LabVIEW™ Core 3 Exercises

Course Software Version 2009
April 2010 Edition
Part Number 325511A-01

Worldwide Technical Support and Product Information
ni.com

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

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

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

Lesson 7
Implementing a Test Plan

Lesson 8
Evaluating VI Performance

Lesson 9
Implementing Documentation

Lesson 10
Deploying the Application

Appendix A
Additional Information and Resources

Course Evaluation

Student Guide

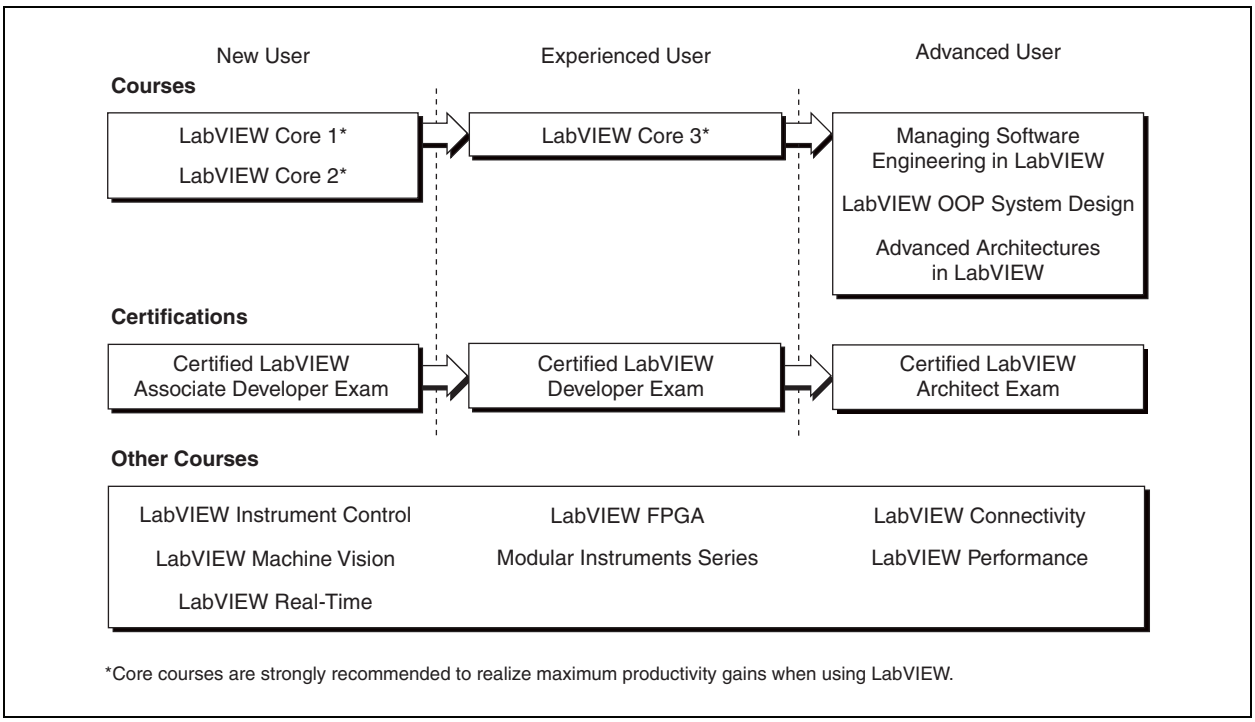
Thank you for purchasing the *LabVIEW Core 3* course kit. This course manual and the accompanying software are used in the three-day, hands-on *LabVIEW Core 3* course.

You can apply the full purchase price of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit `ni.com/training` to register for a course and to access course schedules, syllabi, and training center location information.

Note For course and exercise manual updates and corrections, refer to `ni.com/info` and enter the info code `core3`.

A. NI Certification

The *LabVIEW Core 3* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for exams to become an NI Certified LabVIEW Developer and NI Certified LabVIEW Architect. The following illustration shows the courses that are part of the LabVIEW training series. Refer to `ni.com/training` for more information about NI Certification.



B. Course Description

The *LabVIEW Core 3* course teaches you four fundamental areas of software development in LabVIEW—design, implement, test, and deploy. By the end of the *LabVIEW Core 3* course, you will be able to produce a LabVIEW application that uses good programming practices and is easy to scale, easy to read, and easy to maintain. As a result, you should be able to more effectively develop software with LabVIEW.

This course assumes that you have taken the *LabVIEW Core 1* and *LabVIEW Core 2* courses or have equivalent experience.

This course kit is designed to be completed in sequence. 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 discussion of the topics in the lesson
- A summary 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 the topics in the lesson
- Some lessons include optional and challenge exercise sections or additional exercises to complete if time permits

Note The exercises in this course are cumulative and lead toward developing a final application at the end of the course. If you skip an exercise, use the solution VI for that exercise, available in the `<Solutions>\LabVIEW Core 3` directory, in later exercises.

C. What You Need to Get Started

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

- ❑ Windows 2000 or later installed on your computer; this course is optimized for Windows XP
- ❑ LabVIEW Professional Development System 2009 or later

❑ *LabVIEW Core 3* course CD, containing the following folders:

Filename	Description
`Exercises`	Folder containing VIs and other files used in the course
`Solutions`	Folder containing 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.
2. Follow the prompts to install the course material.

The installer places the `Exercises` and `Solutions` folders at the top level of the root directory. Exercise files are located in the `<Exercises>\LabVIEW Core 3` directory.

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

E. Course Goals

This course prepares you to:

- Establish a software lifecycle for future project development
- Communicate with customers during project definition
- Develop professional user interfaces
- Develop applications that are scalable, readable, and maintainable
- Investigate and implement techniques for timing a VI
- Handle errors that may occur during code execution
- Document VIs effectively

F. Course Conventions

The following conventions are used in this course manual:

»	The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
	This icon denotes a tip, which alerts you to advisory information.
	This icon denotes a note, which alerts you to important information.
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 parameter names, controls and buttons on the front panel, dialog boxes, sections of dialog boxes, menu names, and palette names.
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 enter from the keyboard, sections of code, programming examples, and syntax examples. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

2

Analyzing the Project

Exercise 2-1 Analyze a Requirements Document

Goal

Assess a requirements document that is based on a software requirements document specification.

Scenario

You work with a Certified LabVIEW Architect to develop a requirements document. You develop the requirements document after researching commercially available theatre light control software and analyzing the specifications document. You must analyze the requirements document to ensure that it is complete and accurate.

Implementation

Analyze the requirements document for the Theatre Light Control Software.

Read the following requirements document to gain an understanding of the software you create in this course.

Many organizations use their own techniques to create a requirements document. If your organization is not using a format for a requirements document, you can use this requirements document as a basis for other requirements documents. Refer to the *IEEE Requirements Documents* section of Appendix A of the *LabVIEW Core 3 Course Manual* for another version of this requirements document.

Start of Requirements Document

Requirements Document

ABC Theatre Inc.

Theatre Light Control Software Specifications

Document Number—LV100975

Requirements Document Continued

Section I: General Requirements

The application should do the following:

- Function as specified in *Section II: Application Requirements* of this document.
- Conform to LabVIEW coding style and documentation standards. Refer to the *Development Guidelines* topic of the *LabVIEW Help* for more information about the LabVIEW style checklist and creating documentation.
- Be hierarchical in nature. All major functions should be performed in subVIs.
- Use a state machine that manages states with a type-defined enumerated type control, a queue, or an Event structure.
- Be easily scalable, enabling the addition of more states and/or features without having to manually update the hierarchy.
- Minimize the excessive use of structures, local and/or global variables, and Property Nodes.
- Respond to front panel controls within 100 ms and not utilize 100% of CPU time.
- Close all opened references and handles.
- Be well documented and include the following:
  - Labels on appropriate wires within the main VI and subVIs.
  - Descriptions for each algorithm.
  - Documentation in **VI Properties»Documentation** for the main VI and subVIs.
  - Tip strips and descriptions for each front panel control and indicator.
  - Labels for constants.

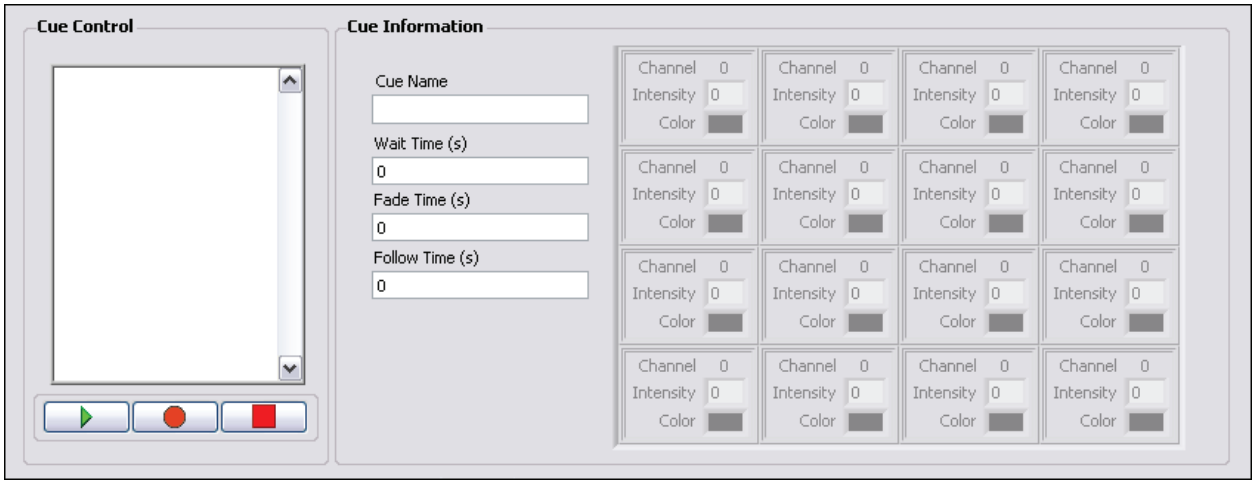
Section II: Application Requirements

Introduction

ABC Theatre Lighting Inc. is the largest provider of theatre lighting systems for major metropolitan theatres worldwide. Theatre light systems must be scalable for as many lights as a particular production might require. A software-based theatre light control system allows theatres to scale the lighting for each production. The control system controls each light individually. Each light contains its own dimmer and color mixing system.

Requirements Document Continued

The color mixing system can mix an appropriate amount of red, green, and blue to define each color. The control software sends signals to a hardware control system that controls the intensity and color of the lights. The user interface for the control software should look similar to the following front panel.



Definitions

This section defines the terminology for the project.

- **Channel**—The most basic element of the Theatre Light Control Application. Each channel corresponds to a physical light.
- **Intensity**—Attribute of the channel that defines the intensity of the physical light.
- **Color**—Attribute of the channel that defines the color of the channel as a combination of red, green, and blue.
- **Cue**—A cue contains any number of independent channels with timing attributes for the channels.
- **Wait time**—A cue timing attribute that defines the amount of time to wait, in multiples of one second, before the cue fires.
- **Fade time**—A cue timing attribute that defines the time it takes, in multiples of one second, before a channel reaches its particular intensity and color.
- **Follow time**—A cue timing attribute that defines the amount of time to wait, in multiples of one second, before the cue finishes.

Requirements Document Continued

Task

Design, implement, test, and deploy a theatre light control system that allows a theatre lighting engineer to easily control and program the theatre lights for any production.

General Operation

Front panel controls determine the operation of the theatre light control software. Front panel indicators display the current status of the theatre light control software.

The controller will store the channel intensity, channel color, channel wait time, channel fade time, channel follow time, and name for the cue when the user clicks the **Record** button. When the user clicks the **Play** button, the controller services each cue in the Cue Control by cycling through the recorded cues starting with the first cue in the Cue Control. A cue that is playing will wait for the specified wait time, then fade the channels to the desired color and intensity within the specified fade time, and then wait for the specified follow time. The next cue in the Cue Control loads and the process repeats, until all of the Cues play. The user can stop a currently playing cue by clicking the **Stop** button. The controller exits when the user selects **File»Exit**.

Sequence of Operation

Application Run

When the application starts, all the front panel controls must initialize to their default states. The **Cue Control** must be cleared to remove all the recorded Cues. The channels must be initialized with their corresponding channel number, zero intensity, and the color black.

Record

Click the **Record** button to activate the cue recording functionality. A custom panel must open that allows the lighting engineer to set the channel intensity and color for the channels. The panel must provide for the capability to name the cue, and specify the wait time, fade time, and the follow time. The minimum time for the wait time and follow time is zero seconds. The minimum time for the fade time is one second. The minimum increment for the wait time, fade time, and follow time is one second.

After a cue is recorded, the cue name is placed into the Cue Control.

Requirements Document Continued

Play

Click the **Play** button to play the recorded cues. When the play begins, the controller should disable the **Record** button and the Cue List. The values of the first cue in the cue list are loaded into memory. The controller waits based on the number of seconds specified for the wait time for the current cue. The controller then fades the channel up or down based on the current channel intensity and the desired channel intensity. The software writes the color and intensity to the theatre lighting hardware control system, and updates the front panel channels. The controller must finish fading within the specified fade time. The controller will finish processing the cue by waiting for the number of seconds specified for the follow time of the current cue. When the play is complete, the controller should enable the **Record** button and the Cue List.

Stop

Click the **Stop** button to stop a currently playing cue. The operation is ignored if a cue is not playing.

Save

Select **File»Save** to save all of the recorded cues in a file for later playback. The user specifies the filename.

Open

Select **File»Open** to open a file that contains recorded cues. The user specifies the filename.

Exit

Select **File»Exit** to exit the application. If an error has occurred in the application, the application reports the errors.

Description of Controls and Indicators

Control Name	Control Description—Function
Cue List	Listbox—Stores a list of recorded cues that the user can select
Play	Boolean—Plays the recorded cues
Record	Boolean—Opens a dialog box that allows the user to specify and record channel attributes
Stop	Boolean—Stops a currently playing cue

Requirements Document Continued

Indicator Name	Indicator Description—Function
Cue Name	String—Displays the name of the current cue
Wait Time	Numeric—Displays the number of seconds of the recorded cue wait time
Fade Time	Numeric—Displays the number of seconds of the recorded cue fade time
Follow Time	Numeric—Displays the number of seconds of the recorded cue follow time
Channel	Cluster—Record that contains the channel number, channel intensity, and channel color

Scalability

Many of the newer theatre lights systems provide motor control to move the light around the stage. The Theatre Light Control Software should provide for the ability to easily implement channel pan and tilt. The software should be easily scalable to control any number of channels.

Documentation

The application documentation should address the needs of the end user and a programmer who might modify the application in the future.

Deliverables

The project includes the following deliverables:

- Documented source code
- Documentation that describes the system

Timeline

The project has the following timeline for completion:

- Day 1—User Interface prototype completed
- Day 2—Application modules completed
- Day 3—Fully functional application

End of Requirements Document

Use the following Requirements Document Checklist to ensure the requirements document is complete and adequate.

- ❑ Each requirement is clear and understandable.
- ❑ Each requirement has a single, clear, unambiguous meaning.
- ❑ The requirements explain every functional behavior of the software system.
- ❑ The requirements do not contradict each other.
- ❑ The requirements are correct and do not specify invalid behavior.
- ❑ Each requirement is testable.

End of Exercise 2-1

Notes

3

Designing the User Interface

Exercise 3-1 User-Interface Design Techniques

Goal

Learn techniques you can use to create professional user interfaces in LabVIEW.

Description

LabVIEW includes features that allow you to create professional user interfaces. Learn techniques to remove borders from clusters, create custom cursors, create custom toolbars, and use the transparency property to enhance the user's experience with your application.

Implementation

Creating Transparent Cluster Borders

Clusters group data elements of mixed types. However, sometimes you do not want the user to know that you have organized the data into a cluster. The clusters from the **Modern** palette visually indicate that the data is stored in a container, however, you can modify a cluster from the **Classic** palette to conceal the fact that data is organized in a cluster.

1. Open a blank VI.
2. Create a cluster from the **Classic** palette and make the borders of the cluster transparent.

Tip If the **Controls** palette is not visible, select **View»Controls Palette** to display the palette.

Tip You can use the **Search** button to find controls, VIs, or functions. On the **Controls** or **Functions** palette, click the **Search** button on the palette toolbar. In the text field at the top of the search dialog, start typing the name of the control, VI, or function you want to find. As you type, the listbox shows all possible matches. When you find the control, VI, or function you are looking for, double-click its name. This opens the palette where the control, VI, or function is located and flashes the icon whose name you just double-clicked.

❑ Add a cluster from the **Classic** palette to the front panel.

Tip Select **Tools»Options**, select **Block Diagram** from the **Category** list, and remove the checkmark from the **Place front panels terminals as icons** checkbox to match the examples in the manual.

❑ Click a blank area of the front panel with the Get Color tool to copy the color of the front panel to the foreground and background colors the Coloring tool uses.

Tip If the **Tools** palette is not visible, select **View»Tools Palette** to display the palette.

❑ Select the Coloring tool. In the Coloring tool section of the **Tools** palette, click the top left (foreground) color block to open the color picker.

❑ Click **T** in the upper right corner of the color picker to change the foreground color to transparent.

❑ Click the border of the cluster with the Coloring tool as shown in Figure 3-1 to make the cluster border transparent.

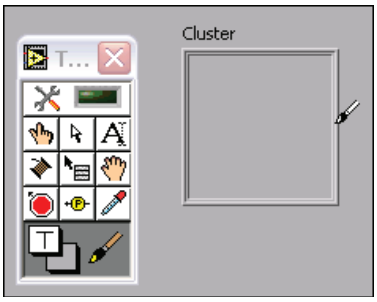


Figure 3-1. Transparent Cluster Borders

You can use this technique with other controls from the **Classic** palette. The controls from the **Classic** palette are easier to modify and customize than controls from the **Modern** palette.

To make the border of a **Modern** control transparent, select the Coloring tool and right-click the border of the control to open the color picker. With the color picker open, press the space bar to toggle the color selection to the foreground. Click **T** in the upper right corner of the color picker to change the foreground color to transparent.

3. Save the VI as `Transparent Cluster Borders.vi` in the `<Exercises>\LabVIEW Core 3\User Interface Design Techniques` directory.

4. Close the VI.

Creating Custom Cursors

You can change the appearance of front panel cursors for a VI. LabVIEW provides tools that allow you to use system cursors, or even define your own custom cursors. Changing the appearance of the cursor in your application provides visual cues to the user on the status of the application. For example, if your application is busy processing data you can programmatically set the cursor to busy while the processing occurs to let the user know that the application is processing. Create a simple VI to test the cursor functionality.

1. Create a VI that contains a While Loop and the Set Cursor VI to change the appearance of the cursor as shown in Figure 3-2.

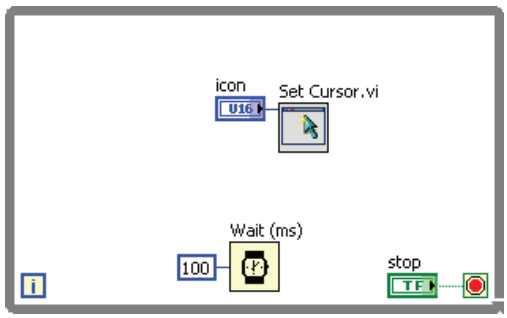


Figure 3-2. VI to Change the Cursor

   - ❑ Open a blank VI.
   - ❑ Add a While Loop to the block diagram.
   - ❑ Right-click the loop conditional terminal and select **Create» Control** from the shortcut menu to create a stop button for the VI.
   - ❑ Add a Wait (ms) function inside the While Loop, and set the wait to a reasonable amount, such as `100` ms.
   - ❑ Add the Set Cursor VI from the **Cursor** palette to the While Loop.
   - ❑ Right-click the **icon** input of the Set Cursor VI and select **Create» Control** from the shortcut menu.

2. Save the VI as `Custom Cursors.vi` in the `<Exercises>\LabVIEW Core 3\User Interface Design Techniques` directory.
3. Switch to the front panel and run the VI.
   - ❑ Change the icon in the icon ring while the VI runs.
4. Close the VI.

Creating Custom Toolbars

Many professional applications include custom toolbars. Providing a toolbar for your application increases the usability of your application. You can use a splitter bar to create a custom toolbar.

To create a toolbar at the top of your VI, add a Horizontal Splitter Bar to the front panel and add a set of controls to the upper pane. Right-click the splitter bar and select the following items from the shortcut menu to configure the splitter bar as a toolbar:

- Select **Locked** and **Splitter Sizing»Splitter Sticks Top** to lock the splitter bar in position.
- Select **Upper Pane»Horizontal Scrollbar»Always Off** and **Upper Pane»Vertical Scrollbar»Always Off** to hide the upper scrollbars of the pane.

You also can paint the pane and resize the splitter so that it blends seamlessly with the menu bar. You can scroll the scrollbars of the lower pane or split the lower pane further without affecting the toolbar controls. Figure 3-3 shows examples of custom toolbars.

Figure 3-3. Custom Toolbar Examples

Complete the following steps to implement the simple custom toolbar shown in Figure 3-4.

Figure 3-4. Custom Word Processor Toolbar

1. Open a blank VI.
2. Add a Horizontal Splitter Bar from the **Containers** palette to the front panel. Position the splitter bar near the top of the VI. Leave enough space above the bar to add the menu controls.
3. To turn off the scrollbars, right-click the splitter bar and select **Upper Pane»Horizontal Scrollbar»Always Off** and **Upper Pane»Vertical Scrollbar»Always Off** from the shortcut menu.

4. To change the style of the splitter bar to system, right-click the splitter bar and select **Splitter Style»System** from the shortcut menu.

5. Add the toolbar controls from the `<Exercises>\LabVIEW Core 3\User Interface Design Techniques\Toolbar Controls` directory to the upper pane created by the splitter bar.

6. Rearrange the controls and color the splitter bar to look like a toolbar.

   ❑ Hide the labels for the controls.

   ❑ Use the **Align Objects** and **Distribute Objects** buttons on the toolbar to align and distribute the controls.

   ❑ Color the background of the pane to blend the controls into the panel.

   – Click the background color block of the Coloring tool to open the color picker.

   – Click the **More Colors** button in the bottom right corner of the color picker to open the **Color** dialog box.

   – Enter the following values and click **OK** to set the background color:

     Red: `231`

     Green: `223`

     Blue: `231`

   – Click the background of the splitter bar pane to change the color.

7. To lock the splitter bar so that the user cannot move it, right-click the splitter bar and make sure **Locked** is checked in the shortcut menu.

8. Save the VI as `Custom Toolbars.vi` in the `<Exercises>\LabVIEW Core 3\User Interface Design Techniques` directory.

9. Close the VI.

Creating Transparent Controls

You can use transparent controls to make the user interface more professional. Modify an existing VI that uses a Tank indicator to display fluid levels. To obtain the current value of the Tank, the user must click the Tank. To modify this VI, place a Boolean control on top of the Tank indicator and change the colors of the Boolean control to transparent.

1. Add a Boolean button to the existing Tank Value VI.

   ❑ Open `Tank Value.vi` from the `<Exercises>\LabVIEW Core 3\User Interface Design Techniques` directory.

   ❑ Add a Flat Square button from the **Classic** palette and position it on top of the Tank. Resize the control to completely cover the Tank and hide the label of the control.

2. Modify the button to make it transparent.

   ❑ Use the Coloring tool to change the True and False color of the button to transparent.

   ❑ Click the button with the Operating tool to verify that the button is transparent whether it is True or False.

3. Add code to cause **Tank Value** to update only when the transparent button is True, as shown in Figure 3-5.

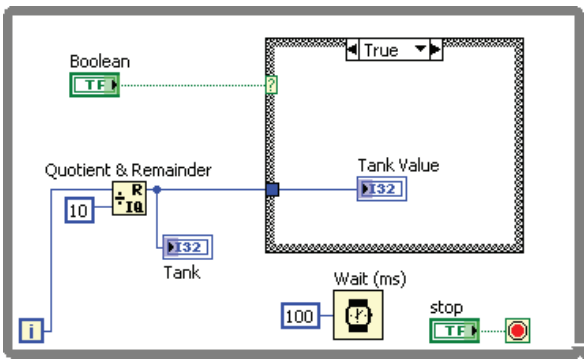


Figure 3-5. Tank Value Modification

   ❑ Add a Case structure inside the While Loop to enclose the Tank Value indicator.

   ❑ Wire the Boolean control to the case selector terminal.

   ❑ Leave the False case empty.

4. Save the VI.

5. Run the VI and test the behavior of the VI when you click the **Tank** indicator.

6. Stop and close the VI.

End of Exercise 3-1

Notes

Notes

4

Designing the Project

Exercise 4-1 User Event Techniques

Goal

Complete a VI that contains a static user interface event and a user event.

Scenario

This VI contains the **Fire Event** Boolean control that causes an LED to light when the user clicks the control. In addition, the block diagram contains a countdown that displays on the slider. When the countdown reaches zero, it generates a programmatic event that lights the LED.

Design

1. Modify the block diagram to create and generate a user event for the LED.
2. Configure the Fire Event event case to handle both the Value Change event on the **Fire Event** Boolean control and the User event.

Note Event-driven programming features are available only in the LabVIEW Full and Professional Development Systems. You can run a VI built with these features in the LabVIEW Base Package, but you cannot reconfigure the event-handling components.

Implementation

1. Open `User Event.vi` from the `<Exercises>\LabVIEW Core 3\User Event Techniques` directory. Figure 4-1 and Figure 4-2 show the front panel and block diagram.

