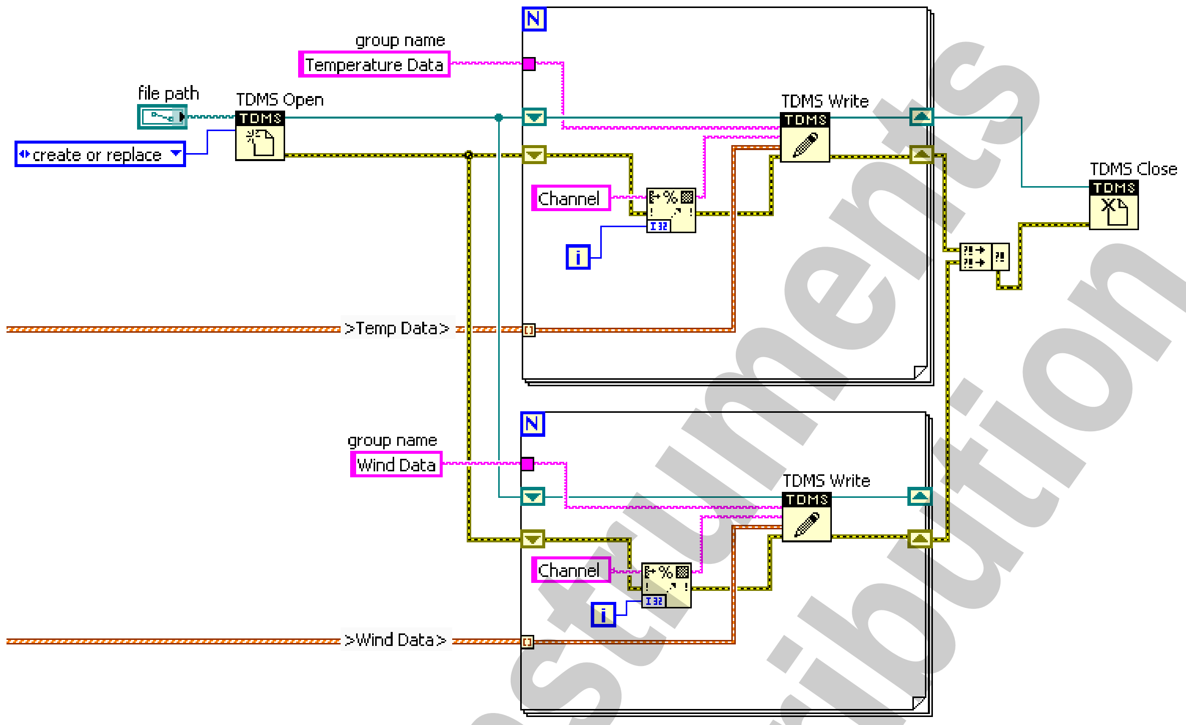


Grouping TDMS Data

Carefully consider the best way to group your data because the data grouping can have a significant impact on the execution speed and implementation complexity of writes and reads. Consider the original format of your data and how you want to process or view the data when choosing a grouping scheme.

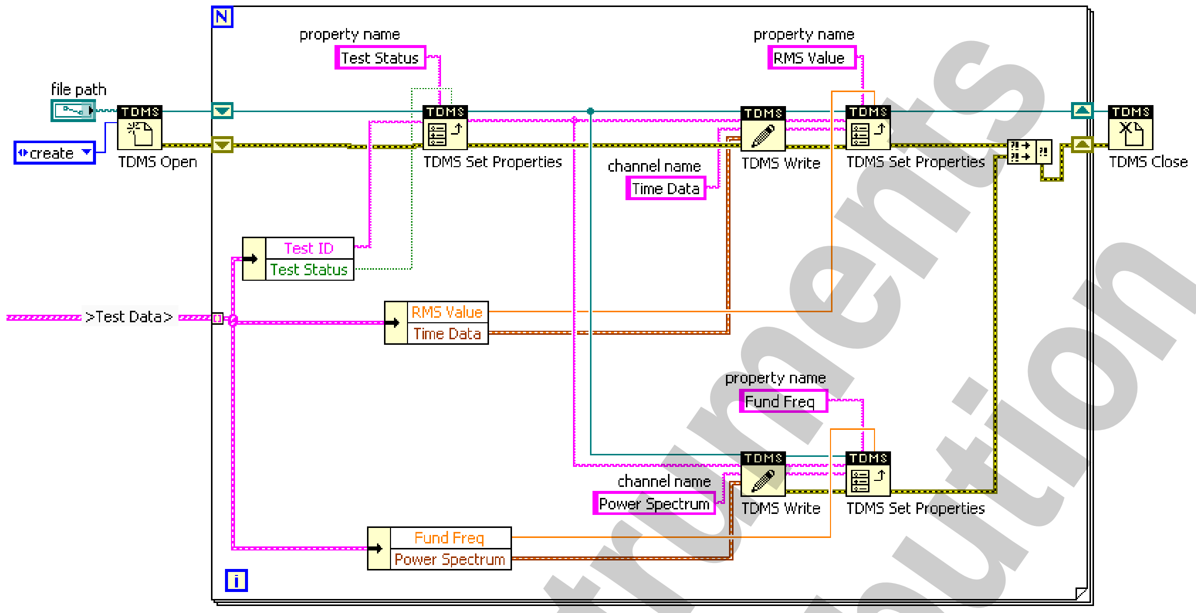
One technique is to group data by the type of data. For example, you might put numeric data in one channel group and string data in another, or you might put time domain data in one group and frequency domain data in another. This makes it easy to compare the channels in a group, but can make it difficult to find two channels that are related to each other. Figure 4-22 shows an example of grouping by the type of data. In this example, the temperature data is placed in one group and the wind data is placed in another. Each group contains multiple channels of data. Notice that when grouping by data type you typically have a fixed number of groups, two in this case, and a dynamically determined number of channels.

Figure 4-22. Grouping Data by Type



Another technique is to group related data. For example, you might put all the data that applies to a single UUT in one group. Grouping related data allows you to easily locate all the related data about a particular subject, but makes it harder to compare individual pieces of data among subjects. Relational grouping helps convert cluster-based storage to a TDMS format. You can store all the information from a given cluster in a channel group, with arrays in the cluster representing channels within the group, and scalar items in the cluster representing properties of the channel group. Figure 4-23 shows an example of relational grouping.

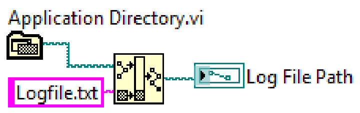
Figure 4-23. Grouping Related Data



Notice that the input data is an array of clusters, each of which contains multiple pieces of information about a test. Each test is stored as a separate channel group. Information that applies to the entire test, such as Test Status, is stored as properties of the channel group. Arrays of data, such as the time data and power spectrum, are stored in channels, and information which relates to the arrays of data, such as the RMS Value and Fundamental Frequency, are stored as properties of the channels. Relational data typically uses a fixed number of channels in a group, but the number of groups is dynamic.

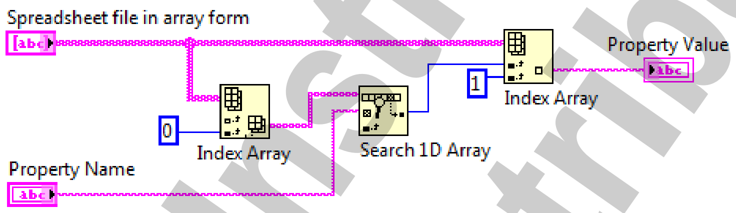
Self-Review: Quiz

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. Same folder as the Application Directory VI

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?



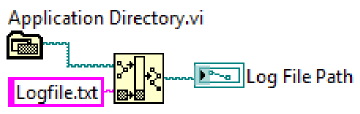
   a. NaN (Not a Number)
   b. 0
   c. –1
   d. Negative Infinity

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

Self-Review: Quiz Answers

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



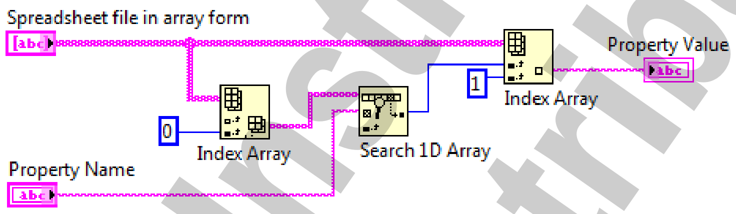
   a. Same folder as the VI that executed the code

   b. Same folder as the LabVIEW Project

   c. Current user's AppData Directory

   d. Same folder as the Application Directory VI

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

Notes

5

Improving an Existing VI

A common problem when you inherit VIs from other developers is that features may have been added without attention to design, thus making it progressively more difficult to add features later in the life of the VI. This is known as software decay. One solution to software decay is to refactor the software. Refactoring is 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.

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

Topics

A. Refactoring Inherited Code

B. Typical Refactoring Issues

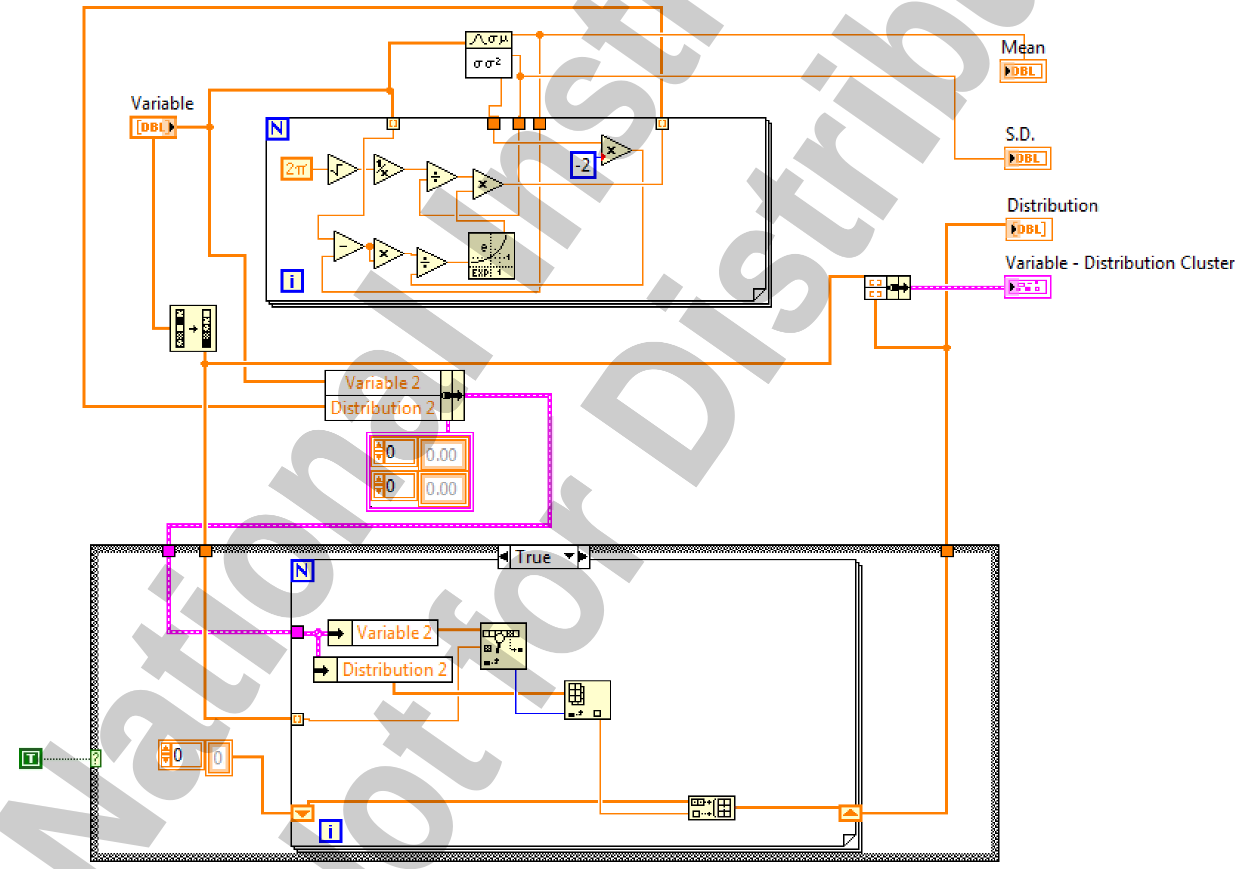
A. Refactoring Inherited Code

Write large and/or long-term software applications with readability in mind because the cost of reading and modifying the software is likely to outweigh the cost of executing the software. It costs more for a developer to read and understand poorly designed code than it does to read code that was created to be readable. In general, more resources are allocated to reading and modifying software than to the initial implementation. Therefore VIs that are easy to read and modify are more valuable than those that are not.