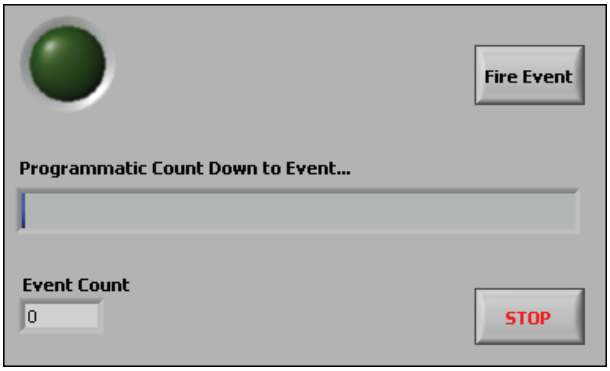


Figure 4-1. User Event VI Front Panel

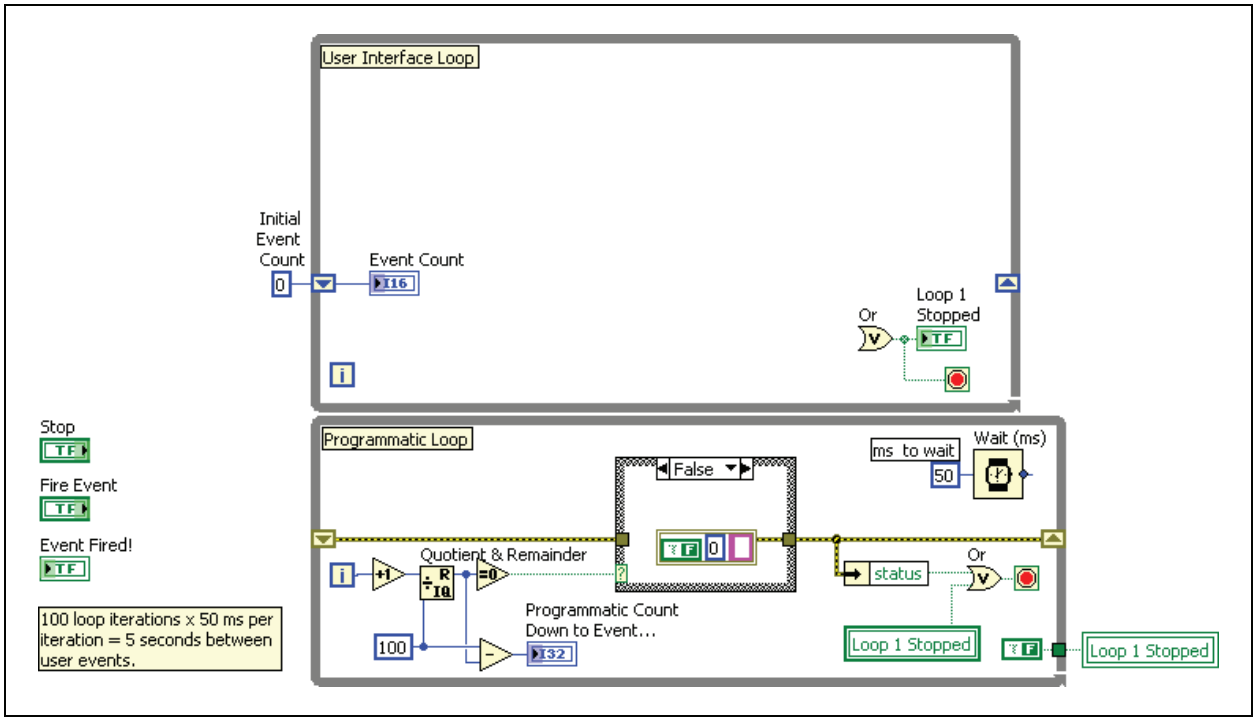


Figure 4-2. User Event VI Block Diagram

Create and Generate User Event

2. Modify the block diagram to create and generate a user event for the LED as shown in Figure 4-3. You will add the True state as shown in Figure 4-3. Wire the user event through the False state.

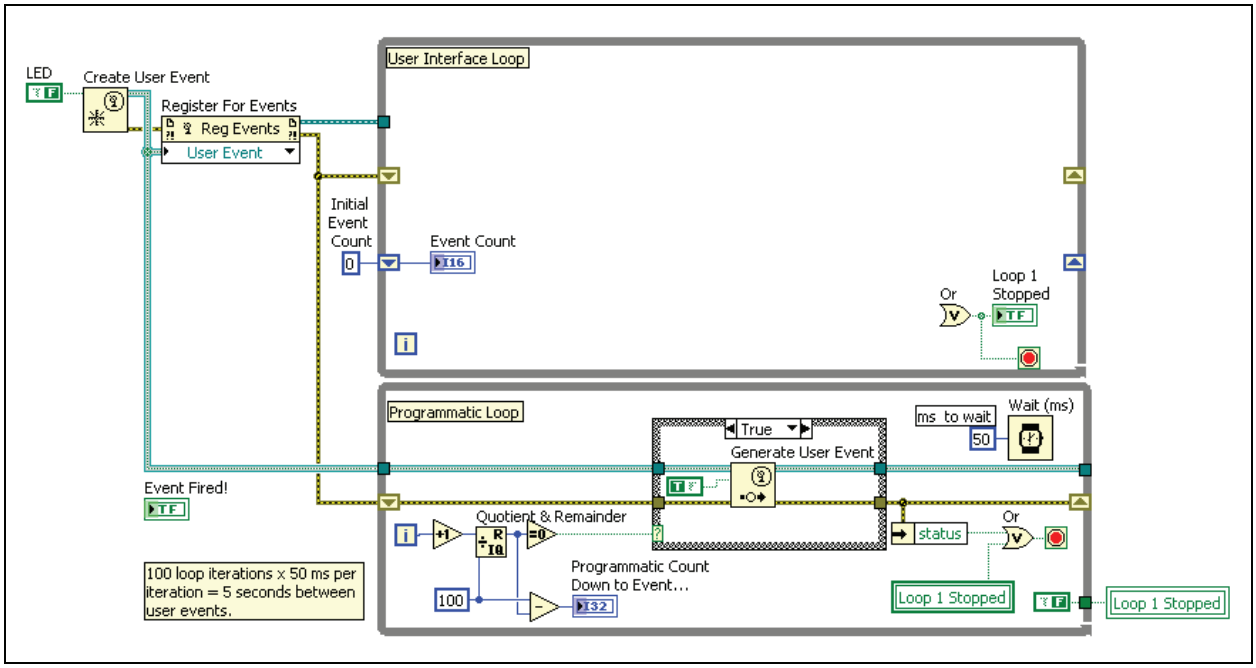


Figure 4-3. Create and Generate a User Event for LED

- ❑ Add a Create User Event function from the **Events** palette to the block diagram. This function returns a reference to a user event when **Programmatic Count Down to Event** reaches zero.
- ❑ Add a False constant from the **Boolean** palette to the block diagram. Label it `LED`. Wire the False constant to the **user event data type** input of the Create User Event function.

- ❑ Add a Register For Events node from the **Events** palette to the block diagram. Wire the node as shown in Figure 4-3.

- ❑ Generate the event within the True case of the programmatic loop.
  - Add a Generate User Event function from the **Events** palette to the block diagram.
  - Add a True constant to the block diagram and wire it to the **event data** input of the Generate User Event function.
  - Wire the Generate User Event function as shown in Figure 4-3. The True case executes only when the countdown reaches zero.
  - Wire the user event reference through the False case of the Case structure.

Configure Events

3. Create and configure an event case to handle the value change event on the **Fire Event** Boolean control and handle the user event.

- ❑ Add an Event structure from the **Structures** palette to the user interface loop.
- ❑ Wire the **Event Registration Refnum** from the Register For Events node to the dynamic event terminal of the Event structure.

Tip If the dynamic event terminal is not visible, right-click the border of the Event structure and select **Show Dynamic Event Terminals** from the shortcut menu.

- ❑ Wire the error and Event Count shift registers from the left border of the User Interface Loop to the left border of the Event structure.

4. Configure a Fire Event case to handle both the Value Change event on the **Fire Event** Boolean control and the user event, as shown in Figure 4-4.

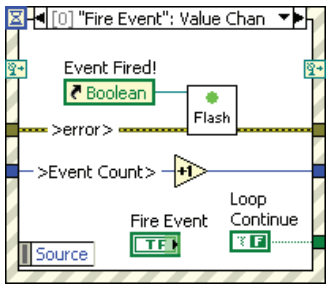


Figure 4-4. Fire Event Case

- ❑ Add the Flash LED VI from the `<Exercises>\LabVIEW Core 3\User Event Techniques` directory to the event case. This subVI turns on the LED for 200 ms.
- ❑ Right-click the Event Fired! terminal and select **Create»Reference** from the shortcut menu to create a reference to the indicator.
- ❑ Wire the Event Fired! reference to the **Bool Refnum** input of the Flash LED subVI.
- ❑ Move the Fire Event Boolean control into the event case so the VI reads the value of the control when the event executes.
- ❑ Add an Increment function to the block diagram to increment the event count within the event case.
- ❑ Add a False constant to the block diagram. Label it `Loop Continue`. Wire the False constant to the right border of the Event structure.
- ❑ Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu to open the **Edit Events** dialog box.
  - – Select **Fire Event** from the **Event Sources** list and select **Value Change** from the **Events** list.
  - – Click the blue **+** to add an event.
  - – Select **Dynamic»<LED>:User Event** from the **Event Sources** list.
  - – Click **OK** to complete configuration.

5. Create and configure the Stop event case to handle the Value Change event on the **Stop** Boolean control as shown in Figure 4-5.

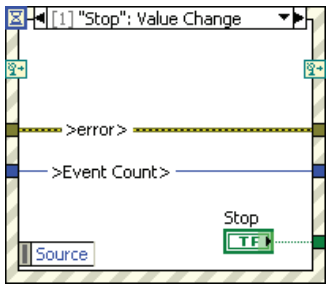


Figure 4-5. Stop Event Case

   - ❑ Right-click the Event structure and select **Add Event Case** from the shortcut menu to open the **Edit Events** dialog box.
   - ❑ Select **Stop** from the **Event Sources** list and **Value Change** from the **Events** list and click **OK**.
   - ❑ Move the **Stop** Boolean control into the Stop event case and wire the control to the right border of the Event structure.
   - ❑ Wire the error and Event Count data through the Stop event case.

6. Complete the block diagram as shown in Figure 4-6 to stop the User Interface Loop, release the user event resources, and handle any errors that have occurred. Use the following items:

   - ❑ Unbundle By Name function—Extracts the **status** Boolean data from the error cluster. If an error has occurred or the user clicks **Stop**, then stop the User Interface Loop.
   - ❑ Unregister For Events function—Releases the resources used to register the user event.
   - ❑ Merge Errors VI—Combines the error outputs from both loops into a single error output.
   - ❑ Destroy User Event function—Releases the resources that were allocated for the user event.
   - ❑ Simple Error Handler VI—Communicates error information to the user.

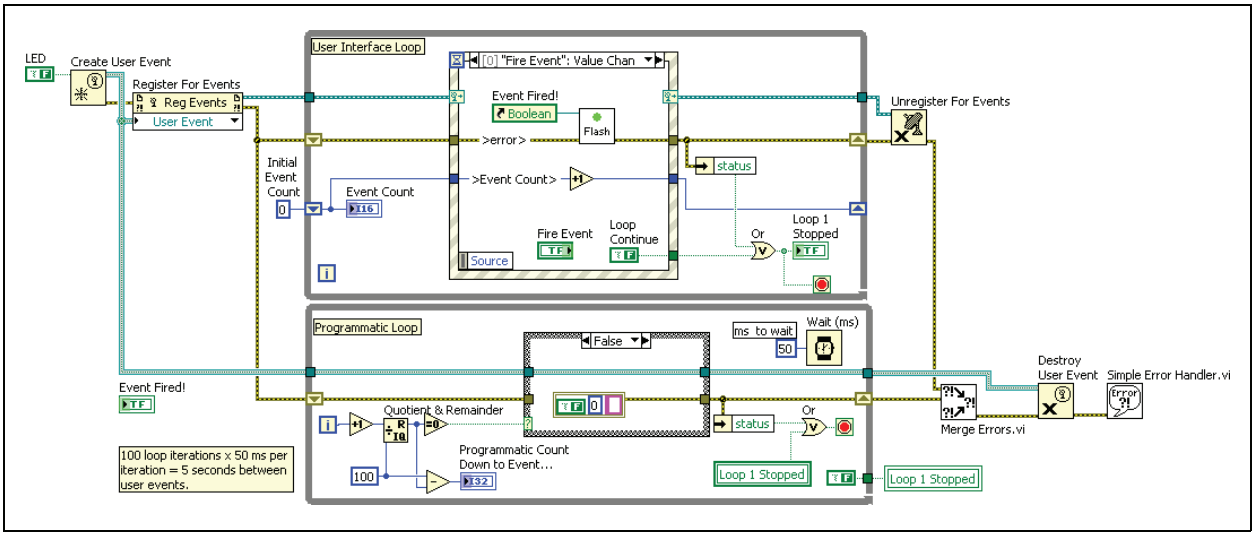


Figure 4-6. Completed User Event VI Block Diagram

7. Save the VI.

Testing

1. Run the VI. Try to generate a user interface event at the same time as a programmatic event. Does the VI record both events?

2. Close the VI.

End of Exercise 4-1

Exercise 4-2 Using the LabVIEW Project

Goal

Create a LabVIEW project for the application.

Scenario

Every large LabVIEW development needs to use a project to control naming and project hierarchy. Using the LabVIEW Project simplifies the development of larger applications.

Design

Create a LabVIEW project that includes folders for modules and controls. Save the project as `TLC.lvproj` in the `<Exercises>\LabVIEW Core 3\Course Project` directory.

Implementation

1. Create a new project.
   - ❑ Select **File»New Project** to open the **Project Explorer** window.
2. Create virtual folders for modules and controls in the **My Computer** hierarchy. You use these virtual folders later in the course.
   - ❑ Right-click **My Computer** in the LabVIEW Project and select **New»Virtual Folder** from the shortcut menu to create a new virtual folder.
   - ❑ Name the virtual folder `Modules`.
   - ❑ Repeat the previous steps to create the **Controls** virtual folder.
3. Save the project as `TLC.lvproj` in the `<Exercises>\LabVIEW Core 3\Course Project` directory.
4. Close the project.

End of Exercise 4-2

Exercise 4-3 Using Project Explorer Tools

Goal

Use the **Project Explorer** tools to resolve conflicts and manage files in a LabVIEW project.

Scenario

Conflicts can arise within a LabVIEW project if top-level VIs are calling incorrect versions of nested code. Applications that are saved in multiple locations as a result of archiving, backup, or division of work can lead to the use of incorrect code and broken applications.

In this exercise, you examine a LabVIEW project that contains conflicts and use the tools in the **Project Explorer** to resolve the conflicts and manage the project.

Design

The project in this exercise contains the following conflicts:

- Two VIs within the project have the same name, `Generate Signal.vi`.
- A VI in the project calls a subVI outside the project that has the same name, `Log to File.vi`, as a VI within the project.

Implementation

Part I: Resolving Conflicts

1. Explore a LabVIEW Project containing conflicts.

   ❑ Open `Conflicts.lvproj` in the `<Exercises>\Project Explorer Tools` directory.

   ❑ Expand **Sine Wave**, **Square Wave**, **File IO**, and **Dependencies** in the project tree, as shown in Figure 4-7.

   Notice that LabVIEW has determined that various VIs have conflicts. A conflict is a potential cross-link that occurs when LabVIEW tries to load a VI that has the same qualified name as an item already in the project. When there is a conflict, it is unclear which VI a calling VI should reference.

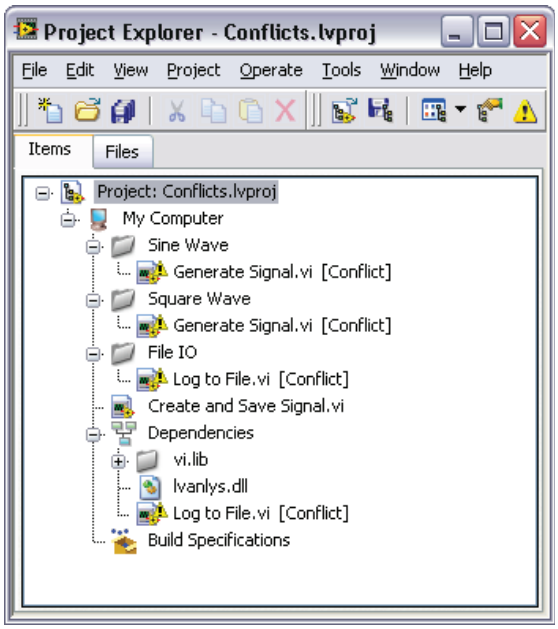


Figure 4-7. LabVIEW Project with Conflicts

- ❑ Double-click **Generate Signal.vi** in the **Sine Wave** virtual folder.
- ❑ Run the VI and observe that this VI generates a sine wave.
- ❑ Close the VI.
- ❑ Double-click **Generate Signal.vi** in the **Square Wave** virtual folder.
- ❑ Run the VI and observe that this VI generates a square wave.
- ❑ Close the VI.

2. View the file paths of the items in the project tree.

   - ❑ In the **Project Explorer** window, select **Project»Show Item Paths**.

Tip The best way to determine if cross-linking exists is to view the full path to the item. Viewing filepaths is often the first step when attempting to resolve conflicts that are caused by cross-linking. You can attempt to rename or move files as needed, but first you must determine which file is the correct file. Enabling **Show Item Paths** displays the file locations in a second column in the **Project Explorer** window.

You can resolve project conflicts from the **Resolve Project Conflicts** dialog box, but in this exercise you practice additional techniques to resolve conflicts.

3. Determine which VIs in the project call the conflicting VIs.

   - ❑ Right-click **Generate Signal.vi** in the **Sine Wave** virtual folder and select **Find»Callers** from the shortcut menu. In the project tree, LabVIEW highlights the Create and Save Signal VI because it calls this VI as a subVI.

   - ❑ Right-click **Generate Signal.vi** in the **Square Wave** virtual folder and select **Find»Callers** from the shortcut menu. Notice that this VI has no callers in the project. However, it is inadvisable to delete this VI because it performs a unique function. Renaming one or both files is a more appropriate action.

4. Manually rename the conflicting files.

   - ❑ In the **Sine Wave** folder, right-click **Generate Signal.vi** and select **Rename**.

   - ❑ Rename the VI `Sine Wave - Generate Signal.vi` and click **OK**.

   - ❑ LabVIEW prompts you to save the changes made to the calling VI, `Create and Save Signal.vi`, to preserve the links. Click **Save**.

   - ❑ In the **Square Wave** folder, right-click **Generate Signal.vi** and select **Rename**.

   - ❑ Rename the VI `Square Wave - Generate Signal.vi` and click **OK**.

Note You also can rename files from the **Files** page view of the project.

5. Resolve a conflict using the **Resolve Project Conflicts** dialog box.

   - ❑ Notice that there is copy of `Log to File.vi` in the **File IO** virtual folder and a copy of `Log to File.vi` in **Dependencies**, which indicates that a file within the project calls the second copy. The filepaths of these two VIs are different, as shown in the **Paths** section of the **Project Explorer** window.

   - ❑ Right-click each copy of `Log to File.vi` and select **Find» Callers** to determine which file, if any, within the project calls each copy.

❑ Double-click **Create and Save Signal.vi** in the LabVIEW Project. The **Resolve Load Conflict** dialog box appears as shown in Figure 4-8 and prompts you to select the subVI you want the caller to reference.

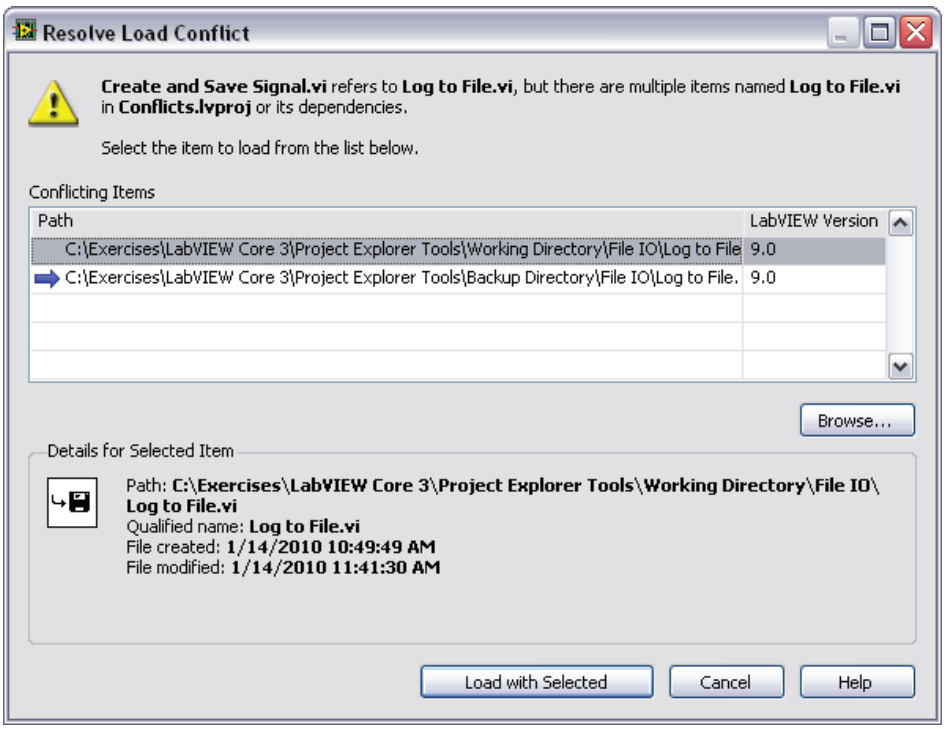


Figure 4-8. Resolve Load Conflict Dialog Box

❑ In the **Resolve Load Conflict** dialog box, select the Log to File VI in `Working Directory` and click **Load with Selected**.

❑ The **Load Summary Warning** dialog box informs you that the Log to File subVI was loaded from a different location. Click **Show Details**.

❑ Examine the **Load and Save Warning List** dialog box. Click **Close**. All conflicts in the project should now be resolved.

❑ The Create and Save Signal VI has unlinked subVIs after resolving the conflicts. To relink the subVIs, open the block diagram, right-click each subVI and select **Relink to SubVI** from the shortcut menu.

❑ Save and close the Create and Save Signal VI.

6. Save the project.

Part II: Other File Management Tools

1. Use auto-populating folders in the project.

   - ❑ Right-click the **Sine Wave** virtual folder in the project tree and select **Convert to Auto-populating Folder** from the shortcut menu.
   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Project Explorer Tools\Working Directory\Sine Wave` directory and click **Current Folder**. Notice in the project tree that the Sine Wave folder icon changes to indicate that the folder is now set to auto-populate.

Note In auto-populate mode, the contents of the project folder reflect the hierarchy of the specified folder on disk, as well as any changes that are made outside the development environment.

   - ❑ Click the Windows **Start** button and select **All Programs»Accessories»Notepad** to launch Notepad.
   - ❑ In Notepad, select **File»Save** and save the file as `Data File.txt` in the `<Exercises>\LabVIEW Core 3\Project Explorer Tools\Working Directory\Sine Wave` directory.
   - ❑ Close Notepad.
   - ❑ Notice that `Data File.txt` has been added to the **Sine Wave** auto-populating folder on the **Items** page of the **Project Explorer** window.

2. Search for project items.

   - ❑ In the **Project Explorer** window, select **Edit»Find Project Items**.
   - ❑ In the **Find Project Items** dialog box, enter `sine` in the textbox, as shown in Figure 4-9, and click **Find**.

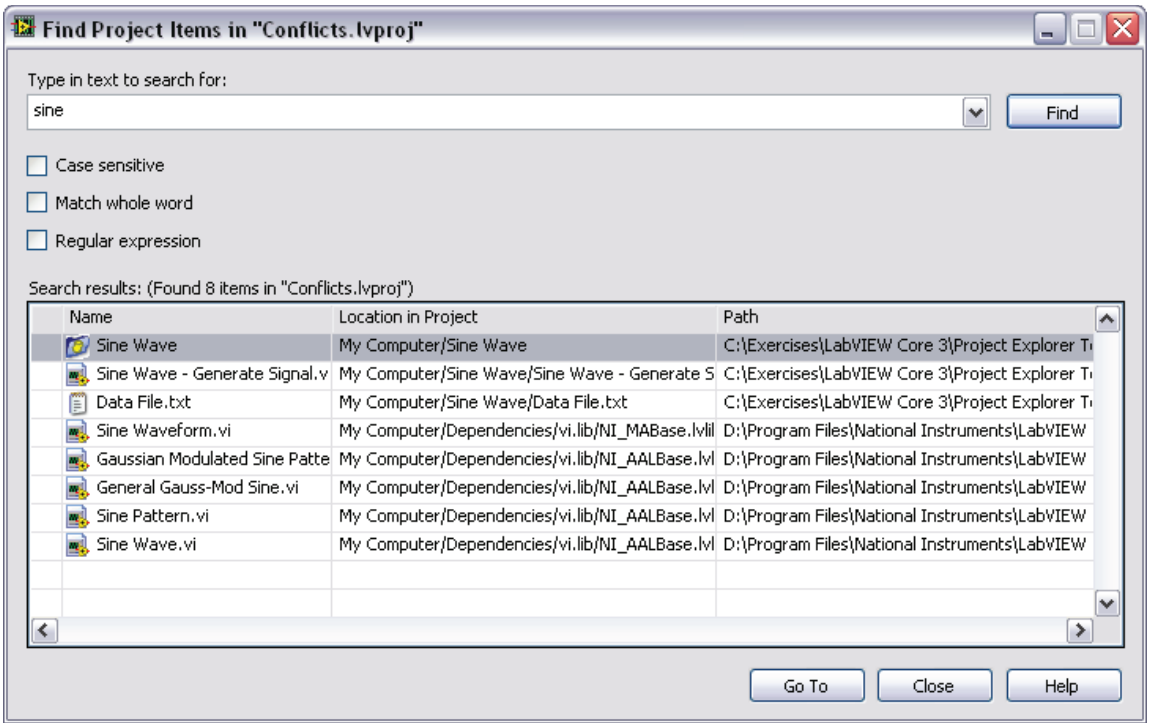


Figure 4-9. Find Project Items Dialog Box

- ❑ Select **Sine Wave - Generate Signal.vi** and click **Go To**. This item should now be highlighted in the **Project Explorer** window.

3. Save and close the project.

End of Exercise 4-3

Exercise 4-4 Choose Data Types

Goal

Design and create the data types that you need for this application.

Scenario

When you develop a VI, you must identify and design the data types that you need to use in the application.

Design

Design a cluster called Cue for the cue information that contains the following data elements:

- Cue Name (string)
- Wait Time (32-bit unsigned integer)
- Fade Time (32-bit unsigned integer)
- Follow Time (32-bit unsigned integer)
- Channels (2D array of `channel.ctl`)

The final cluster should resemble Figure 4-10. Use controls from the **System** palette where appropriate, so your application will look like the native operating system when moved to different locations.

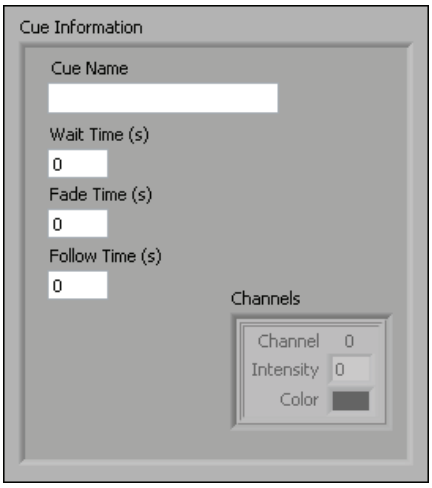


Figure 4-10. Cue Information Cluster

Implementation

1. Open the TLC project.
2. Add `tlc_Cue_Information.ctl` to the **Controls** virtual folder in the TLC project.
   - ❑ Right-click the **Controls** virtual folder in the TLC project tree and select **Add»File** from the shortcut menu.
   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory, select `tlc_Cue_Information.ctl` and click **Add File** to add the file.
3. Open `tlc_Cue_Information.ctl` and verify that the **Control Type** pull-down menu is set to **Type Def**.
4. Create the Cue Name, Wait Time, Fade Time, and Follow Time controls. Use the **System** palette to create the controls. Add the controls to the cluster.
   - ❑ Add a String control to the cluster and label the control `Cue Name`.
   - ❑ Add a Numeric control to the cluster and label the numeric `Wait Time (s)`.
   - ❑ Right-click the numeric and select **Representation»Unsigned Long (U32)** from the shortcut menu.
   - ❑ Create two copies of the Wait Time (s) control in the cluster. Name one `Fade Time (s)` and name one `Follow Time (s)`.
5. Create the Channels 2D array shell. Turn off index display for the array.
   - ❑ Add an array from the **Modern** palette to the cluster. Change the label of the array to `Channels`.
   - ❑ Right-click the array shell and select **Add Dimension** from the shortcut menu to make the array 2D.
   - ❑ Right-click the array shell and select **Visible Items»Index Display** to turn off index display.
6. Add the `channel.ctl` file from the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory to the **Controls** virtual folder of the TLC project.

7. Click and drag `channel.ctl` from the **Project Explorer** window to the Channel array shell you created in step 5.
8. Save and close the tlc_Cue_Information control.
9. Save the project.

End of Exercise 4-4

Exercise 4-5 Information Hiding

Goal

Design a VI that provides an interface to the data.

Scenario

Build a VI that uses a functional global variable to provide an interface to the Cue data type you created in Exercise 4-4. The functional global variable provides a safe way to access the data that the application needs.

Design

To provide an interface to the data in the Cue data type, you need to create a VI that can access the Cue data type. Create a functional global variable to access the data.

The functional global variable implements the following functions:

- Initialize
- Add Cue
- Get Cue Values
- Set Cue Values
- Get Number of Cues
- Get Empty Cue

Implementation

1. Open the TLC project if it is not already open.
2. Add the cue module files to the TLC project.
   - ❑ Right-click **Modules** in the project tree and select **Add»Folder (Snapshot)** from the shortcut menu.
   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Modules\Cue` folder and click **Current Folder** to add the folder and its contents to the project tree.
3. Open `tlc_Cue Module.vi`. The controls, indicators, and structures have already been added to the VI. Notice that the VI uses the custom control you created in Exercise 4-4.

4. Set the default value of the **Fade Time (s)** input to `1`.

   ❑ Set **Fade Time (s)** in the Cue Input cluster to `1`.

   ❑ Right-click the **Fade Time (s)** control and select **Data Operations»Make Current Value Default** from the shortcut menu.

5. Complete the Initialize case, as shown in Figure 4-11. This case executes when the Theatre Light Control software starts.

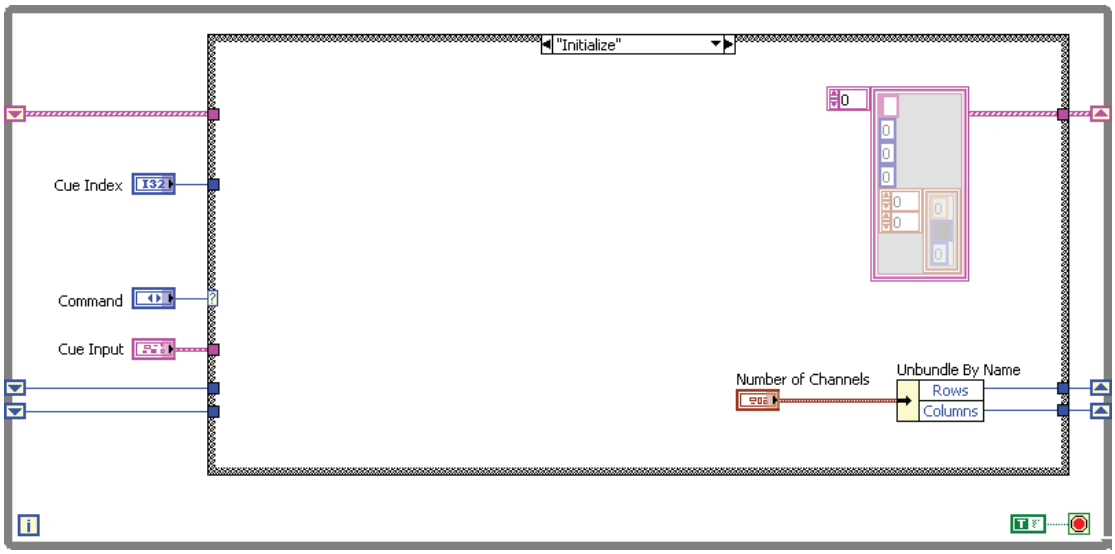


Figure 4-11. Initialize Case

   ❑ Add an Unbundle By Name function to the Initialize case to extract the Rows and Columns elements from the Number of Cues cluster.

   ❑ Add an empty Array Constant to the Initialize case.

   ❑ Drag `tlc_Cue_Information.ctl` from the **Project Explorer** to the array shell to create the Cue Information constant.

   ❑ Right-click the constant and select **Data Operations»Empty Array**.

   ❑ Wire the constant to an output tunnel on the While Loop.

   ❑ Create shift registers on the While Loop tunnels for Cue Information, Rows, and Columns.

- ❑ Right-click the Row output tunnel on the Case structure and select **Linked Input Tunnel»Create and Wire Unwired Cases** from the shortcut menu. Click the corresponding input tunnel to link it to the output tunnel. LabVIEW automatically wires the linked tunnels in all existing cases. If you add a case, LabVIEW wires the tunnels in the new case.

- ❑ Repeat the previous step for the Columns output tunnel.

Note The True constant wired to the loop conditional terminal causes the loop to iterate only once each time it is called.

6. Complete the Add Cue case as shown in Figure 4-12 using the following items. This case adds a new cue to the Cue List.

- ❑ Build Array function—Resize the function to have two inputs.

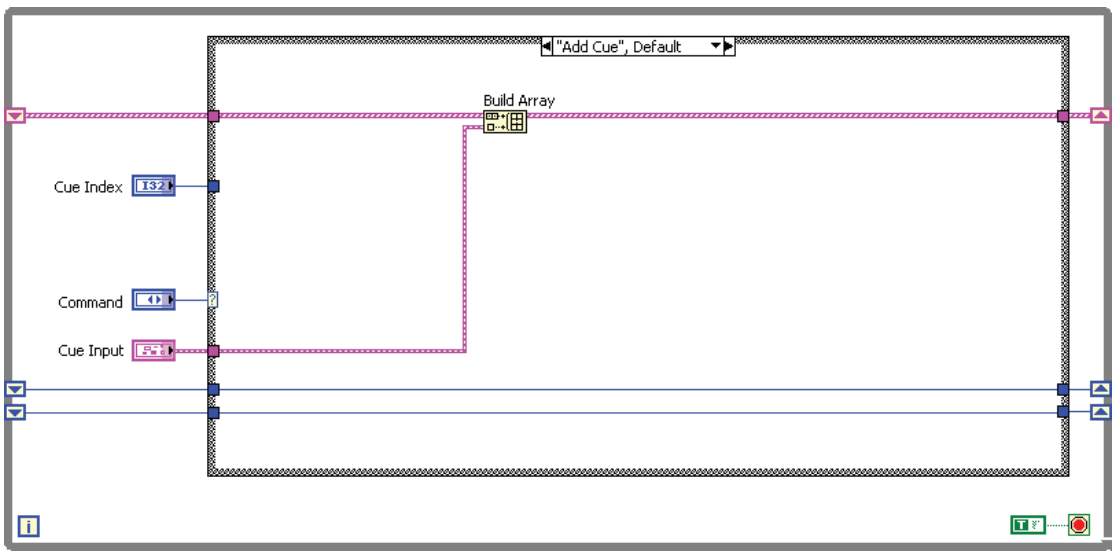


Figure 4-12. Functional Global Variable Add Cue Case

7. Use the Index Array function to complete the Get Cue Values case, as shown in Figure 4-13. This case loads the next cue from the Cue List.

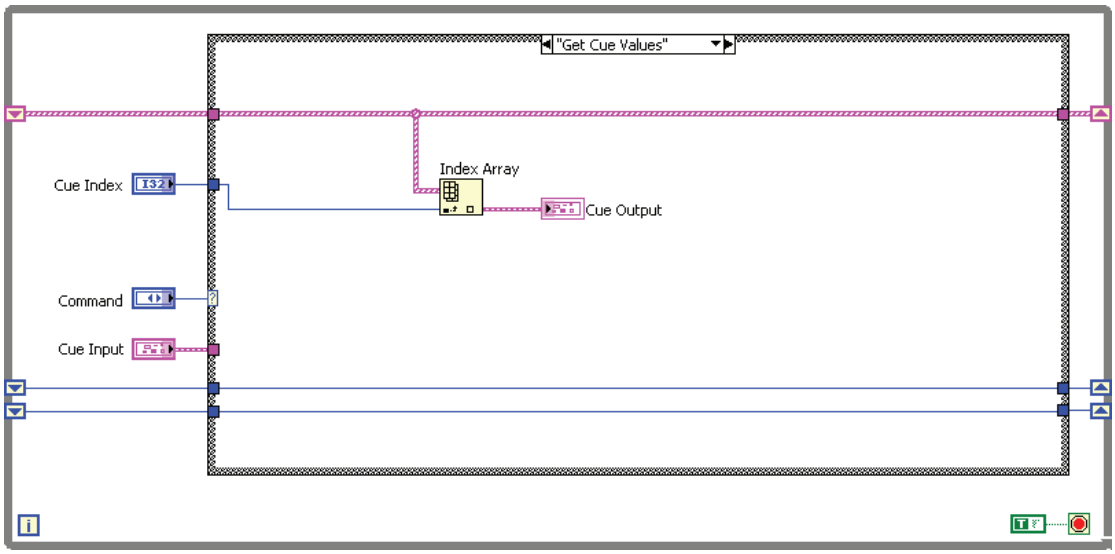


Figure 4-13. Get Cue Values Case

8. Use the Replace Array Subset function to complete the Set Cue Values case as shown in Figure 4-14. This case is modifies an existing cue.

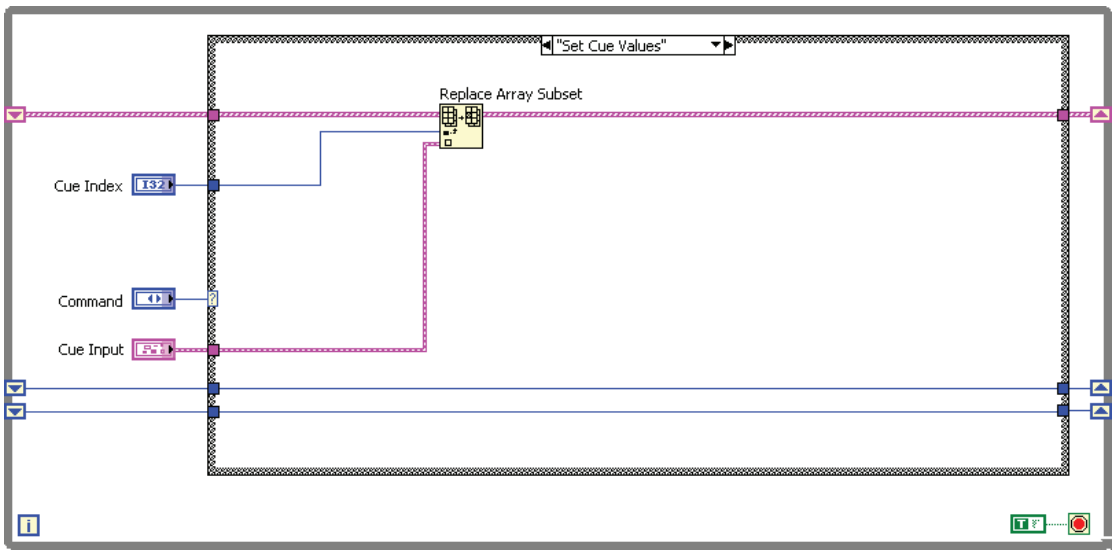


Figure 4-14. Set Cue Values Case

9. Use the Array Size function to complete the Get Number of Cues case, as shown in Figure 4-15. This case retrieves the recorded cues and updates the Cue List.

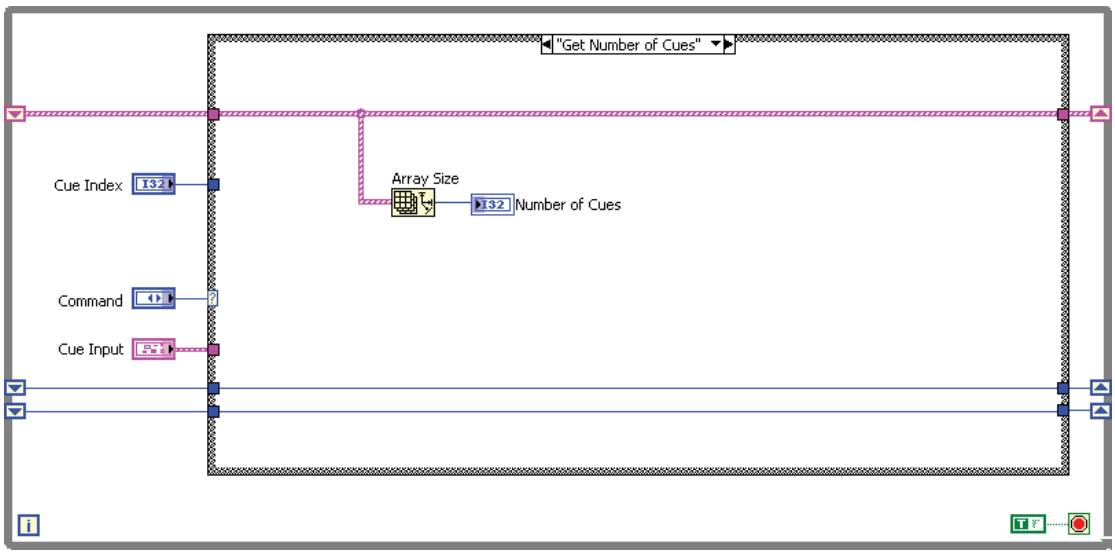


Figure 4-15. Get Number of Cues Case

10. Complete the Get Empty Cue case, as shown in Figure 4-16.

    This case creates a blank cue, for example, to initialize the front panel or record a new cue. A blank cue has an intensity of zero, the color black, and the appropriate channel number.

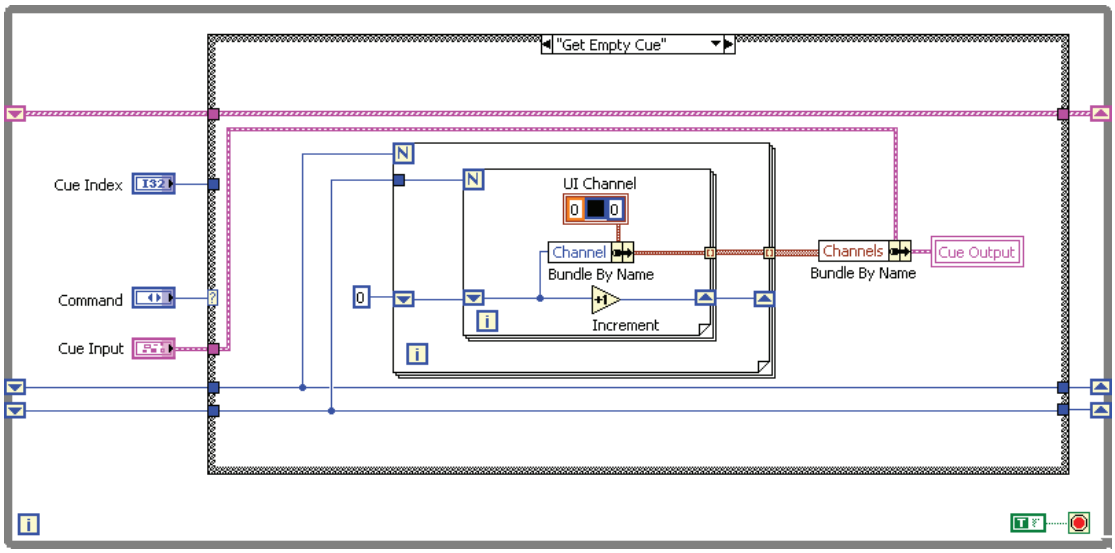


Figure 4-16. Get Empty Cue Case

- ❑ Add two For Loops to the Get Empty Cue case. Connect the Rows data wire to the outer For Loop count terminal. Connect the Columns data wire to the inner For Loop count terminal.

Tip Create more working space for the front panel or block diagram by pressing the <Ctrl> key and using the Positioning tool to drag out a rectangle where you want more space.

- ❑ Add `channel.ctl` constant from the **Controls** virtual folder of the TLC project.

Tip Arrange a cluster horizontally or vertically by right-clicking the cluster border and selecting **Autosizing»Arrange Horizontally** or **Autosizing»Arrange Vertically**. Use this technique to decrease the size of the cluster.

- ❑ I32 numeric constant—This constant ensures that the channel number starts at zero each time this case executes.
- ❑ Two shift registers—The shift registers store the channel numbers as you iterate through the 2D array.
- ❑ Increment function—This function increments the channel number for each iteration of the For Loops.
- ❑ Bundle By Name function—Add this function inside the inner For Loop to create a blank channel from `channel.ctl`, replacing the **Channel** numeric value.
- ❑ Bundle By Name function—Add this function outside the nested For Loops to create a new cue from the **Cue Input** parameter, replacing the **Channels** array value.

  When you wire the **Channel** output of the Bundle By Name function in the inner For Loop to the **Channels** input of the second Bundle By Name function, the tunnels on the For Loops have indexing enabled by default. This creates a 2D array of Channels, where the outer loop determines the number of rows and the inner loop determines the number of columns.
- ❑ Switch to the front panel, right-click the outside border of the Cue Output indicator, and select **Create»Local Variable** to create the local variable for the **Cue Output**.

11. Save the VI.

    This VI provides controlled access to the data stored in the cue. With this type of VI, the data is protected.

Testing

Test the VI to verify its operation.

1. Set the following values for the front panel controls:
   - ❑ **Command** = Initialize
   - ❑ **Rows** = 4
   - ❑ **Columns** = 8
2. Run the VI.
3. Set the following values for the front panel controls:
   - ❑ **Command** = Get Empty Cue
4. Run the VI.
5. Verify that the **Cue Output** contains a 32 element Channel array.
   - ❑ Right-click the Channels indicator and select **Visible Items»Index Display**.
   - ❑ The array should contain 32 elements because you specified 4 rows and 8 columns.

     Verify that there are 4 rows by increasing the Row Index from 0 to 3. Verify that there are 8 rows by increasing the Column Index from 0 to 7. The element in row 3, column 7 should be Channel 31.
   - ❑ Right-click the Channels indicator and select **Visible Items»Index Display** to remove the checkmark and hide the index display.
6. Set the following values for the front panel controls:
   - ❑ **Command** = Add Cue
   - ❑ **Cue Input** = Enter dummy data in the **Wait Time (s)**, **Fade Time (s)**, and **Follow Time (s)** controls.
7. Run the VI.
8. Set the following values for the front panel controls:
   - ❑ **Command** = Get Number of Cues

9. Run the VI.
10. Verify that the **Number of Cues** indicator displays `1`.
11. Set the following values for the front panel controls:
    - ❑ **Command** = Get Cue Values
    - ❑ **Cue Index** = 0
12. Run the VI.
13. Verify that **Cue Output** matches the values you entered in step 6.
14. Save and close the VI and save the project.

End of Exercise 4-5

Exercise 4-6 Design an Error Handling Strategy

Goal

Develop a strategy to handle errors in the application.

Scenario

Creating a custom error code for your application is easy in the LabVIEW environment. Use the custom error code in the application to provide a detailed description of the error that occurred. You can use this information to diagnose the error and the location where the error occurred. Every application that contains multiple states must provide a method and strategy for handling errors.

Design

Use the LabVIEW Error Code File Editor to create the following error code, based on a possible error that can occur in the application.

Error Code	Description
5000	Cue Data Error

Implementation

Part I: Edit the Error Code File

1. Open the TLC project if it is not already open.
2. Edit the Error Code File.
   - ❑ Select **Tools»Advanced»Edit Error Codes** and click **New** to create a new error code file in the **Error Code File Editor**.
   - ❑ Click **Add** to open the **Add Error Code** dialog box. Create the error code as specified in the *Design* section.

Note LabVIEW automatically sets the **New Code** value to `5000` and increments it each time you click **Add**.

   - ❑ Enter the description from the table in the *Design* section in the **New Description** text box. Click **OK** to add the error code.
   - ❑ Select **File»Save As** to save the error code file as `tlc-errors.txt` in the `<labview>\user.lib\errors` directory.
   - ❑ Close the **Error Code File Editor**.

Part II: Modify the Functional Global Variable

1. In Exercise 4-5, you created a functional global variable to control access to the cue data. It is possible to pass an invalid index to the `tlc_Cue Module.vi`. Modify the functional global variable to handle the invalid index and generate error 5000.

   - ❑ Open `tlc_Cue Module.vi` from the TLC project.
   - ❑ Open the block diagram and add a Case structure around the While Loop.
   - ❑ Wire **error in** to the case selector terminal.
   - ❑ Wire the error clusters through all the structures. Right-click each unwired error output tunnel and select **Linked Input Tunnel» Create and Wire Unwired Cases** to automatically wire unwired cases.

2. Modify the Get Cue Values case to determine if the desired cue exists before attempting to obtain its value. Figure 4-17 shows the True and False cases for the Get Cue Values case. To build the Get Cue Values case, use the following items:

   - ❑ Array Size function—This function determines the number of elements in the cue array. The minimum number of cues in the array is zero.
   - ❑ In Range And Coerce function—When you wire the In Range and Coerce limit inputs as shown in Figure 4-17, the **In Range?** output indicates whether the specified **Cue Index** is valid for the existing cue array. If it is not within range, error 5000 generates.
   - ❑ Two numeric constants
   - ❑ Case structure

   - ❑ Error Cluster From Error Code VI—Add this VI to the False case of the Get Cue Values Case structure.

Note The Error Cluster From Error Code VI converts an error or warning code to an error cluster. This VI is useful when you receive a return value from a DLL call or when you return user-defined error codes.

- ❑ True constant—Wire this constant to the **show call chain?** input so that when an error occurs, **source** includes the chain of callers from the VI that produced the error or warning to the top-level VI.