Creating well-designed software facilitates rapid development and decreases possible decay. If a system starts to decay, you can spend large amounts of time tracking down regression failures, which is not productive. Changes also can take longer to implement because it is harder to understand the system if it is poorly designed.

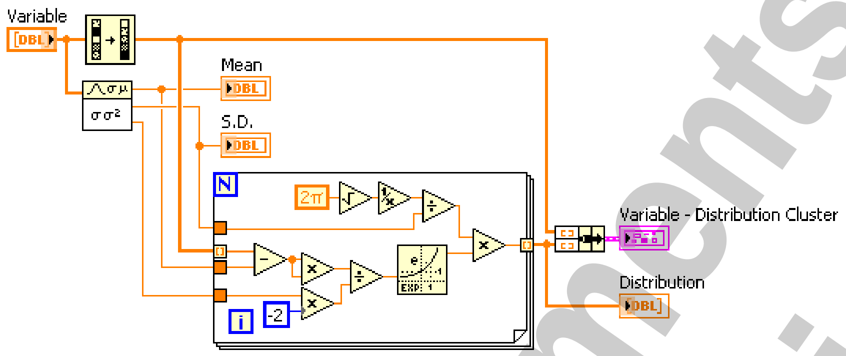
Consider the inherited VI shown in Figure 5-1.

Figure 5-1. Inherited VI



You can refactor the code as shown in Figure 5-2.

Figure 5-2. Refactored Inherited Code



The refactored code performs the same function as the inherited code, but the refactored code is more readable. The inherited code violates many of the block diagram guidelines you have learned.

When you make a VI easier to understand and maintain, you make it more valuable because it is easier to add features to or debug the VI. The refactoring process does not change observable behavior. Changing the way a VI interacts with clients (users or other VIs) introduces risks that are not present when you limit changes to those visible only to developers. The benefit of keeping the two kinds of changes separate is that you can better manage risks.

Refactoring versus Performance Optimization

Although you can make changes that optimize the performance of a VI, this is not the same as refactoring. Refactoring specifically changes the internal structure of a VI to make it easier to read, understand, and maintain. A performance optimization is not refactoring because the goal of optimization is not to make the VI easier to understand and modify. In fact, performance optimization can make VIs more difficult to read and understand, which might be an acceptable trade-off. Sometimes you must sacrifice readability for improved performance, however, readability usually takes priority over speed of performance.

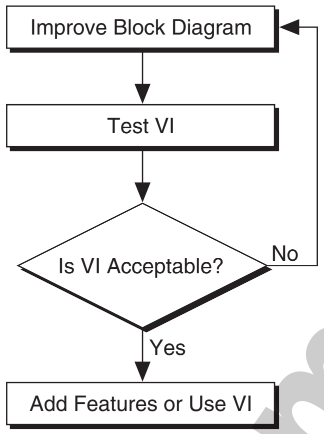
When to Refactor

The right time to refactor is when you are adding a feature to a VI or debugging it. Although you might be tempted to rewrite the VI from scratch, there is value in a VI that works, even if the block diagram is not readable. Good candidates for complete rewrites are VIs that do not work or VIs that satisfy only a small portion of your needs. You also can rewrite simple VIs that you understand well. Consider what works well in an existing VI before you decide to refactor.

Refactoring Process

When you refactor a VI, manage the risk of introducing bugs by making small, incremental changes to the VI and testing the VI after each change. The flowchart shown in Figure 5-3 indicates the process for refactoring a VI.

Figure 5-3. Refactoring Flowchart



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

There are several issues that can complicate working with an inherited VI. The following sections describe typical problems and the refactoring solutions you can use to make inherited VIs more readable.

B. Typical Refactoring Issues

Disorganized or Poorly Designed Block Diagram

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.

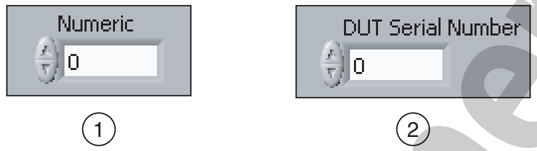
Overly Large Block Diagram

A VI that has a block diagram that is larger than the screen size is difficult to read. You should refactor the VI to make it smaller. The act of scrolling complicates reading a block diagram and understanding the code. Improve a large block diagram by moving objects around. Another technique to reduce the screen space a block diagram occupies is to create subVIs for sections of code within the block diagram. If you cannot reduce the block diagram to fit on the screen, limit the scrolling to one direction.

Poorly Named Objects and Poorly Designed Icons

Inherited VIs often contain controls and indicators that do not have meaningful names. For example, the name of Control 1, shown in Figure 5-4, does not indicate its purpose. Control 2 is the same control, renamed to make the block diagram more readable and understandable.

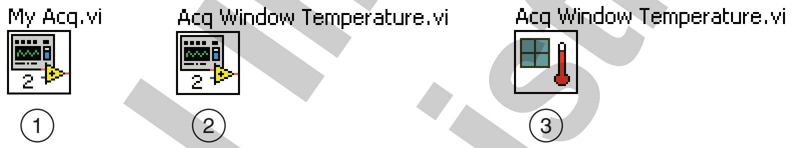
Figure 5-4. Naming Controls



1 Poorly Named Control	2 Meaningfully Named Control

VI names and icons also are important for improving the readability of a VI. For example, the name `My Acq.vi`, shown on the left in Figure 5-5, does not provide any information about the purpose of the VI. You can give the VI a more meaningful name by saving a copy of the VI with a new name and replacing all instances of the VI with the renamed VI. A simpler method is to open all callers of the VI you want to rename, then save the VI with a new name. When you use this method, LabVIEW automatically relinks all open callers of the VI to the new name. `Acq Window Temperature.vi` reflects a more meaningful name for the VI.