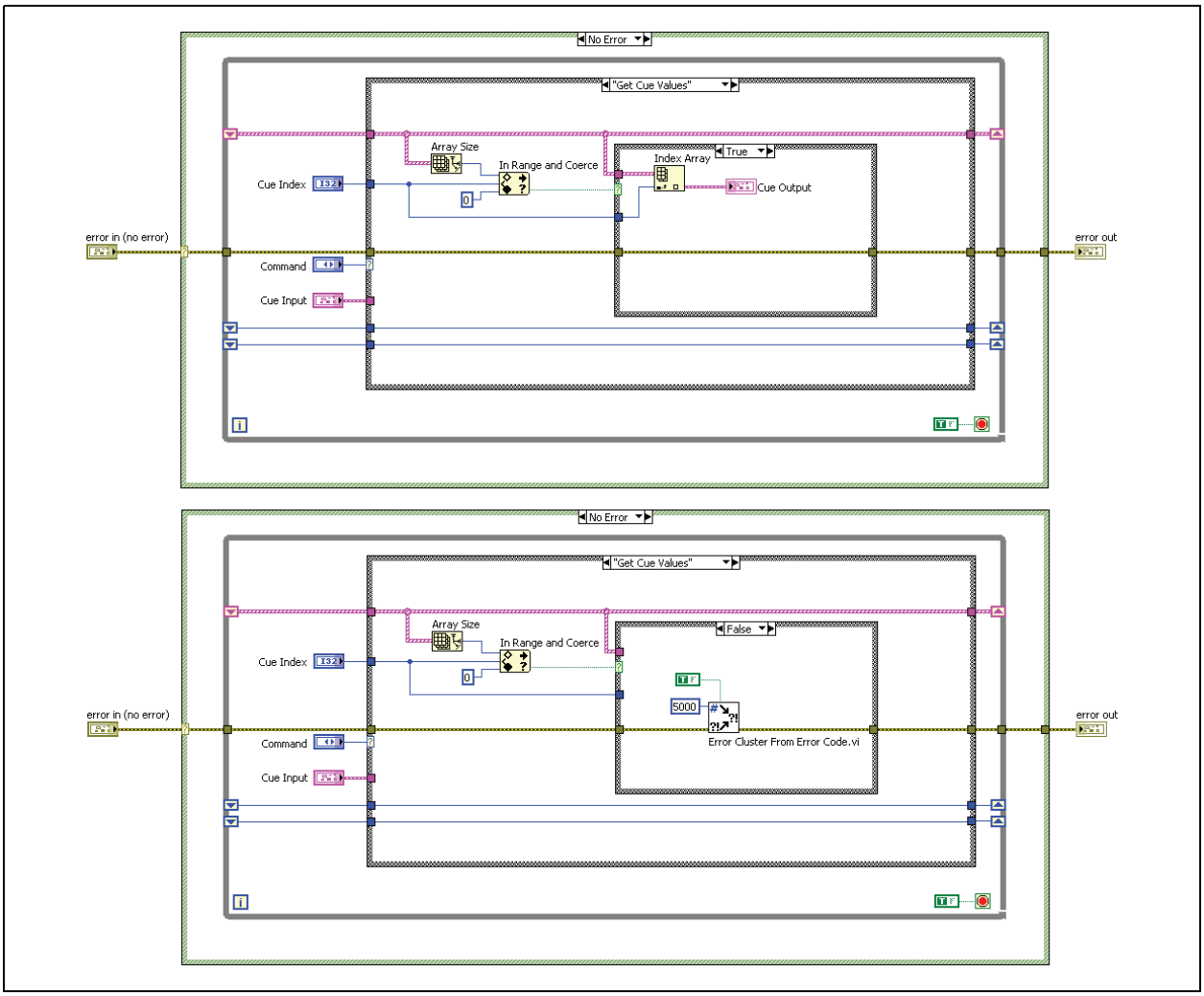


Figure 4-17. Modified Get Cue Values Case

3. Modify the Set Cue Values case to determine if the desired cue exists before attempting to change its value. Figure 4-18 shows the True and False cases for the Set Cue Values case. To build this case, use the same items that you used for the Get Cue Values case in step 2.

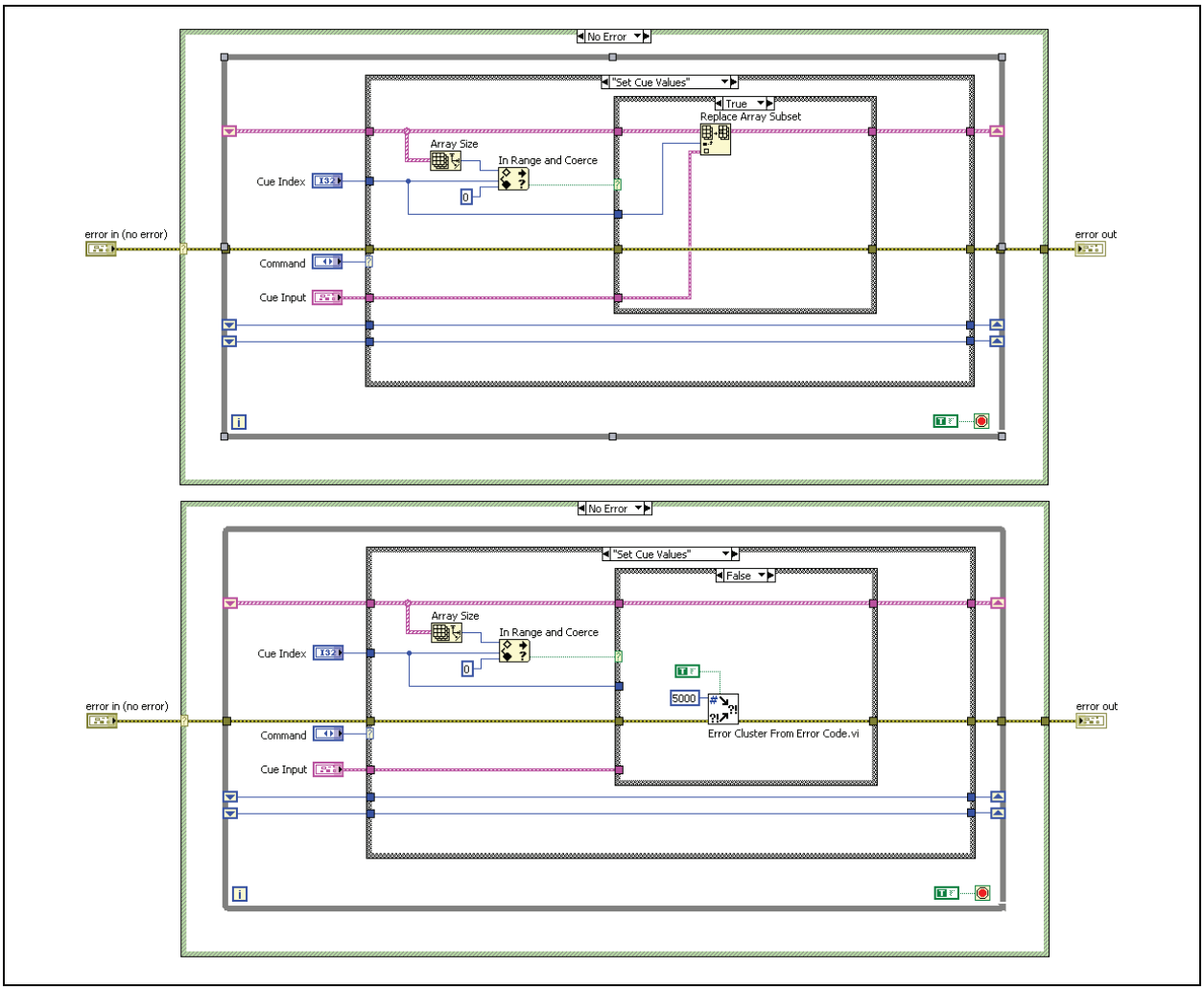


Figure 4-18. Modified Set Cue Values Case

4. Save and close the VI and the project.

Testing

1. Restart LabVIEW to load the error code file. Changes to error code text files take effect the next time you start LabVIEW.
2. Generate an error with the Cue Module VI.
   - ❑ Open the TLC project.
   - ❑ Open `tlc_Cue Module.vi` from the project tree.
   - ❑ Enter an invalid **Cue Index** with **Command** set to **Get Cue Values**.
   - ❑ Run the VI.
3. Verify that the error explanation matches what you specified when you created the error code file.
   - ❑ Verify that **error out** indicates an error occurred.
   - ❑ Right-click the error cluster and select **Explain Error** from the shortcut menu.
   - ❑ Verify that the **Explanation** text box displays the custom error description in the **Possible reason(s):** section.
4. Verify that the same error generates if you enter an invalid **Cue Index** with **Command** set to **Set Cue Values**.
5. Close the VI.

End of Exercise 4-6

Notes

5

Implementing the User Interface

Exercise 5-1 Implement User Interface-Based Data Types

Goal

Implement user interface-based data types.

Scenario

Implement the user interface for the application. The specification from the customer and the requirements document define the user interface for the project.

Design

Figure 5-1 shows the user interface from the requirements document.

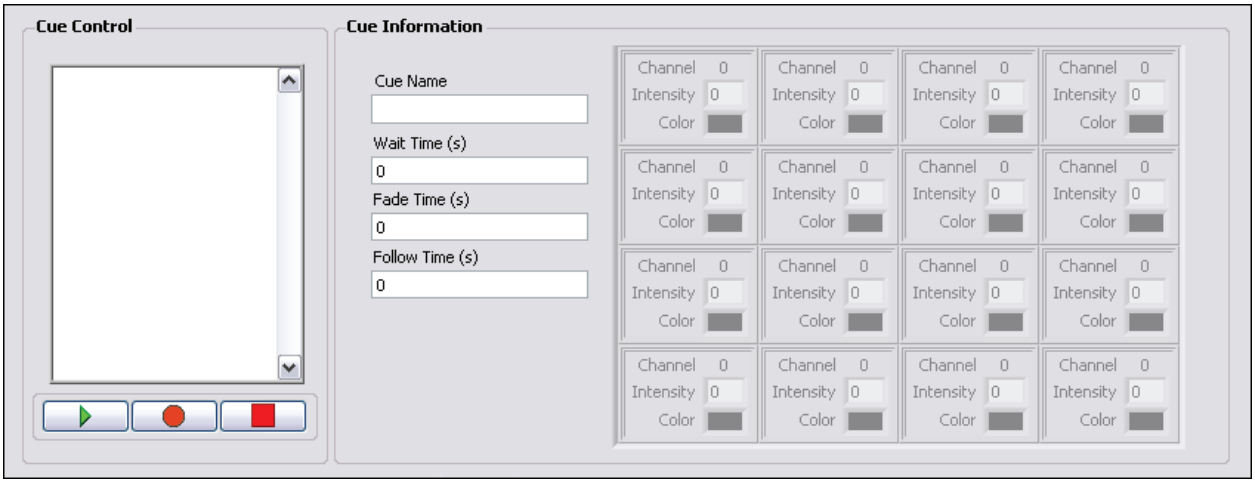


Figure 5-1. Theatre Light Control User Interface

The user interface includes the following inputs and outputs.

Inputs

- **Play** button—Input by the user
- **Record** button—Input by the user
- **Stop** button—Input by the user

Outputs

- **Cue List** listbox—Displays a list of all of the recorded cues
- **Cue Name** string—Displays the name of the currently playing cue
- **Wait Time (s)** numeric—Displays the wait time for the currently playing cue
- **Fade Time (s)** numeric—Displays the fade time for the currently playing cue
- **Follow Time (s)** numeric—Displays the follow time for the currently playing cue
- **Channel** cluster—Displays the channel number, channel intensity, and channel color for each channel

Implementation

Create a front panel similar to the user interface shown in Figure 5-1.

1. Open the TLC project if it is not already open.
2. Create a new VI that uses the producer/consumer (events) design pattern.
   - ❑ Select **File»New** from the **Project Explorer** window to open the **New** dialog box.
   - ❑ In the **New** dialog box, select **VI»From Template»Frameworks» Design Patterns»Producer/Consumer Design Pattern(Events)** and make sure a checkmark appears in the **Add to Project** checkbox.
   - ❑ Click **OK** to open the design pattern.
   - ❑ Select **File»VI Properties** and select **Window Appearance** from the **Category** pull-down menu.
   - ❑ Add a checkmark to **Same as VI name** to use the VI name for the title of the VI window.
   - ❑ Save the VI as `TLC Main.vi` in the `<Exercises>\LabVIEW Core 3\Course Project` directory. LabVIEW automatically adds the file to the project.
3. Delete the **Queue Event** and **STOP** buttons from the front panel.
4. Create the Cue List indicator.
   - ❑ Add a System Listbox from the **System** palette to the front panel.

- ❑ Change the label of the Listbox to `Cue List`.
- ❑ Right-click the Listbox and select **Change to Indicator**.
- ❑ Right-click the Listbox. Select **Visible Items** and deselect **Label** to hide `Cue List`.

5. Customize a **System** button with a decal to create the Play button.
   - ❑ Add a **System** button to the front panel.

Tip Press the <Ctrl-space> keys to display the **Quick Drop** dialog box. Use this dialog box to specify a palette object or project item by name and then place the object on the block diagram or front panel. Press the <Enter> key, double-click the name of the object in the search results text box, or click the block diagram or front panel to attach the object to the cursor. Then click the location on the block diagram or front panel where you want to add the object.

   - ❑ Right-click the button and select **Advanced»Customize** from the shortcut menu to open the Control Editor.
   - ❑ Select **Edit»Import Picture to Clipboard** and select `play.gif` from the `<Exercises>\LabVIEW Core 3\Course Project\Shared\Images` directory to add the image to the clipboard.
   - ❑ Right-click the button in the Control Editor and select **Import Picture from Clipboard»Decal** from the shortcut menu to add the decal to the button.
   - ❑ Right-click the button and deselect **Visible Items»Boolean Text** from the shortcut menu to hide the button text.
   - ❑ Change the label of the control to `Play`.
   - ❑ Save the control as `Play Button.ctl` in the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory.
   - ❑ Close the Control Editor. When prompted, click **Yes** to replace the original control with the custom control.
   - ❑ Select `Play Button.ctl` in the project tree and drag the file to the **Controls** virtual folder to add the control in the project hierarchy.

6. Add `Record Button.ctl` and `Stop Button.ctl` to the project and the front panel.
   - ❑ Right-click the **Controls** virtual folder, select **Add»File** from the shortcut menu and navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory.

- ❑ Select `Record Button.ctl` and `Stop Button.ctl` and click **Add File** to add the files to the **Controls** virtual folder.

Tip <Ctrl>-click to select multiple files.

- ❑ Click and drag the custom controls from the project tree to the front panel.
- ❑ Arrange the controls.
- ❑ Hide the control labels.

7. Add the typedef that contains the Cue Name, Wait Time, Fade Time, Follow Time, and array of Channels to the front panel.
   - ❑ Drag `tlc_Cue_Information.ctl` from the project tree to the front panel.
   - ❑ Click and drag the border of `tlc_Cue_Information.ctl` to resize the cluster to match the specification in Figure 5-1. Click and drag the corner of the array to display a 4×4 array of channels.
   - ❑ Hide the label for the array.
   - ❑ Right-click the cluster border and select **Change to Indicator** from the shortcut menu.
8. Add decorations to the front panel to visibly group objects as shown in Figure 5-1.
   - ❑ Use the System Recessed Frame decoration to create a professional-looking user interface.
   - ❑ Double-click an empty space on the front panel and enter `Cue Control` to create a free label to place above the Listbox.
9. Add **error in** and **error out** clusters to the front panel to pass error data through the VI.
10. Resize the window to hide the error clusters.
11. Save the VI.

Note The **Run** button is broken because you deleted the **Queue Event** and **Stop** buttons. You resolve the broken **Run** button in a later exercise.

End of Exercise 5-1

Exercise 5-2 Implement a Meaningful Icon

Goal

Implement a meaningful icon for the VI.

Scenario

Follow the suggestions for creating an icon to develop an icon that describes the purpose of TLC Main VI.

Design

Create an icon for the TLC Main VI that resembles the icon shown at left.

Implementation

1. Add shared files and the **Shared** virtual folder to the project.
   - ❑ Right-click **My Computer** in the project tree and select **Add» Folder (Snapshot)** from the shortcut menu.
   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Shared` directory and click **Current Folder** to add the folder and its contents to the project tree.
2. Open the front panel of TLC Main VI.
3. Import a graphic to use in the icon.
   - ❑ Select **Edit»Import Picture to Clipboard**.
   - ❑ Navigate to `light bulb.bmp` in `<Exercises>\LabVIEW Core 3\Course Project\Shared\Images` and click **OK** to import the graphic to the clipboard.
4. Right-click the VI icon in the upper right corner of the front panel and select **Edit Icon** from the shortcut menu to open the Icon Editor.
5. Delete the default icon.

Tip Press <Ctrl-A> to select all user layers of the icon.

6. From the **Templates** tab, select **All Templates** and click the template shown at left.

7. Use the **Fill** tool to apply a different color to the top region of the icon.

Tip Use a light color to help keep the icon text readable.

8. Use the **Text** tool to add text to the top section of the icon.

Tip Double-click the **Text** tool to modify the font.

9. Add the light bulb to the bottom section of the icon.
   - ❑ Select **Edit»Paste** to add the graphic to the icon.
10. Click **OK** to close the Icon Editor. Save the VI.

End of Exercise 5-2

Exercise 5-3 Implement an Appropriate Connector Pane

Goal

Implement an appropriate connector pane for the VI.

Scenario

Build every VI with the $4 \times 2 \times 2 \times 4$ connector pane. This connector pane pattern provides for scalability, maintainability, and readability. The $4 \times 2 \times 2 \times 4$ connector pane is very easy to wire.

Design

Modify the TLC Main VI by following the connector pane guidelines in this lesson to create a $4 \times 2 \times 2 \times 4$ connector pane, as shown at left. Connect the **error in** and **error out** clusters to the connector pane terminals.

Implementation

1. Open the front panel of TLC Main VI.
2. Right-click the VI icon in the upper right corner of the front panel and select **Show Connector** from the shortcut menu.
3. Verify that the connector pane uses the $4 \times 2 \times 2 \times 4$ pattern.
4. Use the Wiring tool to connect the **error in** and **error out** clusters to the connector pane.
   - ❑ Click the connector pane terminal you want to connect to the **error in** cluster. Notice that your pointer becomes the wiring tool.
   - ❑ Click the **error in** cluster to assign the control to the connector pane terminal.
   - ❑ Click the connector pane terminal you want to connect to the **error out** cluster.
   - ❑ Click the **error out** cluster to assign the indicator to the connector pane terminal.
5. Right-click the connector pane and select **Show Icon** from the shortcut menu to display the icon for the VI.
6. Save and close the VI.

End of Exercise 5-3

Notes

6

Implementing Code

Exercise 6-1 Implement the Design Pattern

Goal

Use LabVIEW to implement a design pattern as the basis for the application architecture.

Scenario

Using a design pattern for the architecture makes the application readable, scalable, and maintainable. Implementing the producer/consumer (events) design pattern makes the user interface more responsive. Using a variant data type makes the architecture scalable for future needs.

Design

Modify the design pattern as follows to create an architecture that meets the project requirements.

- Create a type definition for the functions that the application performs.
- Use the type definition as the data type to pass data from the producer to the consumer.
- Initialize the design pattern.
- In the consumer loop of the design pattern, verify that a case exists to process each function in the enumerated data type.
- Enqueue an element into the producer/consumer with (events) queue when the producer receives an event.
- Create a custom run-time menu to perform the Load, Save, and Exit functions.
- Add a case to the Event structure in the producer loop to respond to menu selections.
- Create user events that allow the consumer loop to send error data to the producer loop to stop the producer

Implementation

Implement the architecture for the Producer/Consumer (Events) design pattern, as shown in Figure 6-1.

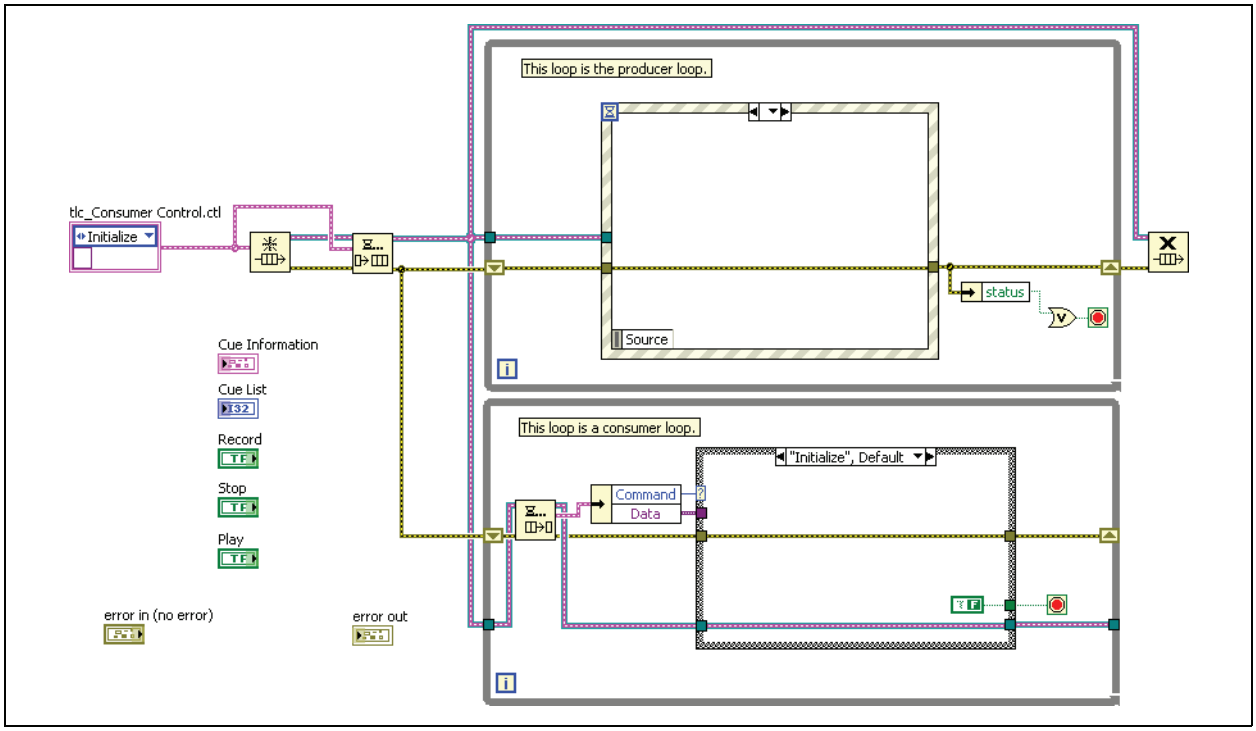


Figure 6-1. Theatre Light Controller Architecture

1. Open the TLC project.
2. Add `tlc_Functions.ctl` to the project. This type definition is an enum which contains the functions that consumer loop of the application will perform. `tlc_functions.ctl` includes the following functions:
   - Initialize
   - Record
   - Load
   - Save
   - Play
   - Stop
   - Exit

   ❑ Right-click **Controls** in the project tree, select **Add»File** from the shortcut menu and navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory

   ❑ Select `tlc_functions.ctl` and click **Add File** to add the control to the **Controls** virtual folder.

3. Create a scalable data type to pass data from the producer loop to the consumer loop. This data type is a cluster that includes the `tlc_Functions.ctl` control and a variant.

   - ❑ Select **File»New** to open the **New** dialog box.
   - ❑ Select **Other Files»Custom Control** from the **Create New** tree.
   - ❑ Verify **Add to project** is selected.
   - ❑ Click **OK** to open the Control Editor.
   - ❑ Add a cluster constant to the front panel of the Control Editor.
   - ❑ Change the label of the cluster to `tlc_Consumer Control.ctl`, to correspond to the name of the file. Using the filename of the control as a label helps you keep track of the block diagram controls.
   - ❑ Drag the `tlc_Functions.ctl` type definition from the **Project Explorer** window to the cluster.
   - ❑ Add a Variant control to the cluster.
   - ❑ Change the label of the variant to `Data`. Figure 6-2 shows the resulting cluster.

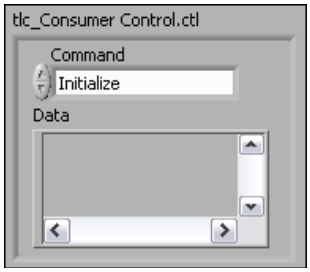


Figure 6-2. Variant and Type Definition Enumerated Control Cluster

   - ❑ Select **Type Def.** from the **Control Type** pull-down menu.
   - ❑ Save the type definition as `tlc_Consumer Control.ctl` in the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory.
   - ❑ Close the Control Editor.
   - ❑ Move the `tlc_Consumer Control.ctl` control to the **Controls** virtual folder in the LabVIEW Project.

4. Open the TLC Main VI.

5. Add `tlc_Consumer Control.ctl` to the block diagram.

   - ❑ Drag the control from the Project Window to the block diagram to add it as a constant.
   - ❑ Position the constant outside the producer and consumer loops.
   - ❑ Set the value of the enumerated type constant within the constant to **Initialize**.

6. Initialize the producer/consumer design pattern.

   - ❑ Delete the empty string constant wired to the Obtain Queue function.
   - ❑ Wire the tlc_Consumer Control constant to the **element data type** input of the Obtain Queue function.
   - ❑ Add the Enqueue Element function to the block diagram. Wire the cluster constant to the Enqueue Element function to initialize the design pattern.

7. In the consumer loop of the design pattern, ensure a case exists to process each function in the enumerated data type. The resulting block diagram should be similar to Figure 6-1.

   - ❑ Add the Unbundle by Name function outside the Case structure in the consumer loop.
   - ❑ Wire the **element** output of the Dequeue Element function to the input of the Unbundle by Name function.
   - ❑ Delete the error cluster wires connected to the case selector terminal and the loop conditional terminal for the consumer loop.

Note You are deleting the initial error handling wiring in preparation for more complex structures in later exercises. In Exercise 6-4 you build a functional global variable to handle errors in the application and in Exercise 7-5, *Integrate Error Module*, you integrate the error module in the main application.

   - ❑ Wire the **Command** element of the Unbundle by Name function to the case selector terminal of the Case structure.
   - ❑ Right-click the border of the Case structure and select **Add Case For Every Value** from the shortcut menu to populate the Case structure with the items in the enumerated type control.

- ❑ Wire a False constant to the loop condition terminal of the While Loop in each case of the Case structure. Change the constant in the Exit case to True to enable the Exit case to stop the consumer loop.
- ❑ Right-click the error cluster tunnel on the consumer loop and select **Replace with Shift Register** from the shortcut menu. Add the other side of the shift register to the right side of the consumer loop.
- ❑ Wire the queue reference and error cluster through each case of the consumer loop.