Figure 5-5. Poorly Named SubVI



1 Poorly Named VI	3 Meaningful VI Name and VI Icon
2 Meaningfully Named VI	

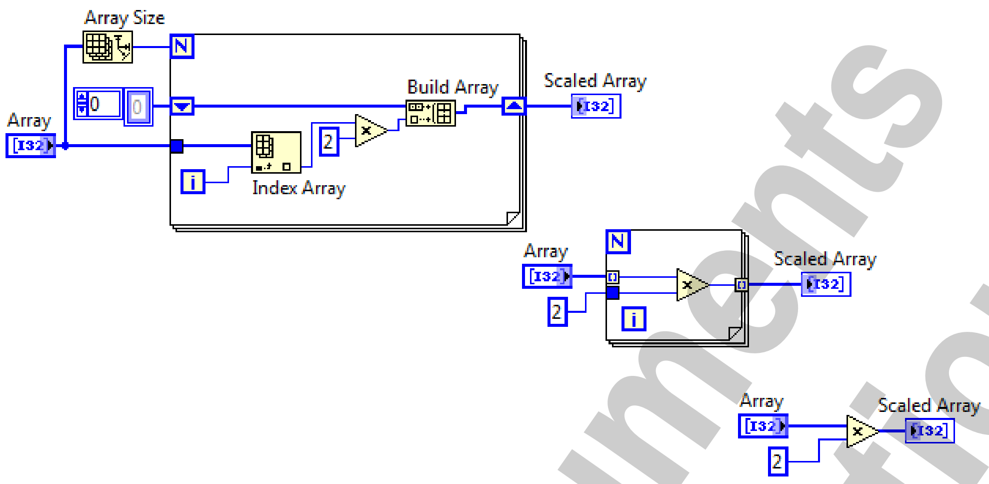
The VI icon also should clarify the purpose of the VI. The default icons used for VI 1 and VI 2 in Figure 5-5 do not represent the purpose of the VI. You can improve the readability of the VI by providing a meaningful icon, as shown for VI 3.

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

Overly Complicated or Unnecessary Logic

The block diagram code at the top of Figure 5-6 contains overly complicated and unnecessary logic. This code can be simplified significantly. Each of the sections of code performs the same functionality, however, the example at the lower right is much easier to read and maintain.

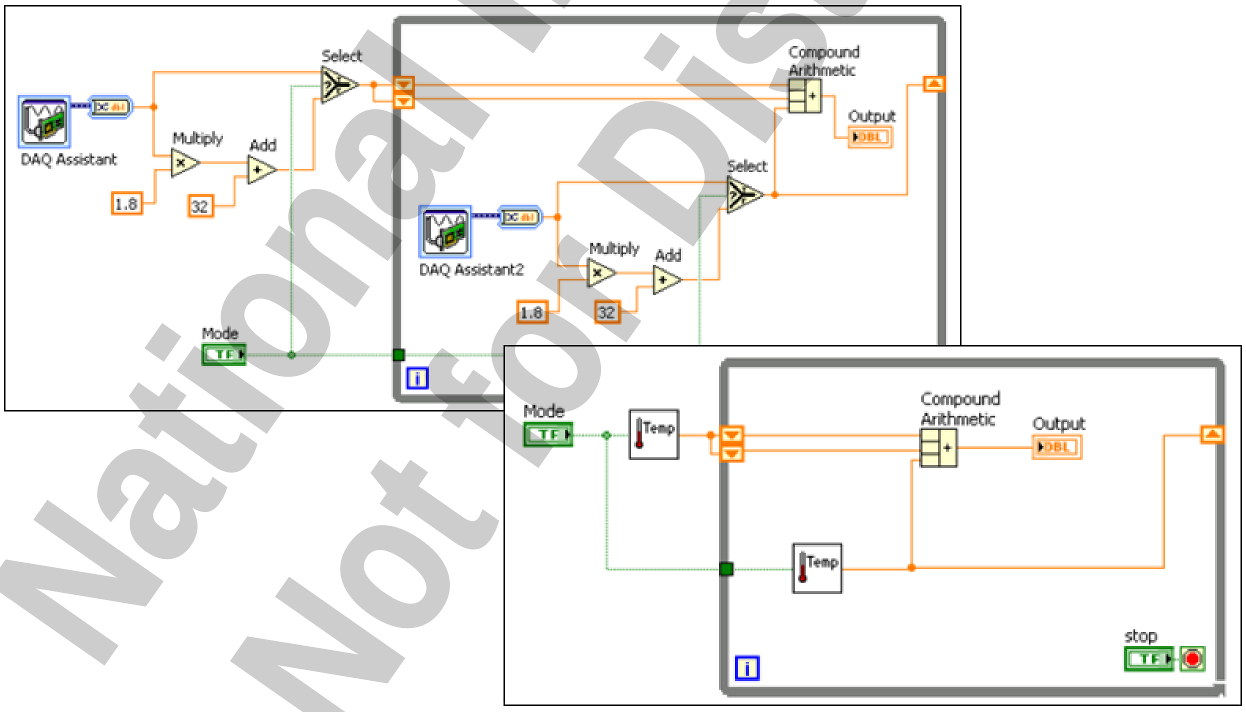
Figure 5-6. Overly Complicated Logic



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 Figure 5-7 shows how you can create a subVI from sections of code that are reused in two places on the block diagram.

Figure 5-7. Duplicate Logic



Dataflow Broken

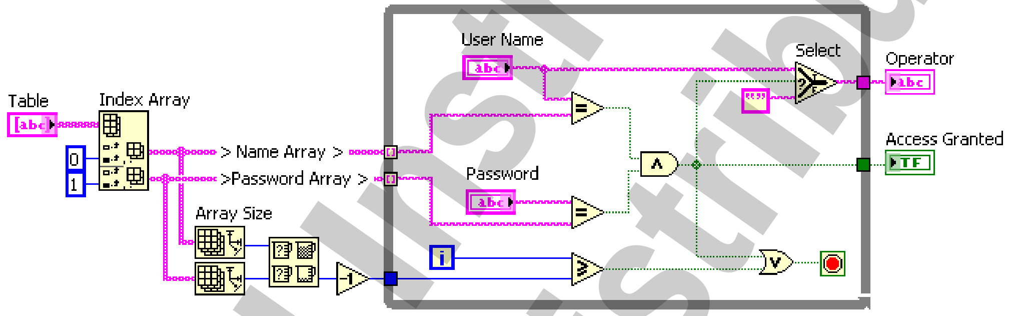
If there are Sequence structures and local variables on the block diagram, the VI probably does not use dataflow to determine the programming flow.

You should replace most Sequence structures with the state machine design pattern. Delete local variables and wire the controls and indicators directly.

Complicated Algorithms

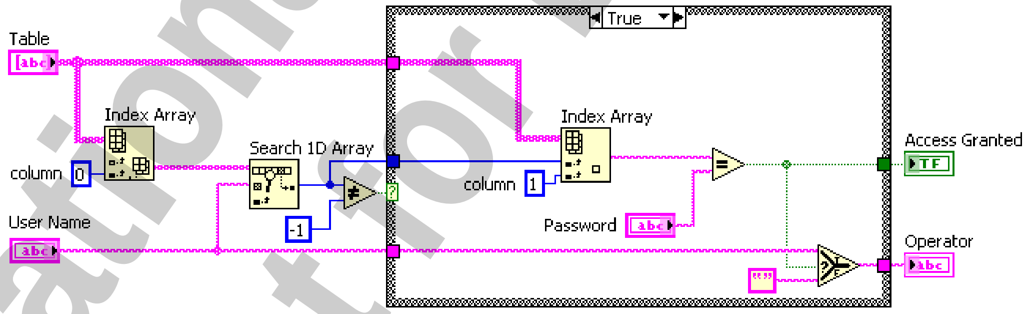
Complicated algorithms can make a VI difficult to read. Complicated algorithms can be more difficult to refactor because there is a higher probability that the changes introduce errors. When you refactor a complicated algorithm, make minor changes and test the code frequently. In some cases you can refactor a complicated algorithm by using built-in LabVIEW functions. For example, the VI in Figure 5-8 checks a user name and password against a database.

Figure 5-8. Complicated Algorithm VI



You could refactor this VI using the built-in functions for searching strings, as shown in Figure 5-9.

Figure 5-9. Refactored VI

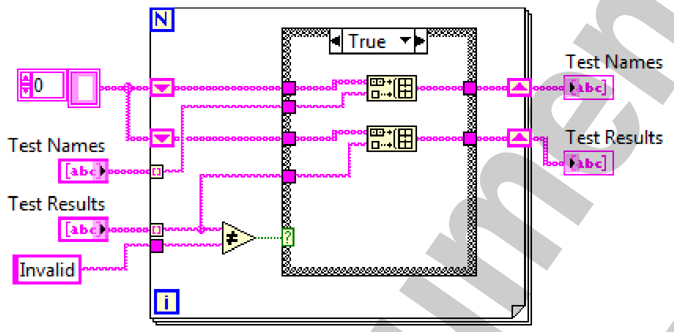


Outdated Practices

Refactor code that was created in earlier versions of LabVIEW and uses outdated practices. For example, you should replace polling-based design with event-based design and use new features that simplify your code.

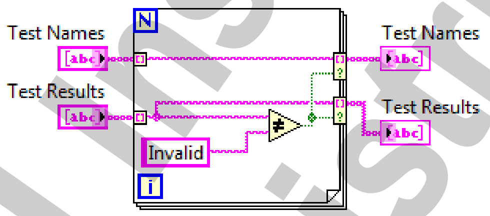
The block diagrams shown in Figure 5-10 and Figure 5-11 both filter out tests with invalid results. The block diagram shown in Figure 5-10 shows code that passes a condition to a Case structure. If the test results are not valid, then the Test Names and Test Results values are concatenated in the array.

Figure 5-10. Code Using Outdated Techniques



The block diagram shown in Figure 5-11 replaces the Case structure with a Conditional terminal on the output tunnel. The Conditional terminal simplifies the block diagram.

Figure 5-11. Refactored Code Using Conditional Output Tunnels



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

Notes

Notes

6
Deploying an Application

This lesson describes the process of creating a stand-alone application and installer for your LabVIEW projects.

Topics

A. Preparing the Files
B. Build Specifications
C. Create and Debug an Application

A. Preparing the Files

A stand-alone application allows the user to run your VIs without installing the LabVIEW development system. Installers distribute the stand-alone application. Installers can include the LabVIEW Run-Time Engine, which is necessary for running stand-alone applications. However, you can also download the LabVIEW Run-Time Engine at `ni.com/downloads`.

Before you can create a stand-alone application with your VIs, you must first prepare your files for distribution. The following topics describe a few of the issues you need to consider as part of your preparation. Refer to the *Preparing Files* section of the *Building Applications Checklist* topic in the *LabVIEW Help* for more information.

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.

Path Names

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.

Quit LabVIEW

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.

External Code

Know what external code your application uses. For example, do you call any system or custom DLLs or shared libraries? Are you going to process command line arguments? These are advanced examples that are beyond the scope of this course, but you must consider them for the application. Refer to the *Calling Shared Libraries* topic in the *LabVIEW Help*.

VI Server Properties and Methods

Not all VI Server properties and methods are supported in the LabVIEW Run-Time Engine. Unsupported VI server properties and methods return errors when called from the Run-Time Engine. Use proper error handling to detect any of these unsupported properties and methods. You can also review the *VI Server Properties and Methods Not Supported in the LabVIEW Run-Time Engine* topic in the *LabVIEW Help*.