8. Modify an event case in the producer loop to respond to the Value Change event for the **Play** button.
   - ❑ Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu to open the **Edit Events** dialog box.
   - ❑ Select **Play** from the **Event Sources** list and select **Value Change** from the **Events** list.
   - ❑ Click **OK**.
9. Modify the Play event case to send a message to the consumer loop to execute Play, as shown in Figure 6-3. Delete the Unbundle By Name and Or functions that are connected to the loop conditional terminal. Add the following items:
   - ❑ `tlc_Consumer Control.ctl`
   - ❑ Bundle By Name function
   - ❑ `tlc_functions.ctl` (set to **Play**)

Tip Right-click the **Command** element of the Bundle By Name function and select **Create»Constant** from the shortcut menu to create `tlc_functions.ctl`.

- ❑ Enqueue Element function
- ❑ False constant

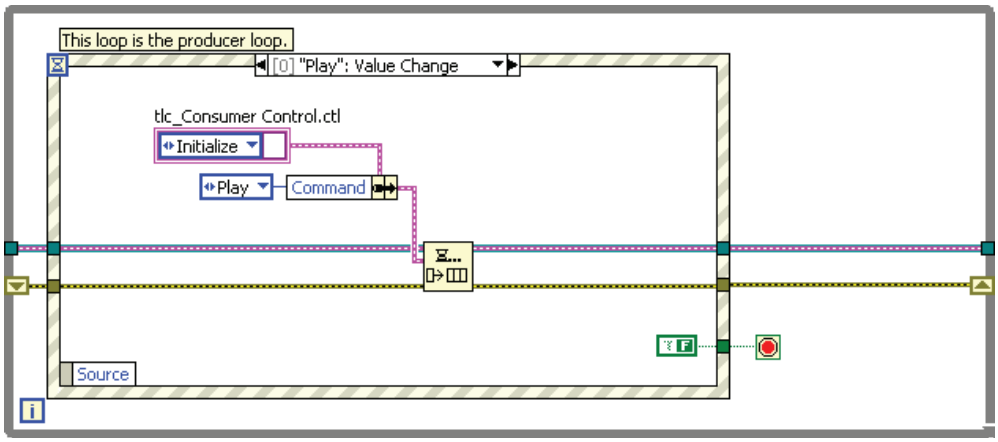


Figure 6-3. Producer Play Event

10. Create a Value Change event case for the following controls: **Record** and **Stop**.

    ❑ Right-click the Play event case, and select **Duplicate Event Case** from the shortcut menu.

Tip If you duplicate an event case with the terminal in the event case, you also duplicate the terminal. Placing a terminal in the corresponding event case is good programming style because it ensures that LabVIEW reads the terminal when the event occurs.

    ❑ Select **Record** and the **Value Change** event and click **OK**.

    ❑ Modify the enumerated type constant to send a Record command to the consumer loop.

    ❑ Repeat the previous step for the Stop control, changing the command the producer sends to the consumer to correspond to the appropriate function.

Table 6-1 shows the appropriate enumerated type control item to add in the queue when each control receives a value change event.

Table 6-1. Control Items

Control	Enum Item
Record button	`Record`
Play button	`Play`
Stop button	`Stop`

11. Move the **Record**, **Play** and **Stop** control terminals to the corresponding event cases to ensure that each control is read when it generates an event.

12. Create a custom run-time menu to perform the Load, Save, and Exit functions. Figure 6-4 shows the completed menu.

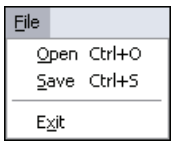


Figure 6-4. LabVIEW Run-Time Menu

- ❑ Right-click **My Computer** in the project tree, select **New»Virtual Folder**, and create the `Menu` virtual folder in the project tree.
- ❑ In the TLC Main VI, select **Edit»Run-Time Menu** to display the **Menu Editor** dialog box.
- ❑ Select **File»New** to create a new run-time menu.
- ❑ Enter `_File` in the **Item Name** textbox.

Tip In the **Item Name** textbox, enter an underscore (_) before the letter you want to associate with the <Alt> key for that menu. This creates a shortcut so the user can press the <Alt> key and the associated key to access the menu.

- ❑ Change the text in the **Item Tag** textbox to `File`.
- ❑ Place your cursor in **Shortcut** and press the <Ctrl-F> keys. LabVIEW records the shortcut key combination for you.
- ❑ Click the blue **+** button on the Menu Editor toolbar to add a new item under the **File** item.
- ❑ Click the right arrow button on the toolbar to make the new item a subitem of the **File** menu.
- ❑ Enter `_Open...` in the **Item Name** textbox to create a menu item for Open.
- ❑ Change the **Item Tag** to `Open`.
- ❑ Place your cursor in **Shortcut** and press <Ctrl-O> to record the keyboard shortcut.

Note The Item Tag is passed to LabVIEW so that you can create decision making code to respond to the selected menu item.

- ❑ Click the blue **+** button on the Menu Editor toolbar to add a new item under the **File** item.
- ❑ Enter `_Save...` in the **Item Name** textbox to create a menu item for Save.
- ❑ Change the **Item Tag** to `Save`.
- ❑ Place your cursor in **Shortcut** and press <Ctrl-S> to record the keyboard shortcut.
- ❑ Click the blue **+** button on the Menu Editor toolbar to add a new item under the **File»Save** item.
- ❑ Create a menu separator by selecting **Separator** from the **Item Type** drop-down menu.
- ❑ Click the blue **+** button on the Menu Editor toolbar to add a new item under the **File** item.
- ❑ Enter `E_xit` in the **Item Name** textbox to create a menu item for Exit.
- ❑ Change **Item Tag** to `Exit`.
- ❑ Select **File** in the **Preview** section to preview the menu and verify that it matches Figure 6-4.
- ❑ Save the run-time menu as `tlc_Menu.rtm` in the `<Exercises>\LabVIEW Core 3\Course Project\Menu` directory.
- ❑ Close the Menu Editor. When prompted, click **Yes** to change the run-time menu to the custom menu.

13. Add a case to the Event structure in the producer loop to respond to menu selections.
    - ❑ Delete any unused cases in the Event structure.
    - ❑ Right-click the Event structure border and select **Add Event Case** from the shortcut menu to open the **Edit Events** dialog box.
    - ❑ Select **<This VI>** from the **Event Sources** list and **Menu Selection (User)** from the **Events** list to create the Menu Selection (User) event case. Click **OK**.

- ❑ Add a Case structure to the Menu Selection (User) event case. Wire the ItemTag event data node to the case selector terminal.
- ❑ Modify the Case structure in the Menu Selection (User) event case to have four cases. A `Default` case and a case for each ItemTag string—`"Open"`, `"Save"`, and `"Exit"`. Be sure the spelling for the case selector matches the spelling of the Item Tags in the Menu Editor.
- ❑ Delete any unused cases in the Case structure.
- ❑ Modify the Exit case to stop the VI when the user selects **File»Exit**, as shown in Figure 6-5. Use the same items that you used for step 9.

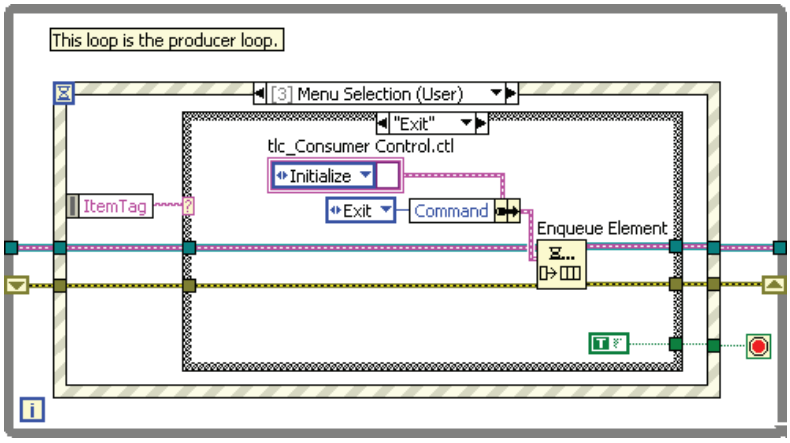


Figure 6-5. Menu Selection Event Case

- ❑ Wire a True constant to the conditional terminal in the Exit case and a False constant in every other case. Set the Command element to Exit. This causes the Exit case to stop the producer loop and the consumer loop.
- ❑ Right-click the output tunnels and select **Linked Input Tunnel» Create and Wire Unwired Cases**, then select the corresponding input tunnels to wire the queue reference and error cluster wires through the remaining cases in the Case structure. You build the remaining cases in later exercises.

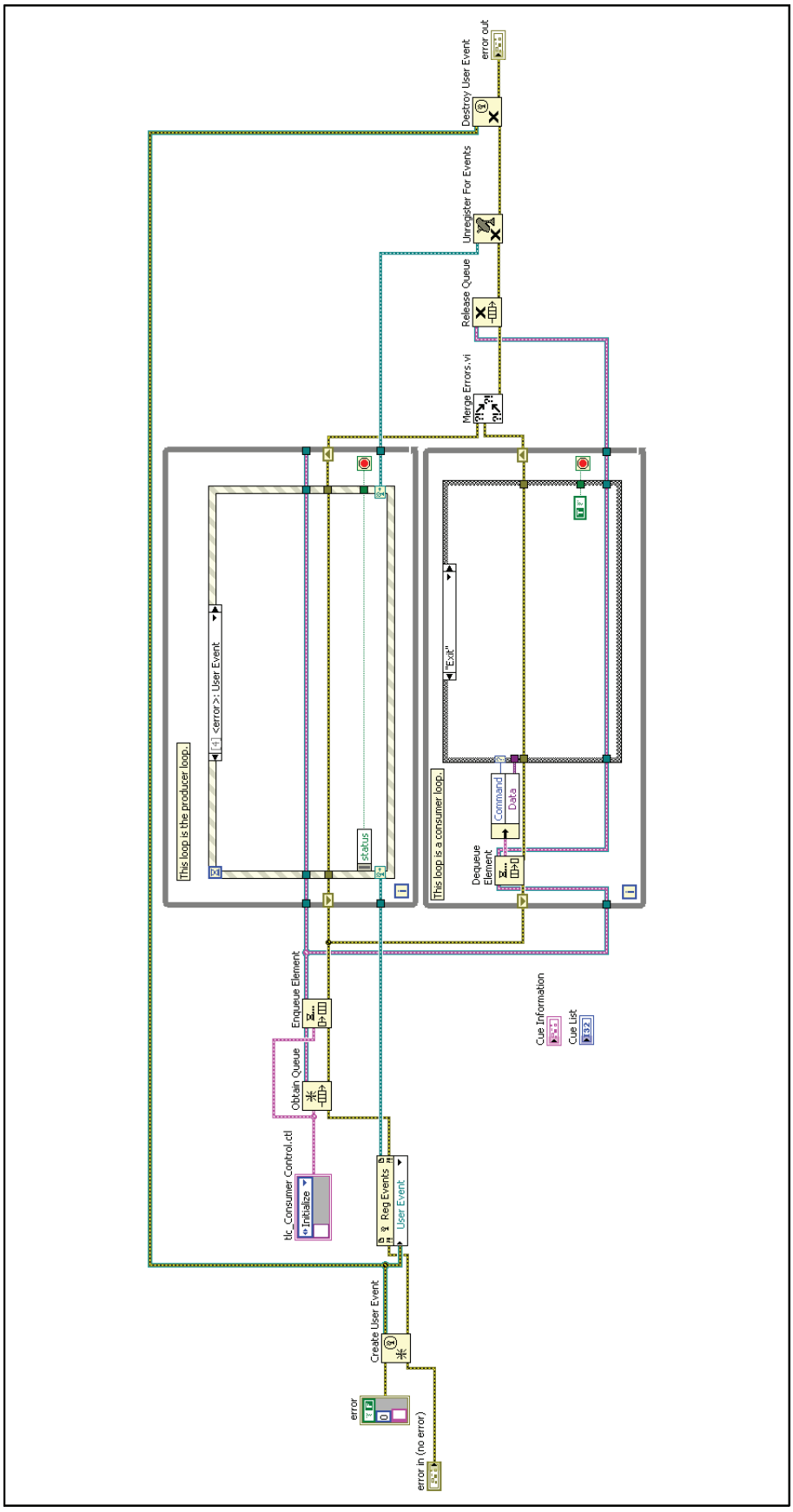


Figure 6-6. Theatre Light Controller Architecture with User Events

14. Complete the block diagram as shown in Figure 6-6 to create user events that allow the consumer loop to send error data to the producer loop to stop the producer.

    ❑ Right-click the border of the Event structure in the producer loop and select **Show Dynamic Event Terminals** from the shortcut menu.

    ❑ Add the Create User Event function to the block diagram. LabVIEW uses the **user event data type** you wire to determine the event name and data type of the event.

    ❑ Right-click the **error in** input of the Create User Event function and select **Create»Constant** to create the error constant. Label the error constant `error` and wire it to **user event data type**.

    ❑ Add the Register for Events node to the block diagram. This function dynamically registers the user event. Wire the event registration refnum output of the Register for Events node to the dynamic terminal of the Event structure.

    ❑ Wire the Create User Event function to the Register for Events node.

    ❑ Add the Dynamic Event case to the Event structure.

        – Right-click the Event structure border and select **Add Event Case** from the shortcut menu.

        – Select **Dynamic»<error>: User Event** from the **Event Sources** list and click **OK**.

        – Wire the queue reference and error cluster through each case of the Event structure.

Note The name of the dynamic event is the same as the owned label for the data structure wired to the Create User Event function.

    ❑ Wire the **status** output from the event data node to the loop conditional terminal.

    ❑ Add the Merge Errors VI to the block diagram. This VI combines multiple error cluster inputs into a single error cluster output.

    ❑ Add the Unregister for Events function to the block diagram. This function unregisters the dynamic event.

❑ Add the Destroy User Event function to the block diagram. This function destroys the reference to the user event.

❑ Wire the Release Queue, Unregister for Events, and Destroy User Events functions and the Merge Error VI as shown in Figure 6-6.

15. Delete any unused Event structure cases from the design pattern.

16. Connect the **error in** and **error out** clusters to the design pattern as shown in Figure 6-6.

17. Save the VI.

Testing

1. Add the Format Into String and One Button Dialog functions to each case of the consumer loop.

   ❑ Wire the **Command** enum to **input 1** of the Format Into String function.

   ❑ Wire the **resulting string** output to the **message** input of the One Button Dialog function to open a dialog box indicating that the case executes when the front panel receives events.

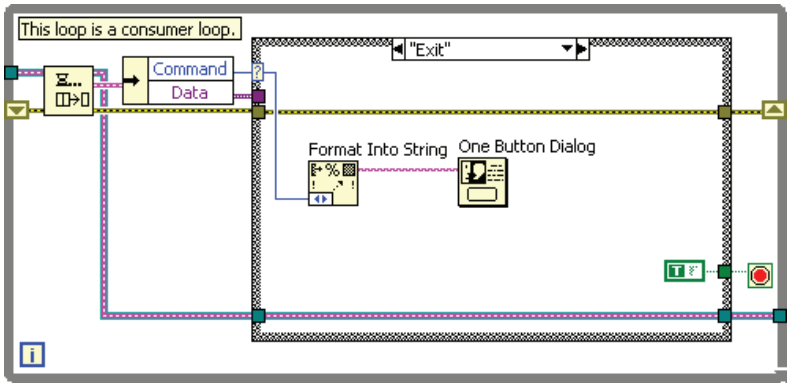


Figure 6-7. Convert Enumerated Type Control to String

2. Save the VI.

3. Run the VI to verify that all the functions listed in step 2 of the *Implementation* section work correctly.

   You can test the functionality by clicking a front panel button to execute the Event structure. When the Event structure executes, it places a message in the queue to execute the consumer.

The only cases that are not functional at this time are Load and Save. You implement this functionality in a later exercise.

4. Verify that you can exit the application from the run-time menu.
5. Close the TLC Main VI.
6. Save the project.

End of Exercise 6-1

Exercise 6-2 Timing

Goal

Create a functional global variable that uses the Get Date/Time In Seconds function to provide accurate software controlled timing.

Scenario

The Theatre Light Controller requires accurate timing to control the cue wait time, fade time, and follow time. The Theatre Light Controller must respond within 100 ms when any operation is running. The timing method you choose for the Theatre Light Controller cannot interfere with the application response. You will develop a functional global variable that will be used in a later exercise to implement software control timing.

Design

Build a functional global variable that you can use to control the timing of the Theatre Light Controller. You will use the functional global variable to control the timing for the wait, fade, and follow times of the Theatre Light Controller. Use the Get Date/Time In Seconds function to control the timing.

This application requires a structure that has precise timing characteristics and does not use the processor. However, the structure also must respond within 100 ms to meet the application requirements. Therefore, you will use the Get Date/Time In Seconds function in a functional global variable to implement software control timing.

A functional global variable provides for a good architecture to modularize the timing functionality. The functional global variable has the following functions:

- `Start`—Starts a new time from zero, which resets the elapsed time.
- `Check Time`—Checks if the target time has elapsed.

Implementation

1. Add the timing module files and **Timing** virtual folder to the **Modules** virtual folder.

   ❑ Right-click **Modules** in the project tree and select **Add»Folder (Snapshot)** from the shortcut menu.

- ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Modules\Timing` directory and click **Current Folder** to add the folder and its contents to the project tree. LabVIEW adds the following files to the project.
  - `tlc_Timing Command Control.ctl`
  - `tlc_Timing Module Unit Test States.ctl`
  - `tlc_Timing Module Unit Test.vi`
  - `tlc_Timing Module.vi`

2. Open the `tlc_Timing Module.vi`.
3. Modify the No Error case to make the tlc_Timing Module VI a functional global variable that keeps track of elapsed time, as shown in Figure 6-8.

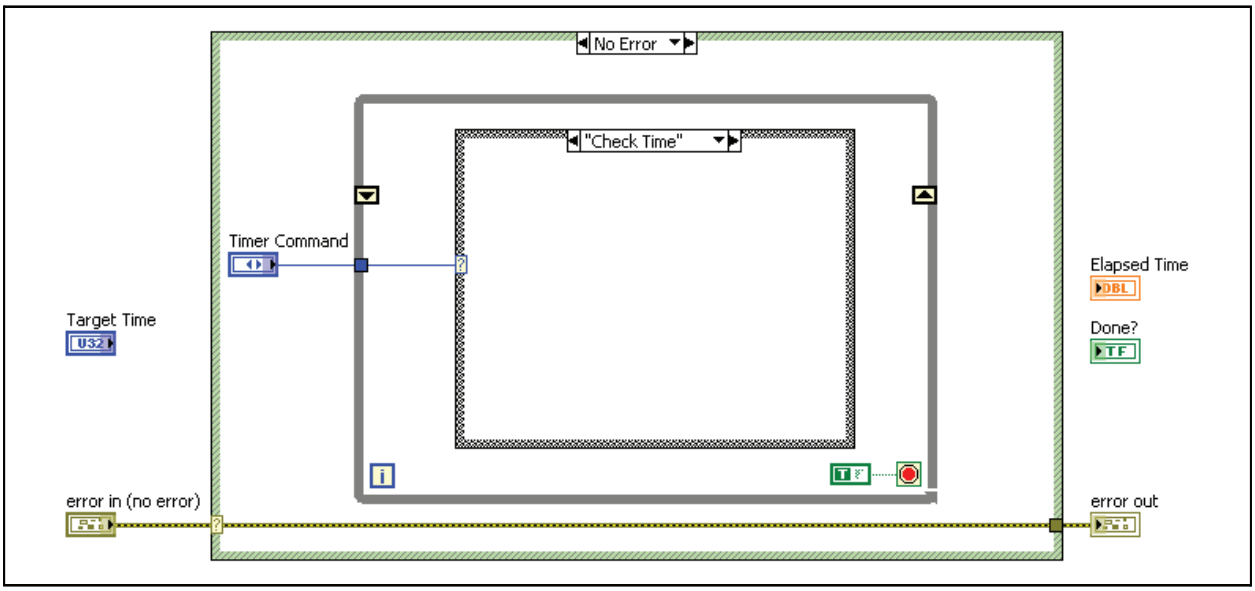


Figure 6-8. Timing Module Block Diagram

- ❑ Add a While Loop in the No Error case of the Case structure.
- ❑ Right-click the border of the While Loop and select **Add Shift Register**.
- ❑ Right-click the conditional terminal of the While Loop and select **Create»Constant**. Set the constant to **True**.
- ❑ Add a Case structure inside the While Loop.

- ❑ Move the Timer Command control to the No Error case and wire it to the case selector input of the Case structure inside the While Loop. The Case structure should now have a Check Time case and a Start case.

4. Modify the Start case to store the start time of the VI, as shown in Figure 6-9. Use the following items:

   - ❑ Get Date/Time In Seconds function
   - ❑ To Double Precision Float function
   - ❑ Numeric constant—Right-click and select **Representation»DBL**
   - ❑ False constant

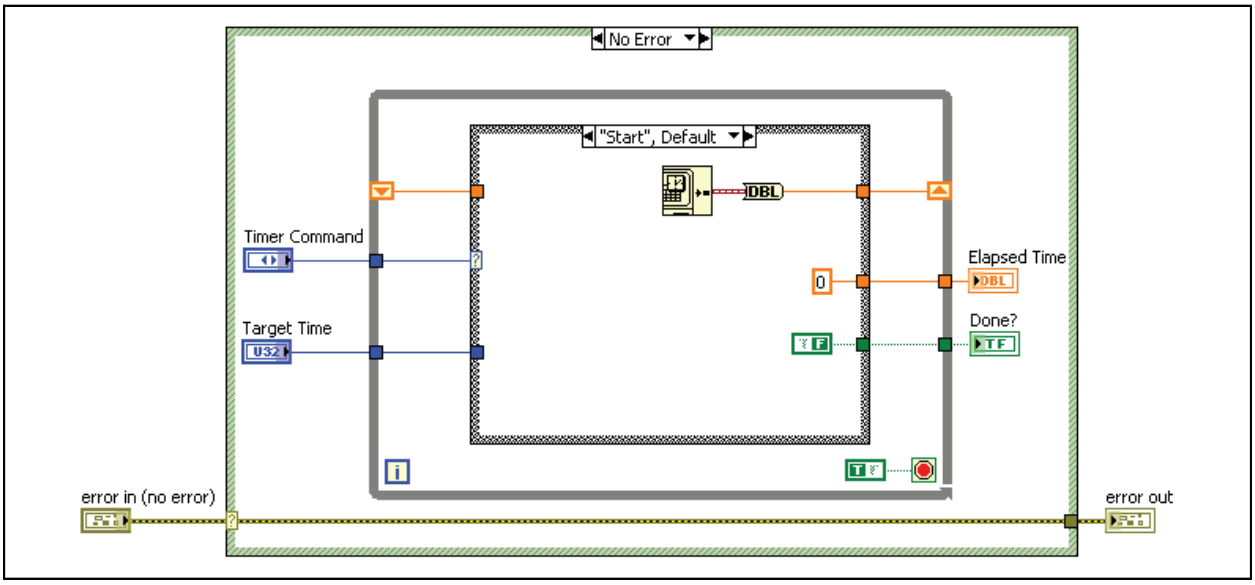


Figure 6-9. Timing Module Start Case

5. Modify the Check Time case to output the elapsed time after a Start command has been sent to the tlc_Timing Module VI, as shown in Figure 6-10. Use the following items:

   - ❑ Get Date/Time In Seconds function
   - ❑ Two To Double Precision Float functions
   - ❑ Subtract function
   - ❑ Greater Or Equal? function

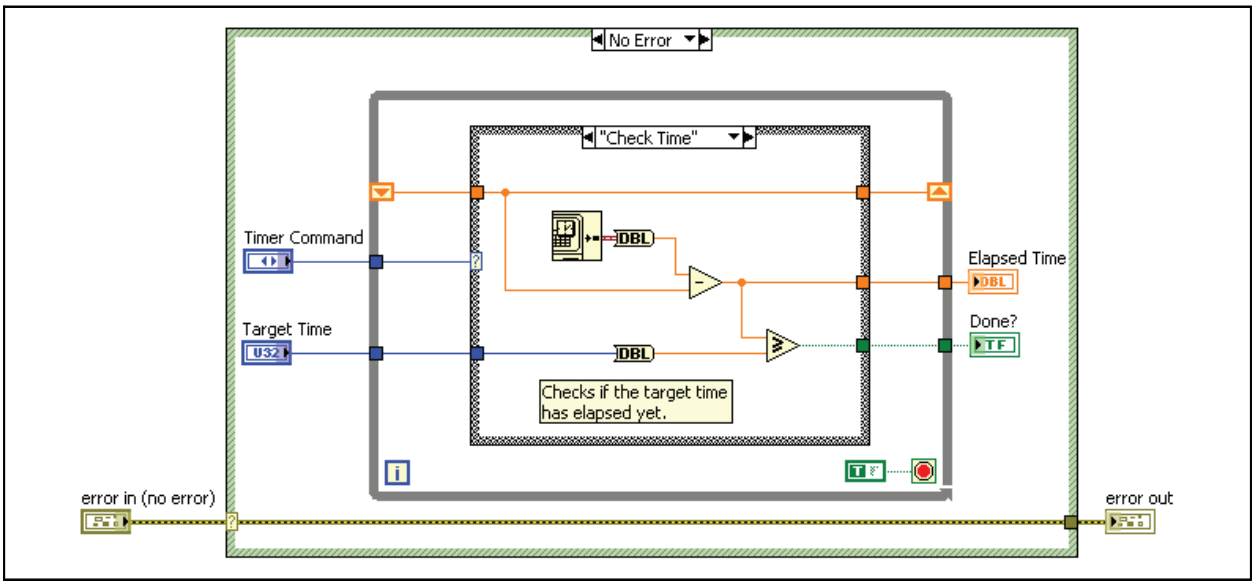


Figure 6-10. Timing Module Check Time Case

6. Save the VI.

Testing

A unit test VI is provided to verify the functionality of the Timing module. The unit test VI repetitively calls the Timing module and displays the elapsed time for the execution of the Timing module.

1. Use the unit test VI to test if the Timing module returns the values you specify in the **Cue** control.

   ❑ Open `tlc_Timing Module Unit Test.vi` from the **Timing** folder in the project tree.

   ❑ Examine the block diagram and observe the functionality of the unit test VI.

   ❑ Specify a value for the wait time, fade time, and follow time in the **Cue Input** control.

   ❑ Run the VI and verify that the times returned in **Measured Test Values** match what you specified in **Cue Input**.

   ❑ Close the unit test VI and tlc_Timing Module VI.

   ❑ Save the project.

End of Exercise 6-2

Exercise 6-3 Implement Code

Goal

Observe and implement the VIs that you identified as modules for the application.

Scenario

When you carefully plan and design an application, your implementation of the system creates scalable, readable, and maintainable VIs. Implement the display, file, and hardware modules. These are the major modules that you identified for the Theatre Light Control Software application. You have already implemented the cue and timing modules.

Design

The functional global variable approach you used to implement the cue and timing modules provides for a very scalable, readable, and maintainable method to build VIs. Continue to use a functional global variable to implement the remaining modules for the application.

Display Module

The tight coupling that exists between the front panel and block diagram in a VI requires that you update the front panel using block diagram terminals or references from a subVI. Each of these methods has advantages and disadvantages. Updating front panel controls and indicators using terminals is very fast and efficient. However, you must have a way to get subVI data to the top-level VI to update the front panel. You can loosen the tight coupling that exists between the front panel and the block diagram by sending a message from subVIs to the top-level VI that contains the control or indicator you want to update. An ideal implementation for this is a functional global variable.

LabVIEW is inherently a parallel programming language. You can take advantage of the parallelism by using a separate display loop in the main VI to update the user interface. The display loop contains a queue that stores commands to perform inside the loop. You can use a functional global variable to control the display loop from anywhere in the application by placing commands in the queue.

The display module uses a functional global variable to store the reference for the display queue. This allows the module to be called from anywhere in the application to control the display loop.

File Module

The file module calls the File I/O VIs. The file module provides the functionality to initialize, load cues from a file, save cues to a file, and shutdown. Implement the file module using the functional global variable architecture.

Hardware Module

The hardware module calls the Theatre Light Control API. The hardware module provides the functionality to initialize, write to, and shutdown the hardware.

Implementation

Display Module

The Display module provides a method to update the front panel controls and indicators from anywhere in the application. This module populates a queue that contains the commands the display module performs and the data needed to perform these tasks. The display module performs the Initialize Front Panel, Update Front Panel Channels, Select Cue in Cue List, Enable/Disable Front Panel Controls, and Update Cue List functions.

To build a system that can perform these functions, first modify the design pattern in TLC Main VI to have a display loop, then create the display module with a functional global variable that sends messages to the display loop.

Create a queue that specifically controls the display loop. The display loop only updates the user interface in the top-level VI.

Design Pattern Modification

1. Open the block diagram of the TLC Main VI.

2. Add a display loop to the block diagram, as shown in Figure 6-11. To create the initial structure of the display loop, use the following items:

   - ❑ While Loop—Add a While Loop below the consumer loop. This will be the display loop.

   - ❑ Case Structure—Add to the display loop.

   - ❑ Unbundle By Name function—Add to the display loop.

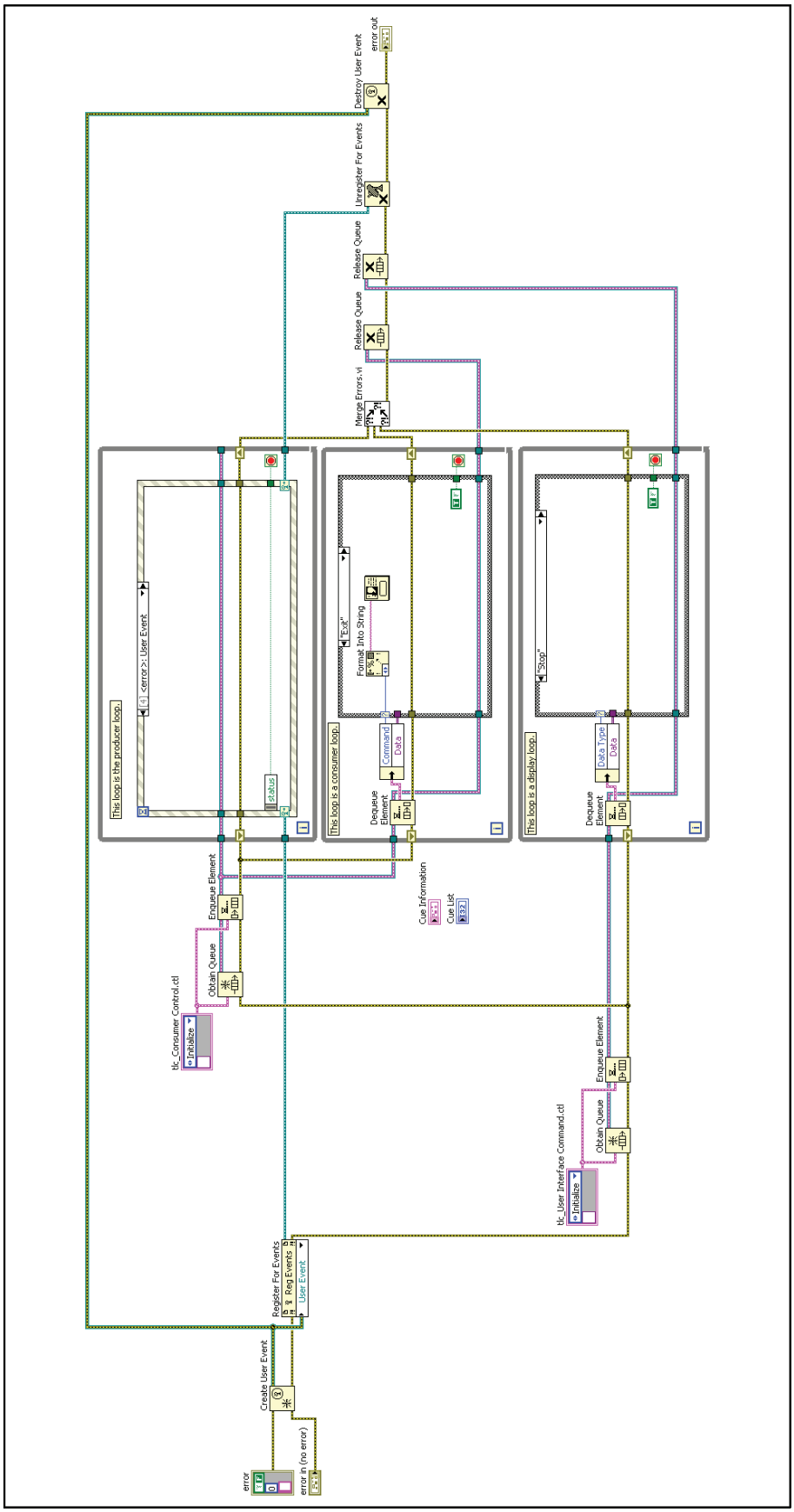


Figure 6-11. Display Loop

3. Create a queue to control the display loop, as shown in Figure 6-11. Use the following items:

   - `tlc_User Interface Command.ctl`—Right-click **Controls** in the project tree, select **Add»File** and navigate to `<Exercises>\LabVIEW Core 3\Course Project\Controls`. Add `tlc_User Interface Command.ctl` and `tlc_User Interface Data Type.ctl` to the project.

     `tlc_User Interface Data Type.ctl` is used as part of `tlc_User Interface Command.ctl`.

     Add `tlc_User Interface Command.ctl` as a block diagram constant. This constant uses a cluster of an enum and a variant to provide a scalable data type for the display loop. Set the enum value to **Initialize**.

   - Obtain Queue function

   - Enqueue Element function

   - Dequeue Element function—Wire the **element** output of the Dequeue Element function to the Unbundle By Name function.

   - Wire the **Data Type** output of the Unbundle By Name function to the case selector terminal.

   - Right-click the Case structure and select **Add Case for Every Value** from the shortcut menu.

   - Boolean constants

     – Add a False constant inside each case of the Case structure. Wire these constants to a single tunnel connected to the loop conditional terminal for the display loop.

     – Set the constant to True in the Stop case.

   - Release Queue function—This function releases the display loop queue references.

4. Wire the queue reference and error cluster through each case of the Case structure.

5. Add `tlc_Display Queue Reference.ctl` from `<Exercises>\LabVIEW Core 3\Course Project\Controls` to the **Controls** virtual folder. This control enqueues items to this queue from subVIs.

6. Save the TLC Main VI and the project.

Display Module

Create a module that stores which function the display should perform. As the application runs, update the display. Complete the following steps to create the display module.

1. Add the display module files and **Display** virtual folder to the **Modules** virtual folder.

   - ❑ Right-click **Modules** in the project tree and select **Add»Folder (Snapshot)** from the shortcut menu.

   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Modules\Display` directory and click **Current Folder** to add the folder and its contents to the project tree. LabVIEW adds the following files to the project.

     - `tlc_Display Module.vi`

     - `tlc_Display_Command Control.ctl`.

2. Open `tlc_Display Module.vi`.

3. Complete the Initialize case as shown in Figure 6-12 by adding a shift register. The Initialize case stores the queue reference in a shift register.

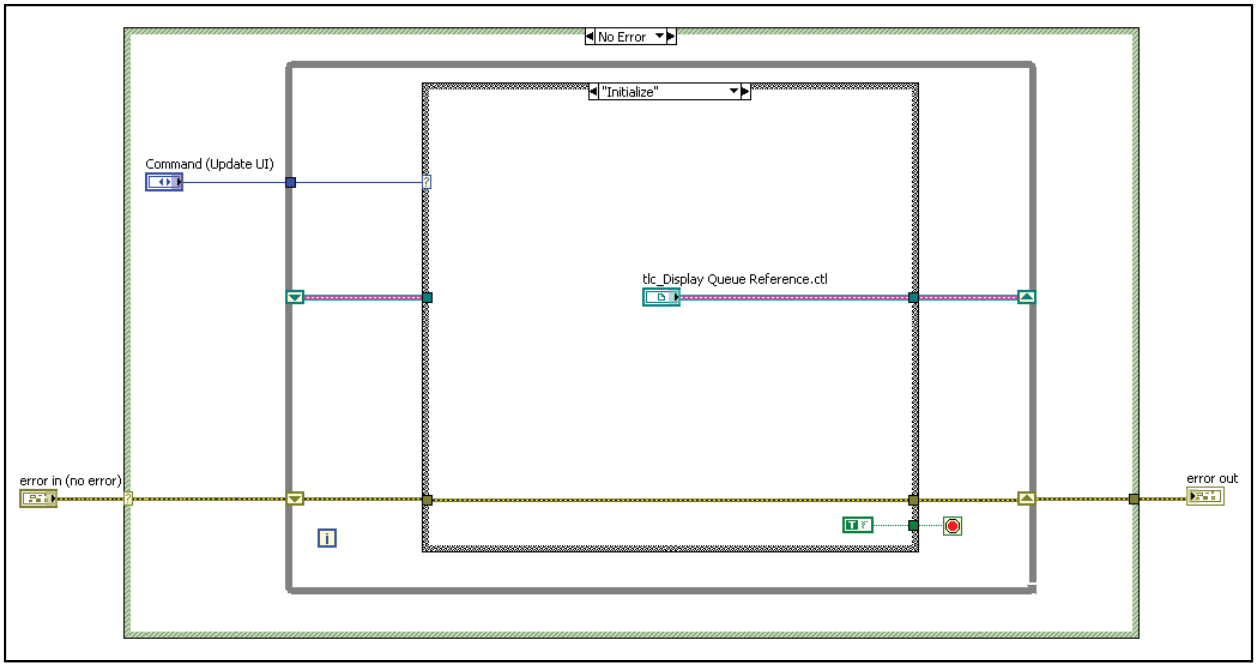


Figure 6-12. Display Module Initialize Case

- ❑ Wire the queue reference through the other cases.
- ❑ Wire all tunnels in the module.

4. Modify the Update UI case to call the Enqueue Element function to pass the command and data to the display loop in the top-level VI, as shown in Figure 6-13. Use the following items to complete this case:

- ❑ `tlc_User Interface Command.ctl`—From the **Controls** virtual folder.
- ❑ Bundle By Name function
- ❑ Enqueue Element function

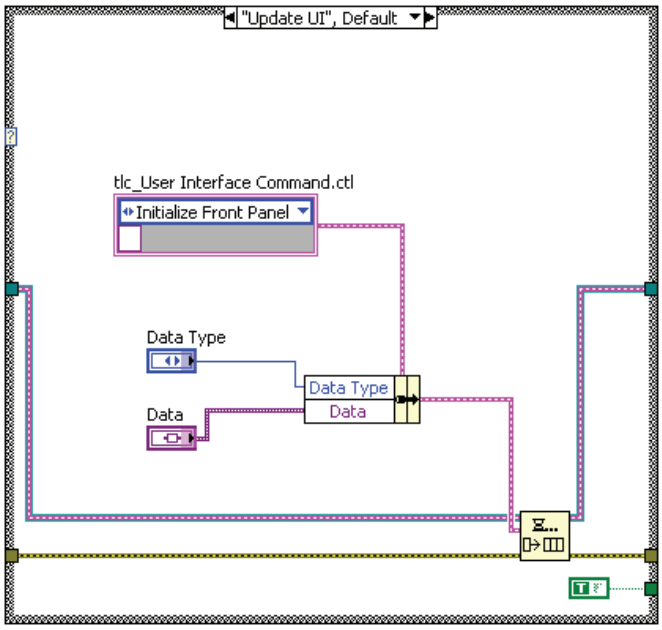


Figure 6-13. Display Module Update UI Case

5. Examine the connector pane as shown in Figure 6-14.

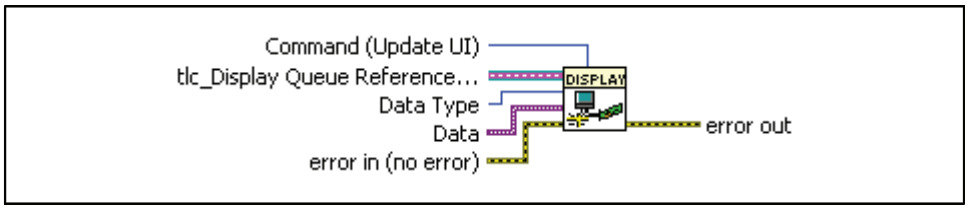


Figure 6-14. Display Module Icon and Connector Pane

6. Save and close the VI.

File Module

The file module saves and loads a file from the application. The file module accepts an array of cues, a file path, and a command, and returns an array of cues. The file module performs the Save Cues and Load Cues functions.

1. Add the file module files and **File** virtual folder to the **Modules** virtual folder.

   ❑ Right-click **Modules** in the project tree and select **Add»Folder (Snapshot)** from the shortcut menu.

- ❑ Navigate to the <Exercises>\LabVIEW Core 3\Course Project\Modules\File directory and click **Current Folder** to add the folder and its contents to the project tree. LabVIEW adds the following files to the project.
  - tlc_File Module.vi
  - tlc_File_Command Control.ctl

2. Save the project.
3. Open tlc_File Module.vi and observe the architecture and design of the module. The module uses standard File I/O VIs to read and write data.

Testing

Test the functionality of the file module with simple hand tests.

1. Set the **Command** control to **Save Cues**.
2. Click the **Browse** button and set a name and save location for a new file. The **Path** control has been set to browse for new or existing files.
3. Enter dummy data in the **Cue Array Input**.
4. Run the VI.
5. Set the **Command** control to **Load Cues**.
6. Run the VI.
7. Verify that **Cue Array Output** matches the data you entered in **Cue Array Input**.
8. Close the VI.

Hardware Module

The hardware module interacts with the Theatre Light Control API. The hardware module performs the Write Color and Intensity function.

1. Add the hardware module files and **Hardware** virtual folder to the **Modules** virtual folder.
   - ❑ Right-click **Modules** in the project tree and select **Add»Folder (Snapshot)** from the shortcut menu.

- ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Modules\Hardware` directory and click **Current Folder** to add the folder and its contents to the project tree. LabVIEW adds the following files to the project.
  - `tlc_Hardware Module.vi`
  - `tlc_Hardware_Command Control.ctl`

2. Open `tlc_Hardware Module.vi`.
3. Observe the architecture and design of the hardware module. Notice that the VI calls the Theatre Light Control API.
4. Close the VI.
5. Save the project.

End of Exercise 6-3

Exercise 6-4 Implement Error Handling Strategy

Goal

Develop a module that handles the errors in the application.

Scenario

Handling errors is an important part of developing applications in LabVIEW. When you are developing the application, you can use error handling to help find bugs in the applications.

A good error handling strategy is to call a module that stores the error information and safely stops the application if an error occurs.

Design

Using a functional global variable, store the error information in an uninitialized shift register. If an error occurs, the VI sends a stop message to the producer to shut the program down.

Implementation

1. Add `tlc_User Event Reference.ctl` and `tlc_Consumer Queue Reference.ctl` to the project. These custom controls are references to the user event created by Create User Event and the queue created by Obtain Queue in the TLC Main VI. These custom controls stop execution of the producer and consumer loops of the TLC Main VI if an error occurs.

   - ❑ Right-click the **Controls** virtual folder, select **Add»File** from the shortcut menu and navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Controls` directory.

   - ❑ Select `tlc_User Event Reference.ctl` and `tlc_Consumer Queue Reference.ctl` and click **Add File** to add the controls to the **Controls** virtual folder.

Tip <Ctrl>-click to select multiple files.

2. Add the error module files and **Error** virtual folder to the **Modules** virtual folder.

   - ❑ Right-click **Modules** in the project tree and select **Add»Folder (Snapshot)** from the shortcut menu.

- ❑ Navigate to the <Exercises>\LabVIEW Core 3\Course Project\Modules\Error directory and click **Current Folder** to add the folder and its contents to the project tree. LabVIEW adds the following files to the project.
  - tlc_Error Module.vi
  - tlc_Error Module Command Control.ctl

3. Open tlc_Error Module.vi.
4. Modify the block diagram to stop the producer, consumer, and display loops if an error occurs. Use the following items to complete the case where the structures are set to Handle Errors, Error, and True, as shown in Figure 6-15.

   - ❑ Flush Queue function—Use two instances of this function to empty the queues that control the consumer and display loops in the top-level VI.
   - ❑ Enqueue Element function—Use three instances of this function. After emptying the consumer and display queues, you need to add two elements to the consumer queue and add one element to the display queue to stop the consumer and display loops.

     Create constants for the **element** input of each instance. For the consumer queue, set the first enum to **Stop** and set the second enum to **Exit**. For the display queue, set the enum to **Stop**.
   - ❑ Generate User Event function—This function generates a user event that can stop the producer loop in the top-level VI.
   - ❑ Merge Errors VI.

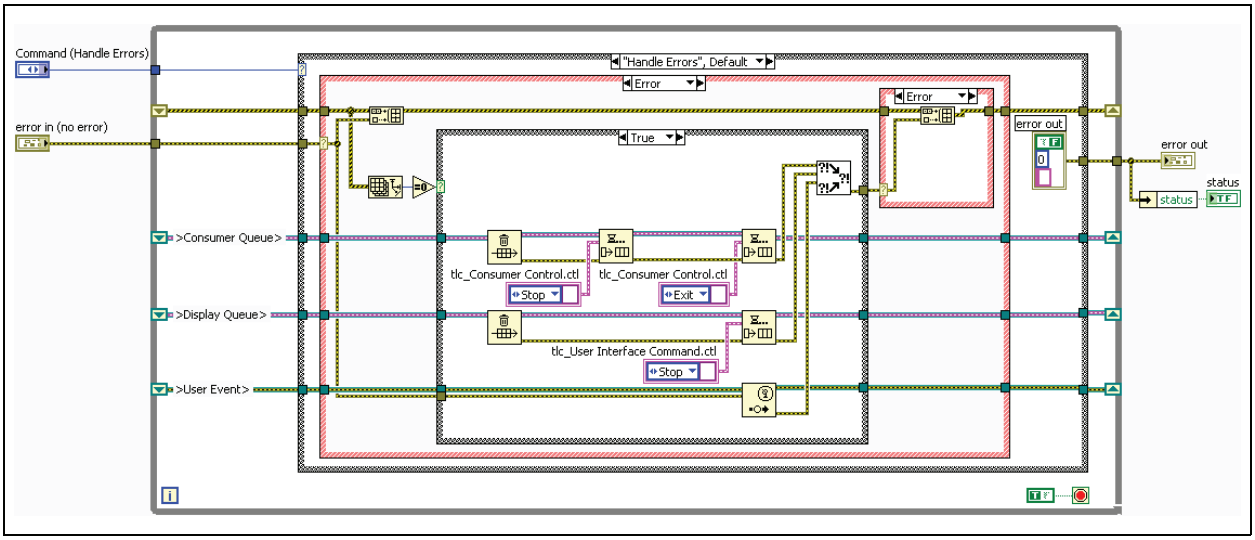


Figure 6-15. Error Handler Error Case

5. Complete all structures by wiring the User Event reference and queue references through all remaining cases.

6. Browse the VI and observe how it operates.

7. Save and close the VI.

8. Save the project.

End of Exercise 6-4

Notes

7

Implementing a Test Plan

Exercise 7-1 Integrate Initialize and Shutdown Functions

Goal

Initialize and shut down a set of code modules.

Scenario

The Initialize function places the application into a stable mode by initializing all of the modules and clearing the user interface.

When the user selects the **File»Exit** menu, the application should safely shut down. Safely shutting down an application requires closing all open memory references.

Design

The Initialize function outlined in the requirements document performs the following actions:

- Initialize Error Module
- Initialize Cue Module
- Initialize File Module
- Initialize Hardware Module
- Initialize Display Module
- Enable Front Panel Controls
- Initialize Front Panel
- Update Cue List

Each module is designed to close any open memory references when the shutdown function is called. The Shutdown VI accesses the shutdown function for each module. The Shutdown VI also sends a Stop message to the display loop to finish execution of the loop. The display, hardware, and file modules need to shut down. You can re-initialize the Cue module to delete any Cues that are stored in the persistent shift register.

Implementation

The implementation of the Initialize VI involves calling the modules that you built and wiring the correct data type to them. There is not a specific

order in which you must call the VIs. Follow the order in the *Design* section to be complete.

Initialize

1. Add the integration files and **Integration** virtual folder to the project.

   - ❑ Right-click **My Computer** in the project tree and select **Add» Folder (Snapshot)** from the shortcut menu.
   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Integration` directory and click **Current Folder** to add the folder and its contents to the project tree.

2. Open `tlc_Initialize.vi`.

3. Complete the block diagram as shown in Figure 7-1. This VI initializes the modules that you have already created. The block diagram already includes these modules. Use the following items to initialize the modules:

   - ❑ Numeric constant—Create this constant with a representation of **U8** and a value of `0`. This constant specifies whether the controls of the main user interface are enabled or disabled. A value of zero enables the controls.
   - ❑ To Variant function—This function converts a numeric to the variant datatype, which is passed to the **Data** input of the tlc_Display Module VI.

Note In later instances, the **Data** input of the tlc_Display Module VI is set to `2`, which disables the controls.

   - ❑ Right-click the **Command** input of tlc_Error Module VI, select **Create»Constant** and set the constant to **Initialize**. Repeat for each of the following modules, as shown in Figure 7-1:
     - tlc_Cue Module VI
     - tlc_File Module VI
     - tlc_Hardware Module VI
     - tlc_Display Module VI
   - ❑ Right-click the **Data Type** input of the second instance of the tlc_Display Module VI. Select **Create Constant** and set the value as shown in Figure 7-1. Repeat this step for the third and fourth instances of tlc_Display Module VI.