Providing Online Help in Your LabVIEW Applications

As you put the finishing touches on your application, you should provide online help to the user. To create effective documentation for VIs, create VI and object descriptions that describe the purpose of the VI or object and give users instructions for using the VI or object.

Use the following functions, located on the **Help** palette, to programmatically show or hide the **Context Help** window and link from VIs to HTML files or compiled help files:

- Use the Get Help Window Status function to return the status and position of the **Context Help** window.
- Use the Control Help Window function to show, hide, or reposition the **Context Help** window.
- Use the Control Online Help function to display the table of contents, jump to a specific topic in the file, or close the online help.
- Use the Open URL in Default Browser VI to display a URL or HTML file in the default Web browser.

B. Build Specifications

After you have prepared your files for distribution, you need to create a build specification for your application. The Build Specifications node in the Project Explorer window allows you to create and configure build specifications for LabVIEW builds. A build specification contains all the settings for the build, such as files to include, directories to create, and settings for VIs.

Note If you previously hid **Build Specifications** in the **Project Explorer** window, you must display the item again to access it in the **Project Explorer** window.

You can create and configure the following types of build specifications:

- Stand-alone applications—Use stand-alone applications to provide other users with executable versions of VIs. Applications are useful when you want users to run VIs without installing the LabVIEW development system. Stand-alone applications require the LabVIEW Run-Time Engine. **(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. Installers that include the LabVIEW Run-Time Engine are useful if you want users to be able to run applications or use shared libraries without installing LabVIEW.
- .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. When you deploy VIs in a packed library, fewer files deploy because the packed library is a single file. The top-level file of a packed library is a project library. Packed libraries

contain one or more VI hierarchies that compile for a specific operating system. 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. Shared libraries are useful when you want to share the functionality of the VIs you build with other developers. Other developers can use the shared libraries but cannot edit or view the block diagrams unless you enable debugging. **(Windows)** Shared libraries have a `.dll` extension. **(Mac OS)** Shared libraries have a `.framework` extension. **(Linux)** Shared libraries have a `.so` extension. You can use `.so` or you can begin with `lib` and end with `.so`, optionally followed by the version number. This allows other applications to use the library.
- Source distributions—Use source distributions to package a collection of source files. Source distributions are useful if you want to send code to other developers to use in LabVIEW. 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.
- Web services (RESTful)—**(Windows)** Publish VIs within LabVIEW Web services to provide a standardized method for the LabVIEW Web Server to deploy applications that any HTTP client can access. Web services support clients across most major platforms and programming languages and allow you to easily implement and deploy Web applications over a network using LabVIEW.
- 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.

Refer to the *Configuring Build Specifications* section of the *Building Applications Checklist* topic in the *LabVIEW Help* for more information.

C. Create and Debug an Application

Why Create an Installer?

Building an installer will let you make sure the application installs with the correct version of the RTE. Applications developed in LabVIEW 2012, for example, require the LabVIEW 2012 Run-Time Engine.

If an application relies on drivers (like NI-DAQmx), you also will need to be install them on the target.

If you have just one destination machine, you can copy the `.exe` onto the destination computer, and then install support drivers and files separately. However, to replicate the application on several machine and ensure that files are copied over correctly on each machine, using an installer ensures that files are installed properly every time.

Professional applications are almost always distributed with an installer. Customers expect to run `setup.exe` on any Windows machine when they are installing software.

System Requirements

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.

Configuring Build Specifications

You must create build specifications in the **Project Explorer** window. Expand **My Computer**, right-click **Build Specifications**, select **New** and the type of build you want to configure from the shortcut menu. Use the pages in the **Source Distribution Properties**, **Application Properties**, **Shared Library Properties**, **Installer Properties**, **.NET Interop Assembly Properties**, **Web Service (RESTful) Properties**, **Packed Library Properties**, or **Zip File Properties** dialog boxes to configure settings for the build specification. After you define these settings, click the **OK** button to close the dialog box and update the build specification in the project. The build specification appears under **Build Specifications**. Right-click a specification and select **Build** from the shortcut menu to complete the build. You also can select **Build All** from the shortcut menu to build all specifications under **Build Specifications**. If you rebuild a given specification, LabVIEW overwrites the existing files from the previous build that are part of the current build.

Refer to the *Caveats and Recommendations for Building Installers* topic in the *LabVIEW Help* for more information.

Debugging Executables

You can debug stand-alone applications and shared libraries that you create with the Application Builder.

Complete the following instructions to debug a stand-alone application or shared library.

Note When you debug applications and shared libraries, you cannot debug reentrant panels that an Open VI Reference function creates or reentrant panels that are entry points to LabVIEW-built shared libraries. You also cannot debug subVIs within Diagram Disable structures. SubVIs within Diagram Disable structures appear as question marks while you debug an application. However, because code in a Diagram Disable structure does not execute, this does not affect debugging.

1. Before you build the application or shared library, you must enable debugging in the build specification. For an application, place a checkmark in the **Enable debugging** checkbox on the

Advanced page of the Application Properties dialog box. For a shared library, place a checkmark in the **Enable debugging** checkbox on the Advanced page of the Shared Library Properties dialog box. Enabling debugging retains the block diagrams of the VIs in the build so you can perform debugging.

2. Place a checkmark in the **Wait for debugger on launch** checkbox if you want the application or shared library to wait to run any VIs until you run the debugging tool.
3. Build the application or shared library.
4. Run the built application or call the shared library from outside of LabVIEW.
5. From the Project Explorer window, select **Operate»Debug Application or Shared Library** to display the Debug Application or Shared Library dialog box. The dialog box displays a list of applications and shared libraries with debugging enabled.
6. If the application or shared library you want to debug is running on a different computer, enter the computer name in the **Machine name or IP address** text box. Click the **Refresh** button to view the list of applications and shared libraries with debugging enabled on the remote computer.
7. Select the application or shared library you want to debug.
8. Click the **Connect** button to perform debugging. LabVIEW downloads the application or shared library and the front panel of the startup VI appears for debugging. If you enabled Wait for debugger on launch, you must click the Run button to start the application or shared library.
9. Use the block diagram of the startup VI to debug the application or shared library. You can use probes, breakpoints, and other debugging techniques to identify problems. You then can correct any problems found during debugging.