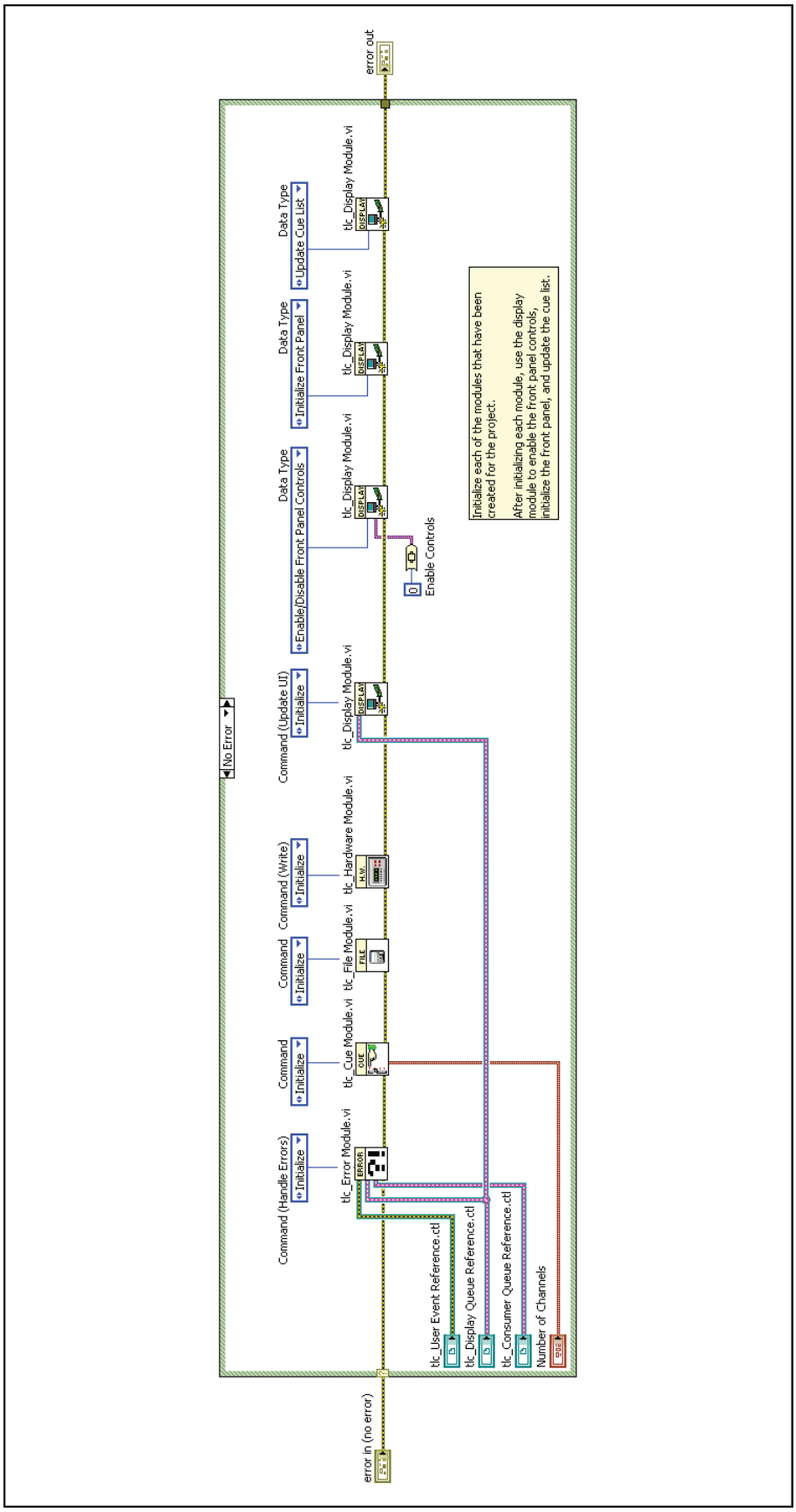


Figure 7-1. Initialize Function

4. Modify the connector pane for the tlc_Initialize VI, as shown in Figure 7-2.

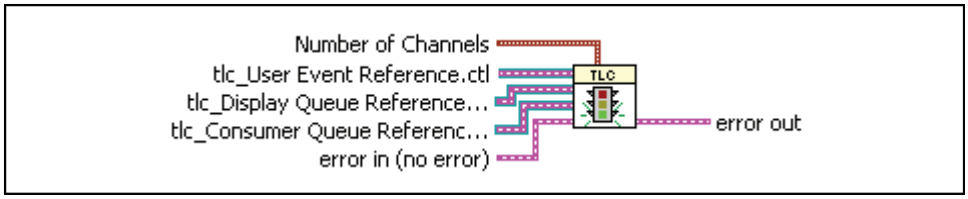


Figure 7-2. Initialize Function Icon and Connector Pane

5. Save and close the VI and save the project.

6. Open the TLC Main VI.

7. Integrate `tlc_Initialize.vi` into the consumer loop of the TLC Main VI as shown in Figure 7-3. This case executes when the application starts. To build the case, use the following items:

Note As you integrate each function into the application, delete the Format Into String and One Button Dialog functions in the corresponding case of the Case structure.

- `tlc_Initialize.vi`—Add this VI to the Initialize case of the consumer loop.
- Number of Channels—Create this constant from the **Number of Channels** input of the tlc_Initialize VI. This determines the number of rows and channels to initialize in the **Cue Information** indicator. Set the values of this constant to initialize 4 rows and 4 columns.

Note The labels for the cluster constants in Figure 7-3 have been made visible to differentiate between the parameters in each cluster.

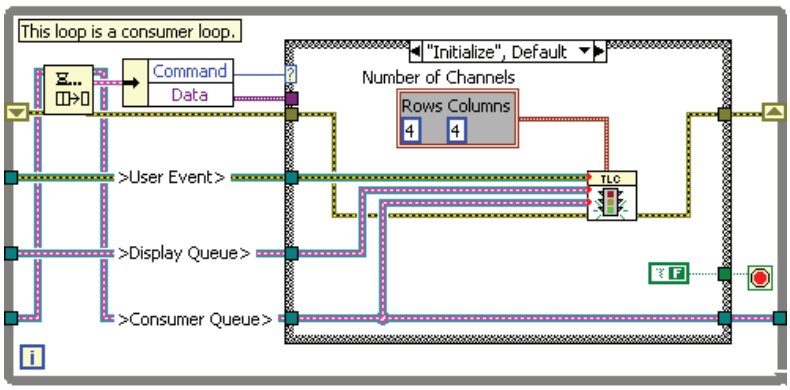


Figure 7-3. Initialize Case in Consumer Loop

8. Wire the Initialize case of the consumer loop.

   ❑ Wire the **user event out** output of the Create User Event function outside the While Loops to the **tlc_User Event Reference.ctl** input of the tlc_Initialize VI.

   ❑ Wire the **queue out** output of the Obtain Queue function outside the display loop to **tlc_Display Queue Reference.ctl** input of the tlc_Initialize VI.

   ❑ Wire the **queue out** output of the Dequeue Element function inside the consumer loop to the **tlc_Consumer Queue Reference.ctl** input of the tlc_Initialize VI.

9. Save and close the VI.

Shutdown

1. Open `tlc_Shutdown.vi` from the **Integration** virtual folder.

2. Complete the block diagram as shown in Figure 7-4.

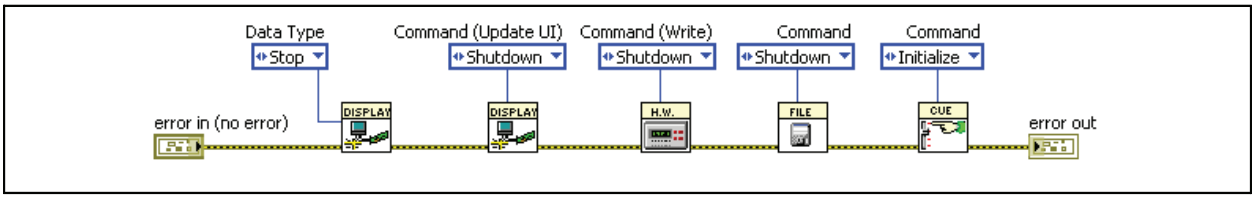


Figure 7-4. Shutdown Function

   ❑ Because the Cue module is set to **Initialize**, it initializes the shift register in the Cue module with an empty Cue array.

3. Save and close the VI.

4. Open the TLC Main VI and add the tlc_Shutdown VI to the Exit case of the consumer loop. Run the error wire through the tlc_Shutdown VI.

5. Save the VI.

Testing

1. Run the TLC Main VI and select **File»Exit**. Verify that the shutdown function causes the application to end. After you integrate the display functionality, you also can verify that the initialize function executes.

End of Exercise 7-1

Exercise 7-2 Integrate Display Module

Goal

Update the front panel controls.

Scenario

The application architecture splits the functionality into three separate loops. The producer loop uses an Event structure to monitor changes to the front panel. The consumer loop handles all the processing for the application. The display loop updates the user interface. The advantage of this architecture is that functionality is contained within individual parallel processes. This improves the performance and stability of the application. The three loop architecture also improves the maintainability and scalability of the application.

Implement the code in the display loop to update the user interface.

Design

In order to perform the functions specified in the requirements document, you need to have the following display functions:

- Initialize Front Panel
- Update Front Panel Channels
- Select Cue in Cue List
- Enable/Disable Front Panel Controls
- Update Cue List.

Implement these functions in the display loop of the top-level VI. The advantage to implementing this code in the top-level VI is that you have direct access to the front panel terminals.

Implementation

Edit the following cases in the display loop Case structure of the TLC Main VI.

Initialize Front Panel

This case initializes the 2D array of channels in **Cue Information** to the color black and the intensity zero. It also increments channel numbers across the array.

1. Modify the Initialize Front Panel case as shown in Figure 7-5 using the following items.

   - ❑ `tlc_Cue Module.vi`—Initializes the 2D array of channels in **Cue Information**.
   - ❑ Create a constant from the **command** input of the tlc_Cue Module VI and set its value to **Get Empty Cue**. This sets the Cue Module to return an empty cue with an intensity of zero, a color of black, and configured channel numbers.
   - ❑ Cue Information local variable—Right-click the Cue Information terminal and select **Create»Local Variable** from the shortcut menu.

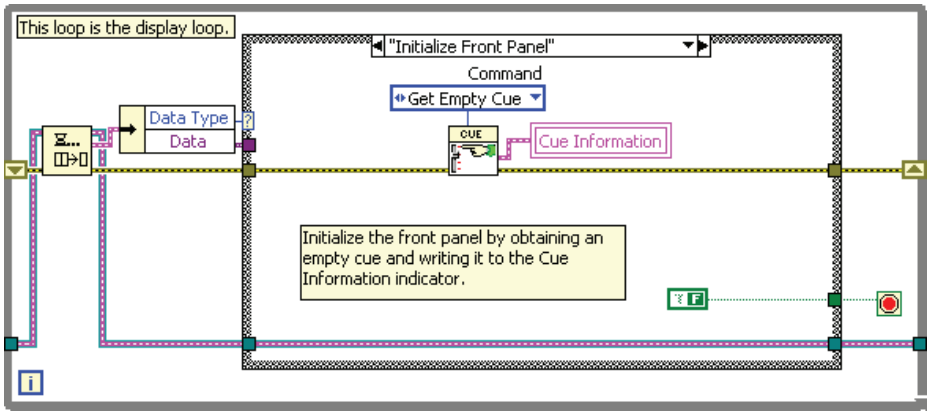


Figure 7-5. Initialize Front Panel

2. Save the VI.

Select Cue in Cue List

This case highlights a row in the Cue List when the user selects a cue or as the application iterates through cues.

1. Modify the Select Cue in Cue List case of the display loop as shown in Figure 7-6. Use the following items:

   - ❑ Variant To Data—Converts the variant datatype of **Data** into the datatype specified by the **type** input.
   - ❑ I32 Numeric constant—Converts **Data** into this datatype.
   - ❑ Cue List terminal—Move this terminal into the Select Cue in Cue List case of the display loop.

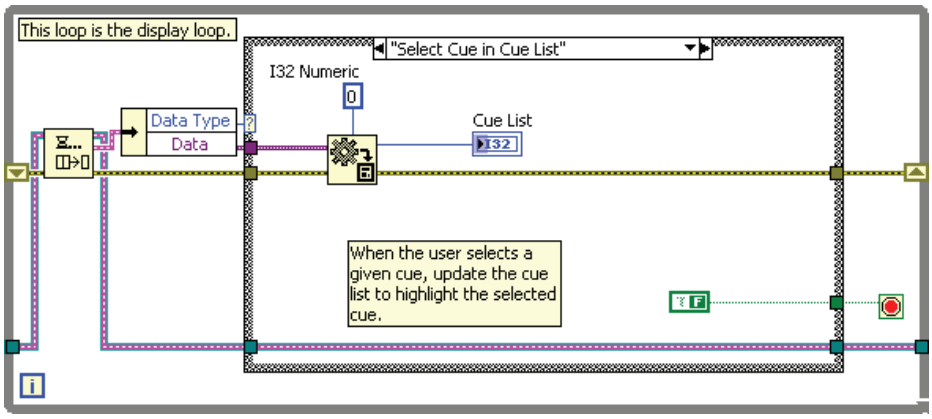


Figure 7-6. Select Cue in Cue List Case

2. Save the VI.

Enable/Disable Front Panel Controls

This case enables or disables the **Record** and **Cue List** controls.

1. Modify the Enable/Disable Front Panel Controls case as shown in Figure 7-7. Use the following items:

   - ❑ Cue List Property Node—Right-click the Cue List terminal or indicator and select **Create»Property Node»Disabled**. Right-click the Property Node and select **Change All To Write**.
   - ❑ Record Property Node—Right-click the Record terminal or button and select **Create»Property Node»Disabled** to create the Record Disabled Property Node. Right-click the Property Node and select **Change All To Write**.
   - ❑ Variant to Data function
   - ❑ U8 Numeric constant—Connect this constant to the **type** input of Variant to Data to set the **data** output to the correct datatype for Cue List. A **data** output of `0` enables the controls. A **data** output of `1` disables the controls. A **data** output of `2` disables and dims the controls.

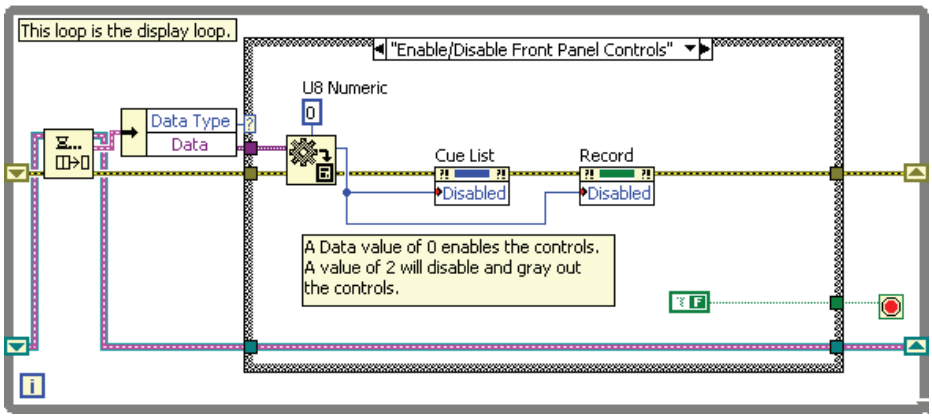


Figure 7-7. Enable/Disable Front Panel Controls Case

Note Coercion dots appear where you wire **data** to the Property Nodes because the Property Nodes expect an enum instead of a numeric value.

2. Save the VI.

Update Cue List

This case retrieves the recorded cues and updates the Cue List. After you determine the number of cues, obtain and display the names of the cues in order.

1. Modify the Update Cue List case of the display loop as shown in Figure 7-8. Use the following items:

   - ❑ For Loop
   - ❑ First instance of `tlc_Cue_Module.vi`
     - – Add the VI inside the Case structure and outside the For Loop.
     - – Right-click the **command** input of the Cue Module. Select **Create Constant** and set the value to **Get Number of Cues** to return the number of cues in the current Cue List.
     - – Wire the **Number of Cues** output to the count terminal of the For Loop to execute the loop once for each cue in the list.
   - ❑ Second instance of `tlc_Cue_Module.vi`
     - – Add the VI inside the For Loop.
     - – Right-click the **command** input of the Cue Module and select **Create Constant**. Set the value to **Get Cue Values** to return a

cluster containing the cue name, wait time, fade time, follow time, and the 2D array of channels associated with the cue.

- Unbundle By Name—Wire **Cue Output** from the Cue Module to the input cluster to retrieve the **Cue Name** element.

❑ Wire the error clusters of the Cue Module VIs through the For Loop. Create shift registers on the For Loop error tunnels.

❑ Cue List Property Node—Create a Property Node to modify the Item Names property of the Cue List indicator.

- Right-click the Property Node and select **Change All To Write** from the shortcut menu.
- Wire the **Cue Name** output of the Unbundle By Name function to the Cue List Property Node.
- Verify that auto-indexing is enabled for the Cue Name For Loop output tunnel.

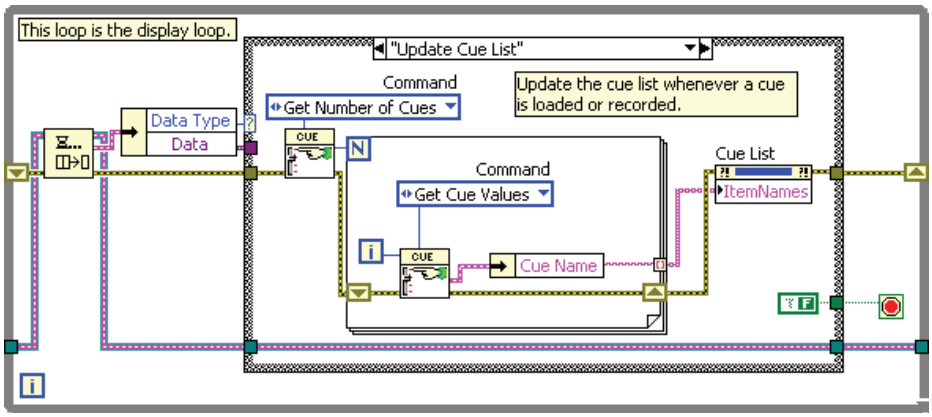


Figure 7-8. Update Cue List Case

2. Save the VI.

Update Front Panel Channels

This case updates the value of the 2D array of channels. The VI calls this case every time the channel data changes, such as when a cue is playing and the channel intensities and colors change. The VI also calls this case when the user selects a cue from the Cue List to display the information for the selected cue.

1. Modify the block diagram for the Update Front Panel Channels case of the display loop as shown in Figure 7-9. Use the following items:

   - ❑ Variant To Data function
   - ❑ `tlc_Cue_Information.ctl`—Add this control to the block diagram as a constant and wire it to the **type** input of the Variant to Data function. This converts the **Variant** input of Variant to Data to a cue in order to display the cue information.
   - ❑ Cue Information terminal—Move this terminal into the Update Front Panel Channels case of the display loop.

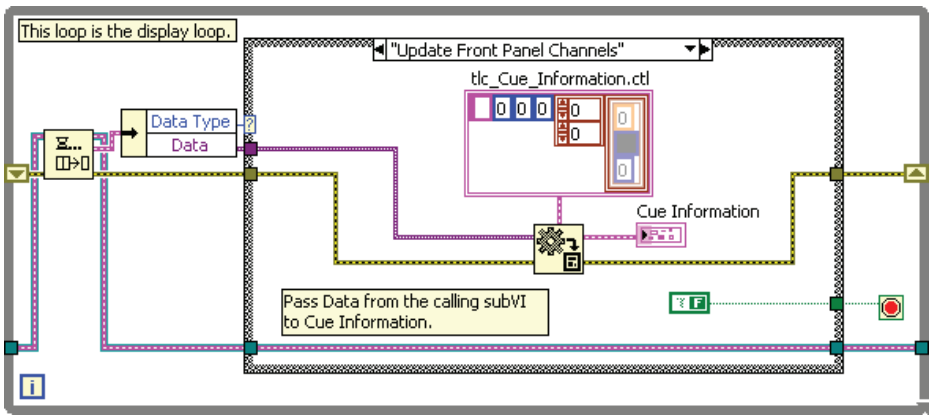


Figure 7-9. Update Front Panel Channels Case

2. Save the VI.

Testing

1. Run the VI.
2. The VI calls the Initialize function. The function initializes the front panel. All the front panel channels should initialize with the correct channel numbers and the color black.
3. Select **File»Exit** to exit the VI.
4. Close the VI.

End of Exercise 7-2

Exercise 7-3 Integrate Record Function

Goal

Pass data from the user interface to the rest of the application. Use an Event structure to create a dialog box where the user enters cue information to record a cue.

Scenario

The record function must prompt the user for the Cue Name, Wait Time, Fade Time, Follow Time, and settings for the channels. Prompt the user with a dialog box where the user can enter values. The values are passed to the Cue module and the user interface updates.

Design

A **Record** dialog box has been created using the Dialog Based Events framework. This framework uses an Event structure to monitor user interface events. Complete the functionality to retrieve the inputs from the user. Pass a Cue Data type from the producer loop into the consumer loop to store the recorded cue in the application. Use a variant for the design pattern makes it easier to perform this functionality.

Implementation

1. Open the tlc_Record Dialog Box VI from the **Integration** virtual folder. When this dialog box displays, the user enters values for **Cue Name**, **Wait Time (s)**, **Fade Time (s)**, and **Follow Time (s)** and sets cue intensities and colors in the channel array.

2. Set the minimum value for **Fade Time (s)** to one second.

   - ❑ Right-click the control and select **Data Entry**.
   - ❑ Remove the checkmark from **Use Default Limits** and enter `1` as the **Minimum** value.
   - ❑ Click **OK**.

3. Open the block diagram and create a constant from the **command** input of the tlc_Cue Module VI. Set the command to **Get Empty Cue**, as shown in Figure 7-10.

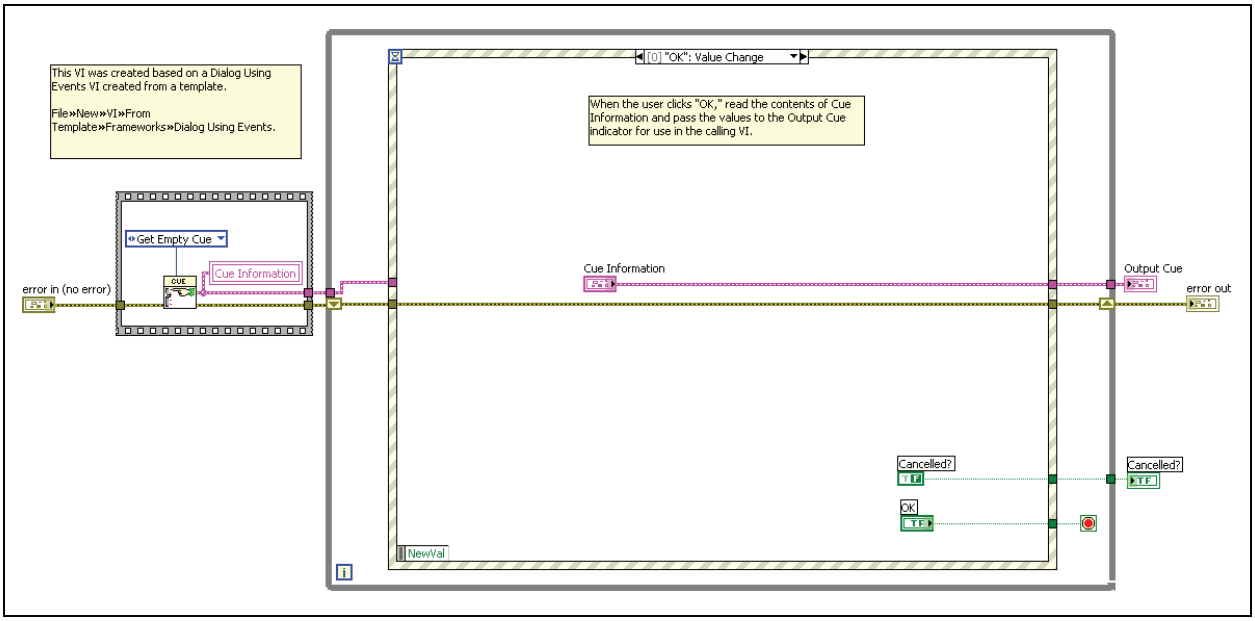


Figure 7-10. Record Function Dialog Box

4. Modify the connector pane to pass the **Output Cue** parameter, as shown in Figure 7-11.

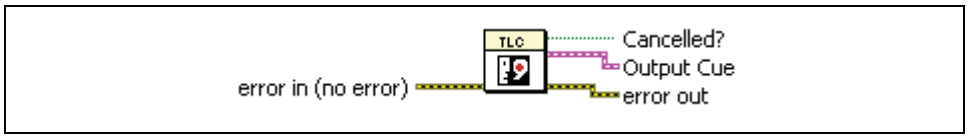


Figure 7-11. Record Dialog Box Connector Pane

5. Prepare the VI for use as a dialog box. When complete, it should be similar to Figure 7-12.

   ❑ Modify the front panel of the VI to expand the input cluster, align the objects, and hide the error clusters.

   ❑ Resize the front panel to show only the buttons and the input cluster.

   ❑ Select **File»VI Properties**, and select **Window Appearance** from the **Category** pull-down menu. Verify that window style is set to **Dialog**. Change the **Window Title** to `Cue Record` and click **OK**.

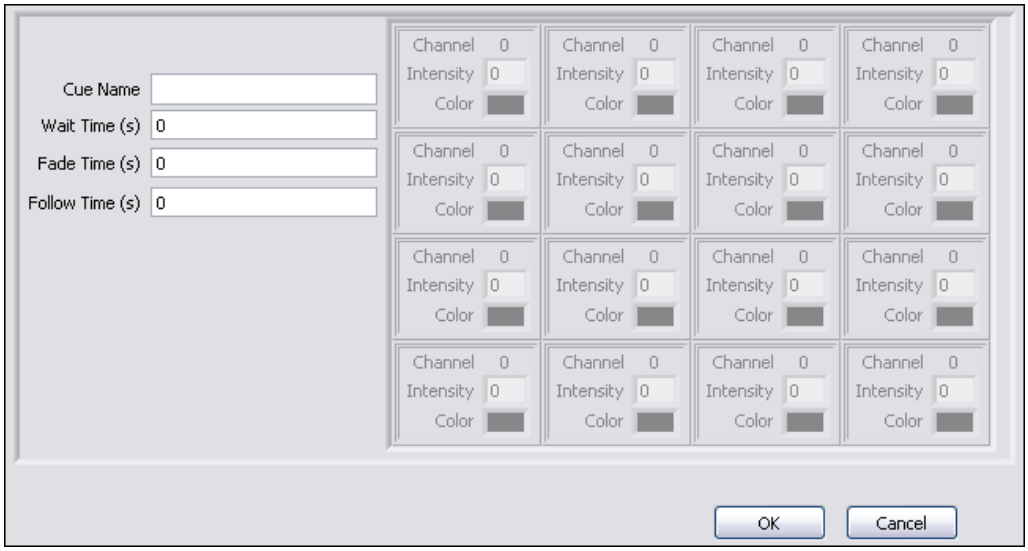


Figure 7-12. Front Panel of Record Dialog Box

6. Save and close the VI.

7. Modify the TLC Main VI to call `tlc_Record Dialog Box.vi` in the producer loop and pass the Cue data to the Record function in the consumer loop as shown in Figure 7-13. Use the following items:

   - `tlc_Record Dialog Box.vi`—Obtains the cue that the user wants to record.
   - Case structure—Add the Case structure around the Enqueue Element and Bundle By Name functions, and make this case False.
     - Wire the **Cancelled?** output of the tlc_Record Dialog Box VI to the case selector terminal. If the user cancels or closes the dialog box, no cue should be recorded.
   - To Variant function—Wire the **Output Cue** of the tlc_Record Dialog Box VI to this function to convert it to the variant datatype the Bundle By Name function requires.
   - Wire the queue refnum and error wires through the True case.