Note If you are debugging using a custom probe that uses a Call Library Function Node on a shared library that was not built with the application or shared library you want to debug, you must place the shared library in the same directory as the application or shared library you want to debug.

10. After you finish debugging, close the startup VI, which also closes the remotely controlled application or shared library. If you want to disconnect the remotely controlled application or shared library without closing the startup VI, right-click the startup VI and select **Remote Debugging»Quit Debug Session** from the shortcut menu.

The following scenarios might cause the **`No debuggable applications or runtime libraries found`** error to appear.

- The debuggable application or shared library was not loaded or the debuggable shared library was unloaded by the application.
- The configuration (`ini`) file of the shared library or application, created by the Application Builder, was not distributed with the shared library or application.

Summary

- LabVIEW features the Application Builder, which enables you to create stand-alone executables and installers. The Application Builder is available in the Professional Development Systems or as an add-on package.
- Creating a professional, stand-alone application with your VIs involves understanding the following:
  - The architecture of your VI
  - The programming issues particular to the VI
  - The application building process
  - The installer building process

Notes

A

Additional Information and Resources

This appendix contains additional information about National Instruments technical support options and LabVIEW resources.

National Instruments Technical Support Options

Visit the following sections of the award-winning National Instruments Web site at `ni.com` for technical support and professional services:

- **Support**—Technical support at `ni.com/support` includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, 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.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support, as well as exclusive access to self-paced online training modules at `ni.com/self-paced-training`. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    For information about other technical support options in your area, visit `ni.com/services` or contact your local office at `ni.com/contact`.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. The NI Alliance Partners joins system integrators, consultants, and hardware vendors to provide comprehensive service and expertise to customers. The program ensures qualified, specialized assistance for application and system development. To learn more, call your local NI office or visit `ni.com/alliance`.

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

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.

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.

LabVIEW Resources

This section describes how you can receive more information regarding LabVIEW.

LabVIEW Publications

LabVIEW Books

Many books have been written about LabVIEW programming and applications. The National Instruments web site contains a list of all the LabVIEW books and links to places to purchase these books. Visit `ni.com/reference/books` for more information.

Glossary

A

automatic scaling	Ability of scales to adjust to the range of plotted values. On graph scales, autoscaling determines maximum and minimum scale values.

B

block diagram	Pictorial description or representation of a program or algorithm. The block diagram consists of executable icons called nodes and wires that carry data between the nodes. The block diagram is the source code for the VI. The block diagram resides in the block diagram window of the VI.
Boolean controls and indicators	Front panel objects to manipulate and display Boolean (TRUE or FALSE) data.
broken Run button	Button that replaces the Run button when a VI cannot run because of errors.
broken VI	VI that cannot run because of errors; signified by a broken arrow in the broken Run button.

C

channel	1. Physical—A terminal or pin at which you can measure or generate an analog or digital signal. A single physical channel can include more than one terminal, as in the case of a differential analog input channel or a digital port of eight lines. A counter also can be a physical channel, although the counter name is not the name of the terminal where the counter measures or generates the digital signal. 2. Virtual—A collection of property settings that can include a name, a physical channel, input terminal connections, the type of measurement or generation, and scaling information. You can define NI-DAQmx virtual channels outside a task (global) or inside a task (local). Configuring virtual channels is optional in Traditional NI-DAQ (Legacy) and earlier versions, but is integral to every measurement you take in NI-DAQmx. In Traditional NI-DAQ (Legacy), you configure virtual channels in MAX. In NI-DAQmx, you can configure virtual channels either in MAX or in your program, and you can configure channels as part of a task or separately. 3. Switch—A switch channel represents any connection point on a switch. It can be made up of one or more signal wires (commonly one, two, or four), depending on the switch topology. A virtual channel cannot be created with a switch channel. Switch channels may be used only in the NI-DAQmx Switch functions and VIs.
checkbox	Small square box in a dialog box which you can select or clear. Checkboxes generally are associated with multiple options that you can set. You can select more than one checkbox.
conditional terminal	Terminal of a While Loop that contains a Boolean value that determines if the VI performs another iteration.
Context Help window	Window that displays basic information about LabVIEW objects when you move the cursor over each object. Objects with context help information include VIs, functions, constants, structures, palettes, properties, methods, events, and dialog box components.
control	Front panel object for entering data to a VI interactively or to a subVI programmatically, such as a knob, push button, or dial.
Controls palette	Palette that contains front panel controls, indicators, and decorative objects.
current VI	VI whose front panel, block diagram, or Icon Editor is the active window.

D

DAQ	*See* data acquisition (DAQ).
DAQ Assistant	A graphical interface for configuring measurement tasks, channels, and scales.
DAQ device	A device that acquires or generates data and can contain multiple channels and conversion devices. DAQ devices include plug-in devices, PCMCIA cards, and DAQPad devices, which connect to a computer USB or IEEE 1394 port. SCXI modules are considered DAQ devices.
data acquisition (DAQ)	1. Acquiring and measuring analog or digital electrical signals from sensors, acquisition transducers, and test probes or fixtures. 2. Generating analog or digital electrical signals.
data flow	Programming system that consists of executable nodes that execute only when they receive all required input data. The nodes produce output data automatically when they execute. LabVIEW is a dataflow system. The movement of data through the nodes determines the execution order of the VIs and functions on the block diagram.
data type	Format for information. In LabVIEW, acceptable data types for most VIs and functions are numeric, array, string, Boolean, path, refnum, enumeration, waveform, and cluster.
default	Preset value. Many VI inputs use a default value if you do not specify a value.
device	An instrument or controller you can access as a single entity that controls or monitors real-world I/O points. A device often is connected to a host computer through some type of communication network. *See also* DAQ device and measurement device.
drag	To use the cursor on the screen to select, move, copy, or delete objects.
driver	Software that controls a specific hardware device, such as a DAQ device.
dynamic data type	Data type used by Express VIs that includes the data associated with a signal and attributes that provide information about the signal, such as the name of the signal or the date and time LabVIEW acquired the data. Attributes specify how the signal appears on a graph or chart.

E

Error list window	Window that displays errors and warnings occurring in a VI and in some cases recommends how to correct the errors.
error message	Indication of a software or hardware malfunction or of an unacceptable data entry attempt.
Express VI	A subVI designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.

F

For Loop	Iterative loop structure that executes its subdiagram a set number of times. Equivalent to text-based code: `For i = 0 to n – 1, do....`
front panel	Interactive user interface of a VI. Front panel appearance imitates physical instruments, such as oscilloscopes and multimeters.
function	Built-in execution element, comparable to an operator, function, or statement in a text-based programming language.
Functions palette	Palette that contains VIs, functions, block diagram structures, and constants.

G

General Purpose Interface Bus	GPIB. Synonymous with HP-IB. The standard bus used for controlling electronic instruments with a computer. Also called IEEE 488 bus because it is defined by ANSI/IEEE Standards 488-1978, 488.1-1987, and 488.2-1992.
graph	2D display of one or more plots. A graph receives and plots data as a block.

I

I/O	Input/Output. The transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.
icon	Graphical representation of a node on a block diagram.
indicator	Front panel object that displays output, such as a graph or LED.
instrument driver	A set of high-level functions that control and communicate with instrument hardware in a system.

Instrument I/O Assistant	Add-on launched from the Instrument I/O Assistant Express VI that communicates with message-based instruments and graphically parses the response.

L

label	Text object used to name or describe objects or regions on the front panel or block diagram.
LabVIEW	Laboratory Virtual Instrument Engineering Workbench. LabVIEW is a graphical programming language that uses icons instead of lines of text to create programs.
LED	Light-emitting diode.
legend	Object a graph or chart owns to display the names and plot styles of plots on that graph or chart.

M

MAX	*See* Measurement & Automation Explorer.
Measurement & Automation Explorer	The standard National Instruments hardware configuration and diagnostic environment for Windows.
measurement device	DAQ devices such as the E Series multifunction I/O (MIO) devices, SCXI signal conditioning modules, and switch modules.
menu bar	Horizontal bar that lists the names of the main menus of an application. The menu bar appears below the title bar of a window. Each application has a menu bar that is distinct for that application, although some menus and commands are common to many applications.

N

NI-DAQ	Driver software included with all NI DAQ devices and signal conditioning components. NI-DAQ is an extensive library of VIs and ANSI C functions you can call from an application development environment (ADE), such as LabVIEW, to program an NI measurement device, such as the M Series multifunction I/O (MIO) DAQ devices, signal conditioning modules, and switch modules.

NI-DAQmx	The latest NI-DAQ driver with new VIs, functions, and development tools for controlling measurement devices. The advantages of NI-DAQmx over earlier versions of NI-DAQ include the DAQ Assistant for configuring channels and measurement tasks for your device for use in LabVIEW, LabWindows/CVI, and Measurement Studio; NI-DAQmx simulation for most supported devices for testing and modifying applications without plugging in hardware; and a simpler, more intuitive API for creating DAQ applications using fewer functions and VIs than earlier versions of NI-DAQ.
node	Program execution element. Nodes are analogous to statements, operators, functions, and subroutines in text-based programming languages. On a block diagram, nodes include functions, structures, and subVIs.
numeric controls and indicators	Front panel objects to manipulate and display numeric data.

O

object	Generic term for any item on the front panel or block diagram, including controls, indicators, structures, nodes, wires, and imported pictures.
Operating tool	Tool to enter data into controls or to operate them.

P

palette	Displays objects or tools you can use to build the front panel or block diagram.
plot	Graphical representation of an array of data shown either on a graph or a chart.
Positioning tool	Tool to move and resize objects.
project	A collection of LabVIEW files and files not specific to LabVIEW that you can use to create build specifications and deploy or download files to targets.
Project Explorer window	Window in which you can create and edit LabVIEW projects.
Properties dialog boxes	Dialog boxes accessed from the shortcut menu of a control or indicator that you can use to configure how the control or indicator appears in the front panel window.

pull-down menus	Menus accessed from a menu bar. Pull-down menu items are usually general in nature.
PXI	PCI eXtensions for Instrumentation. A modular, computer-based instrumentation platform.

S

sample	Single analog or digital input or output data point.
scale	Part of graph, chart, and some numeric controls and indicators that contains a series of marks or points at known intervals to denote units of measure.
shortcut menu	Menu accessed by right-clicking an object. Menu items pertain to that object specifically.
string	Representation of a value as text.
structure	Program control element, such as a Flat Sequence structure, Stacked Sequence structure, Case structure, For Loop, While Loop, or Timed Loop.
subpalette	Palette that you access from another palette that is above the subpalette in hierarchy.
subVI	VI used on the block diagram of another VI. Comparable to a subroutine.

T

task	A collection of one or more channels, timing, triggering, and other properties in NI-DAQmx. A task represents a measurement or generation you want to perform.
template VI	VI that contains common controls and indicators from which you can build multiple VIs that perform similar functions. Access template VIs from the New dialog box.
terminal	Object or region on a node through which data pass.
tip strip	Small yellow text banners that identify the terminal name and make it easier to identify terminals for wiring.
tool	Special cursor to perform specific operations.
toolbar	Bar that contains command buttons to run and debug VIs.

Traditional NI-DAQ (Legacy)	An older driver with outdated APIs for developing data acquisition, instrumentation, and control applications for older National Instruments DAQ devices. You should use Traditional NI-DAQ (Legacy) only in certain circumstances. Refer to the *NI-DAQ Readme* for more information about when to use Traditional NI-DAQ (Legacy), including a complete list of supported devices, operating systems, and application software and language versions.

V

VI	*See* virtual instrument (VI).
virtual instrument (VI)	Program in LabVIEW that models the appearance and function of a physical instrument.

W

waveform	Multiple voltage readings taken at a specific sampling rate.
waveform chart	Indicator that plots data points at a certain rate.
While Loop	Loop structure that repeats a section of code until a condition occurs.
wire	Data path between nodes.
Wiring tool	Tool to define data paths between terminals.