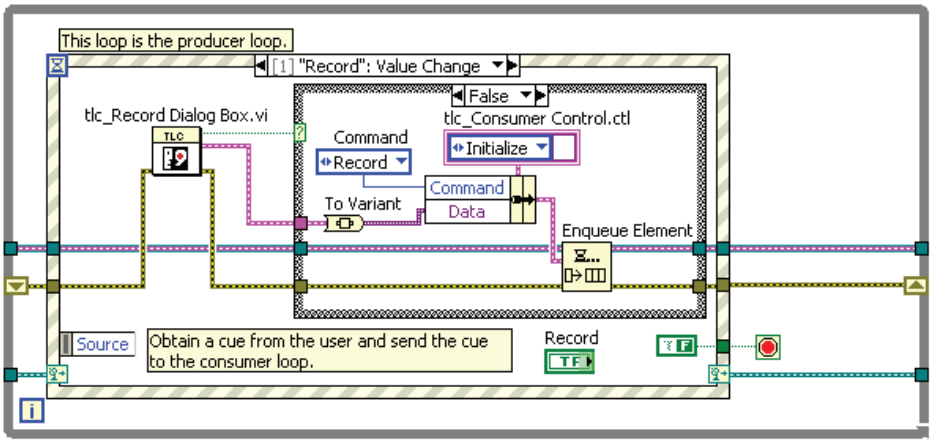


Figure 7-13. Record Event Case

8. Save the VI.

9. Test the TLC Main VI. Verify that the **Cue Record** dialog box appears when you click **Record**. After clicking **OK** in the **Cue Record** dialog box, a One Button Dialog should appear indicating that you are in the Record case of the consumer loop. Click **OK**.

10. Select **File»Exit** to stop the VI.

11. Open the tlc_Record VI from the **Integration** virtual folder and examine the block diagram. The VI first uses the display module to update the 2D array of channels. Next, it calls the cue module to add the new cue to the functional global variable cue array. Finally, it calls the display module again to update the Cue List.

12. Close the VI.

13. Modify the consumer loop to call the Record VI and convert the variant data into data that the Record VI can accept. Figure 7-14 shows the completed Record case in the consumer loop. Use the following items:

    - ❑ Variant To Data function
    - ❑ `tlc_Cue_Information.ctl` constant
    - ❑ `tlc_Record.vi`

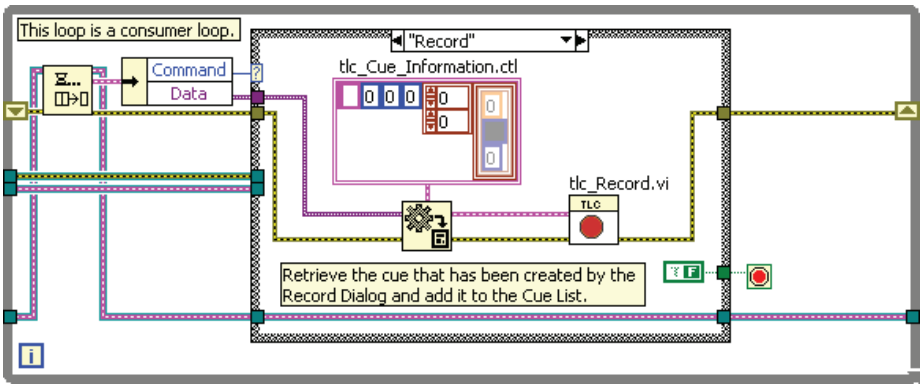


Figure 7-14. Record Case in Consumer Loop

14. Save the VI.

Testing

1. Run the TLC Main VI.
2. Click **Record** and record a cue.
3. If the application is running correctly, the front panel displays the cue you recorded, and the Cue List updates with the name of the cue that you entered in the **Cue Record** dialog box.
4. Select **File»Exit** to stop the VI.

Note If time permits, proceed to Exercise 7-7. Alternately, you can copy the save and load functionality into your project at the beginning of Exercise 7-5.

End of Exercise 7-3

Exercise 7-4 Integrate Play Function

Goal

Execute a state machine design pattern in a producer/consumer (events) design pattern.

Scenario

The Play functionality for the Theatre Light Controller is best implemented as a state machine. Figure 7-15 shows the states of the Play function.

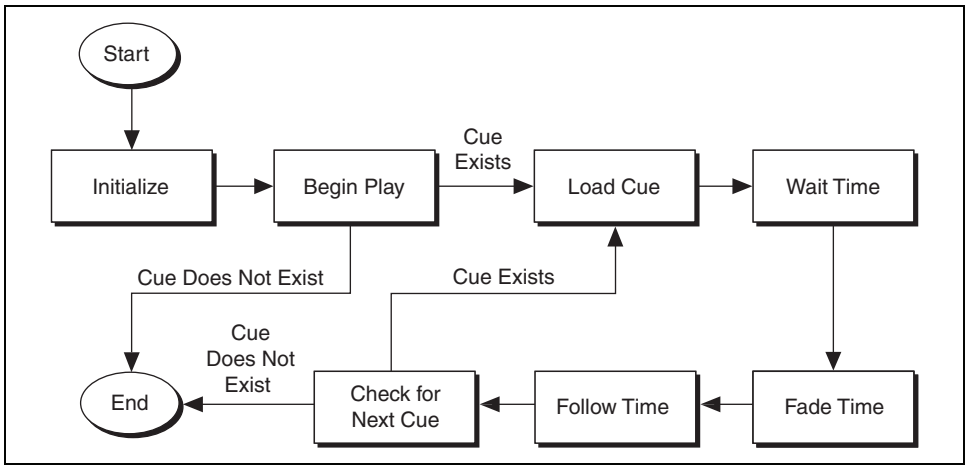


Figure 7-15. Play Function Flowchart

Implementing a state machine inside a producer/consumer (events) design pattern requires some insight into how the producer/consumer (events) design pattern operates. As you have seen in previous exercises, the design pattern receives an event in the producer loop and sends a message to the consumer loop to do some computation on the event. The consumer loop is designed to compute a single message from the producer. Implementing a state machine requires a looping mechanism that is not native to the producer/consumer (events) design pattern. You can approximate a loop with the producer/consumer (events) design pattern by placing a message in the consumer queue. In your application, you want to stay in the Play function to implement the wait, fade, and follow timing requirements for the application. Add messages in the consumer queue to stay in the Play case in the consumer loop until the Play function completes. This method of implementing a state machine inside the producer/consumer (events) design pattern introduces some complexities when you implement the capability to stop a play.

Design

The Play function uses a state machine and the timing module to implement the wait, fade, and follow timing requirements.

After you develop the play module, integrate the play function into the TLC Main VI.

Implementation

Build the play functionality to add in the consumer loop. Start by examining the play VIs.

1. Open `tlc_Play_Update Cue.vi` from the **Integration** virtual folder and examine how the VI operates.

   The VI calls the Light Color Controller VI, which calculates a color based on desired color, desired intensity, elapsed time, and target time.

   The VI sends a command to the hardware module and display module.

   ❑ Close the VI.

2. Open `tlc_Play.vi` from the **Integration** virtual folder and examine how the VI operates. The VI uses a state machine to perform the Wait, Fade, and Follow timing requirements for the application.

   The VI uses the state machine design pattern with a True constant wired to the loop conditional terminal. This VI is designed to be called repeatedly in order to move from one state to the next. To control the VI, the state machine returns a Boolean value to indicate whether the cue list has completed execution. Examine how the Initialize, Begin Play, and Load Cue states operate.

3. Modify the Wait Time state in the tlc_Play VI to use the tlc_Timing Module functional global variable to control the timing, as shown in Figure 7-16. Use the following items:

   ❑ Unbundle By Name function—Wire **Current Cue** to this function and set the cluster element to **Wait Time (s)**.

   ❑ `tlc_Timing Module.vi`

   – Add two tlc_Timing Module VIs inside the Wait Time state.

   – Right-click the **Timer Command** input of each instance of the tlc_Timing Module VI and select **Create»Constant**. Set each constant as shown in Figure 7-16.

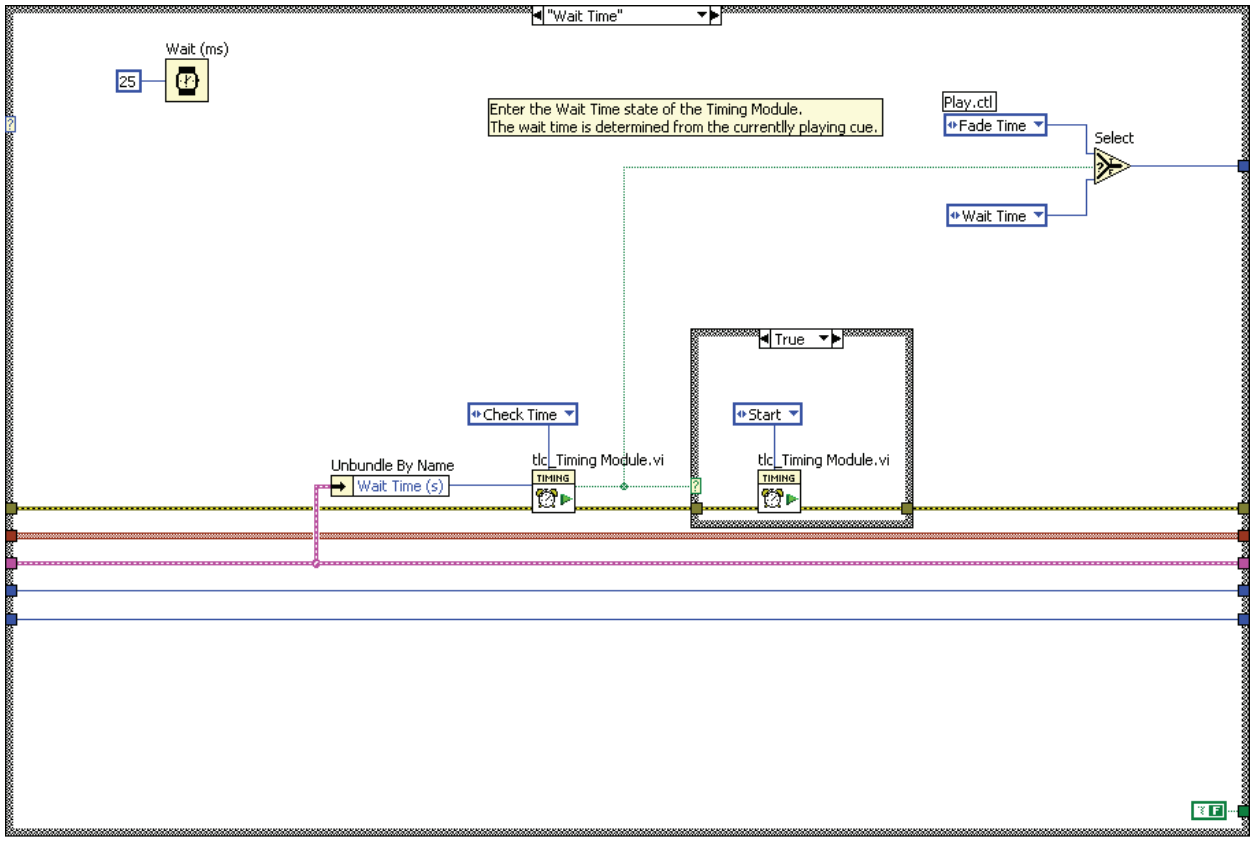


Figure 7-16. Play Module Wait Time Case

- ❑ Case structure
  - – Add the Case structure around the second tlc_Timing Module VI and ensure that this VI is in the True case.
  - – Wire the **Done?** output of the tlc_Timing Module VI to the case selector terminal. If the tlc_Timing Module VI outputs True from **Done?**, the Case structure sends the timing module a Start command to set a new start time.
  - – Wire the error cluster through the False case.
- ❑ Select function—Wire the Select function output to the top Case structure output tunnel. This tunnel is wired to the While Loop shift register. When the tlc_Timing Module VI outputs True from the **Done?** output, the Select function transitions the state machine to the Fade Time state.
- ❑ Wait (ms) function—Add a Wait (ms) function to the case and wire a numeric constant of `25` to the Wait function.

4. Modify the Fade Time state in the tlc_Play VI to use the tlc_Timing Module functional global variable to control the timing as shown in Figure 7-17. Use the following items:

   ❑ Unbundle By Name function—Wire **Current Cue** to this function and set the cluster element to **Fade Time (s)**.

   ❑ `tlc_Timing Module.vi`

   – Add two copies of this VI inside the Fade Time state.

   – Right-click each tlc_Timing Module VI and select **Create» Constant**. Set each constant as shown in Figure 7-17.

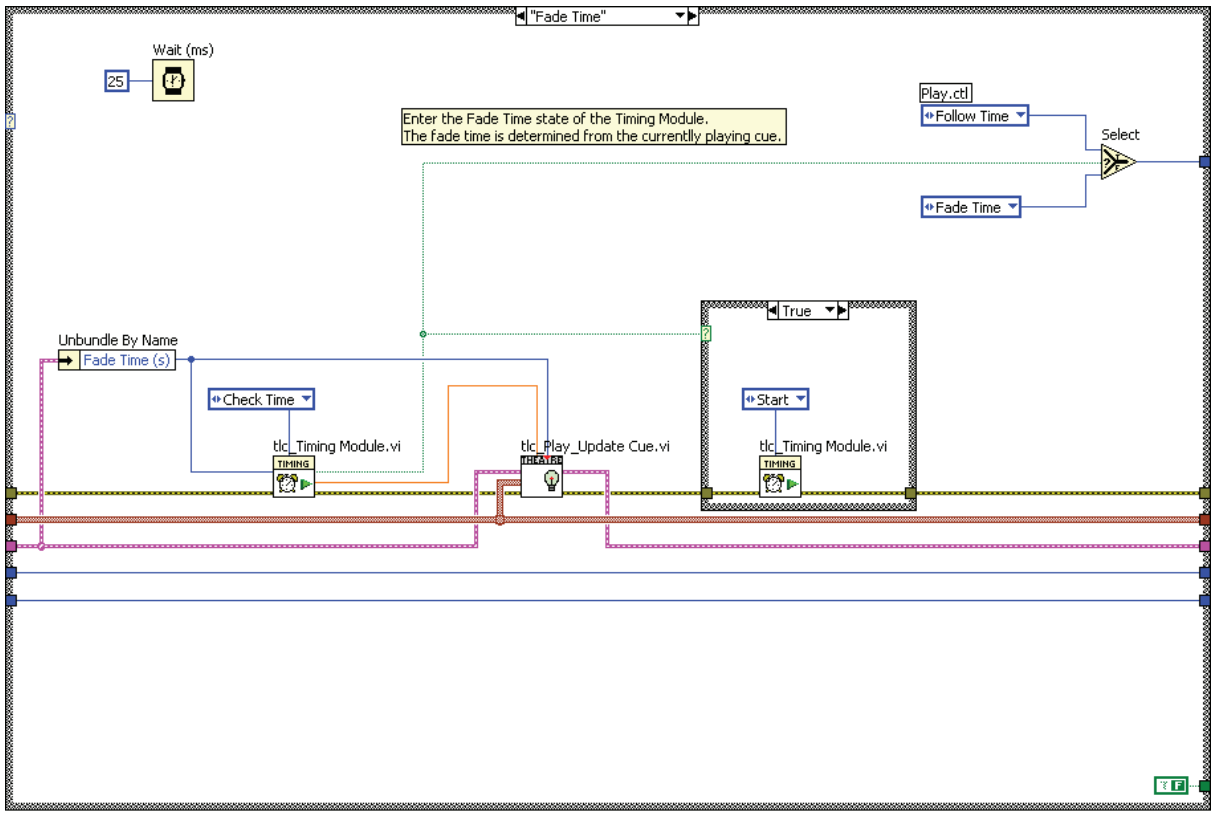


Figure 7-17. Play Module Fade Time Case

❑ `tlc_Play_Update Cue.vi`—Wire the terminals of this VI as shown in Figure 7-17.

- ❑ Case structure
  - – Add the Case structure around the second tlc_Timing Module VI, and ensure that this VI is in the True case.
  - – Wire the **Done?** terminal of the tlc_Timing Module VI to the case selector terminal. If the tlc_Timing Module VI outputs True from the **Done?** output, the Case structure sends the tlc_Timing module a Start command to set a new start time.
  - – Wire the error cluster through the False case.
- ❑ Select function—Wire the Select function output to the top Case structure output tunnel. This tunnel is wired to the While Loop shift register. When the tlc_Timing Module VI outputs True from the **Done?** output, the Select function transitions the state machine to the Follow Time state.
- ❑ Wait (ms) function with a 25 ms wait.

5. Modify the Follow Time state in the tlc_Play VI to use the tlc_Timing Module functional global variable to control the timing as shown in Figure 7-18. Use the following items:
   - ❑ Unbundle By Name function—Wire **Current Cue** to this function and set the cluster element to **Follow Time (s)**.
   - ❑ `tlc_Timing Module.vi`—Right-click the tlc_Timing Module VI and select **Create»Constant**. Set the constant to **Check Time**.
   - ❑ Select function—Wire the Select function output to the top Case structure output tunnel. This tunnel is wired to the While Loop shift register. When the tlc_Timing Module VI outputs True from the **Done?** output, the Select function transitions the state machine to the Check for Next Cue state.
   - ❑ Wait (ms) function with a 25 ms wait.
6. Save and close the tlc_Play VI.

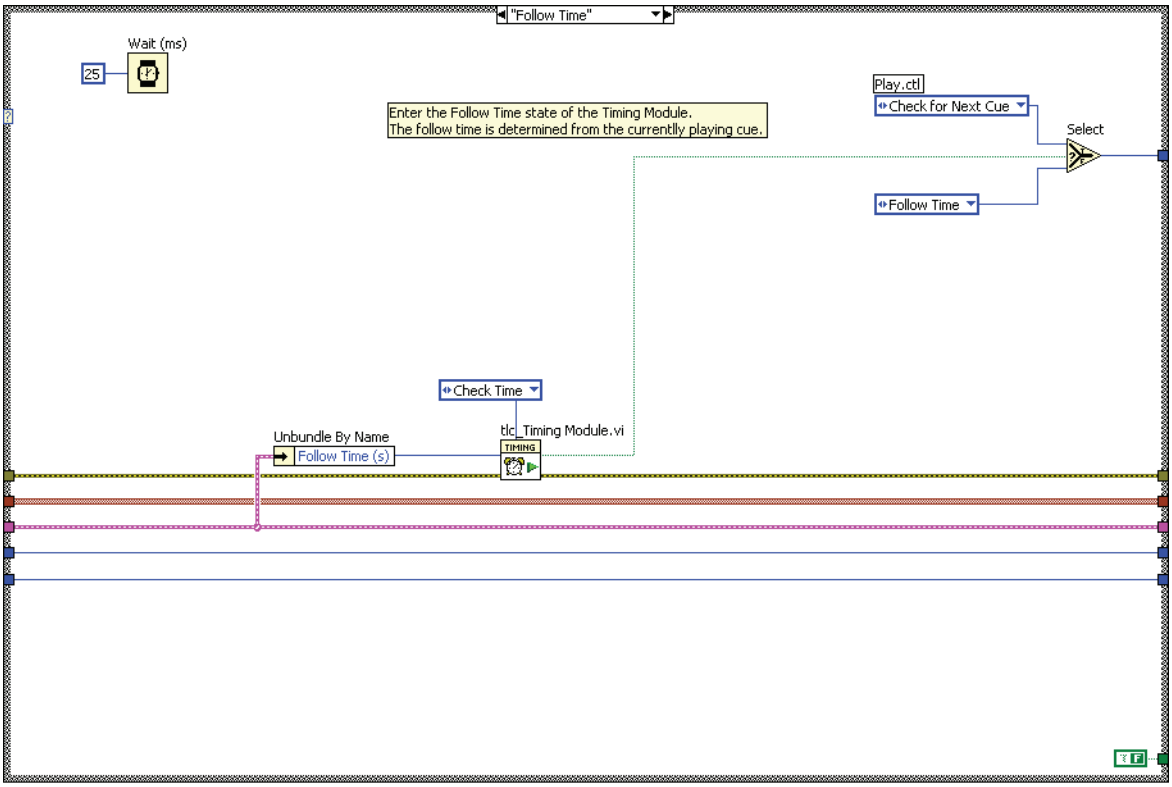


Figure 7-18. Play Module Follow Time Case

7. Open TLC Main VI and modify the Play state in the consumer loop to call the tlc_Play VI. Add the functionality to repeatedly call the tlc_Play VI, as shown in Figure 7-19. Use the following items:

   - ❑ Case Structure—Wire the **Stop?** output of the tlc_Play VI to the case selector terminal.
   - ❑ Enqueue Element function
   - ❑ `tlc_Consumer Control.ctl` constant—Set the enum for the constant to **Play**.
   - ❑ Run wires through the True case.

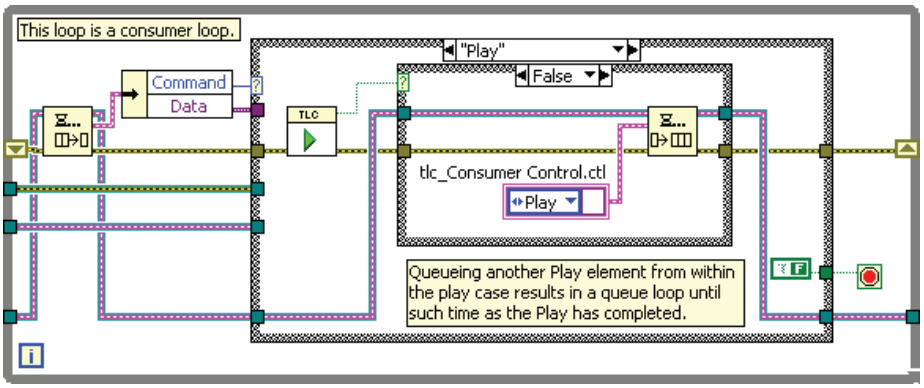


Figure 7-19. Play Case in Consumer Loop

8. Save the VI.

Testing

At this point, you can run the VI, record a cue, and play back the cue.

1. Run the VI.
2. Record a cue with a single channel that contains a color other than black, and an intensity of `100`. Set the Wait time to `0`, Fade time to `10`, and Follow time to `0`. Click **OK**.
3. Click **Play**.
4. The channel that you recorded should fade from 0% to 100% intensity within 10 seconds.
5. Try recording another cue and observe the response of the system.
6. Select **File»Exit** to stop the VI.

Note If time permits, proceed to Exercise 7-8. Alternately, you can copy the stop functionality into your project at the beginning of Exercise 7-5.

End of Exercise 7-4

Exercise 7-5 Integrate Error Module

Goal

Integrate an error handling module into the design pattern.

Scenario

When working on large applications, it is often helpful to distribute the development effort for the project among several developers. Another developer received a copy of your code after you integrated the Play functionality. That developer has integrated the Save, Load, and Stop functions into `TLC Main.vi`. Before you can integrate your error handling module into the design pattern, you must overwrite your working copy of the TLC project with their files.

Note Refer to Exercise 7-7 and Exercise 7-8 for more information about the changes the developer made. If you want to make these changes yourself, complete Exercise 7-7 and Exercise 7-8 before you begin Exercise 7-5.

The Error module that you built is designed to safely shutdown the application if an error occurs. Shutting down the producer loop requires sending a user event to the producer loop. Shutting down the consumer loop requires placing an Exit message in the queue. Shutting down the display loop requires placing a Stop message in the queue.

When designing and developing an application, be mindful of how the application should respond when an error occurs. With the Theatre Light Controller, an error should cause the system to gracefully shut down by executing the cases in each of the application loops that cause the loops to stop.

Design

Update the TLC project to the current version and modify the TLC Main VI to call the Error module after each computation in the producer, consumer, and display loop.

Implementation

1. Copy the updated version of the TLC project and overwrite your working copy of the code.

Note If you have completed self-study Exercises 7-7 and 7-8, skip to step 3 of this exercise.

❑ Close the TLC project and the TLC Main VI.

- ❑ Navigate to `<Exercises>\LabVIEW Core 3\Stop-Save-Load\` and copy `TLC.lvproj` and `TLC Main.vi` to the clipboard.
- ❑ Navigate to `<Exercises>\LabVIEW Core 3\Course Project\` and paste `TLC.lvproj` and `TLC Main.vi`. When prompted, overwrite the existing files.
- ❑ Open the TLC project and the TLC Main VI.

2. Test the new code.
   - ❑ Run the TLC Main VI.
   - ❑ Record a cue and play it. Verify the stop functionality by clicking the **Stop** button during play.
   - ❑ Select **File»Save** and save the data file as `File Test.dat` in the `<Exercises>\LabVIEW Core 3` directory.
   - ❑ Stop the VI by selecting **File»Exit**.
   - ❑ Run the TLC Main VI.
   - ❑ Load `File Test.dat` by selecting **File»Open** and navigating to the `<Exercises>\LabVIEW Core 3` directory.
   - ❑ Play the loaded cue list and verify that the cues match the saved cues.
   - ❑ Select **File»Exit** to stop the VI.
3. Modify the TLC Main VI to call the Error module after each computation in the producer, consumer, and display loop, as shown in Figure 7-20. Use the following items:
   - ❑ `tlc_Error Module.vi`—Add an Error module to each loop to handle any errors that occur and stop all loops in the event of an error. Add another instance after the Destroy User Event function.
   - ❑ Set the **command** of the last Error module to **Report Errors** because this instance returns any errors that occurred.
   - ❑ Simple Error Handler VI—If an error occurs, this VI displays a dialog box that returns a description of the error.
4. Save the VI.

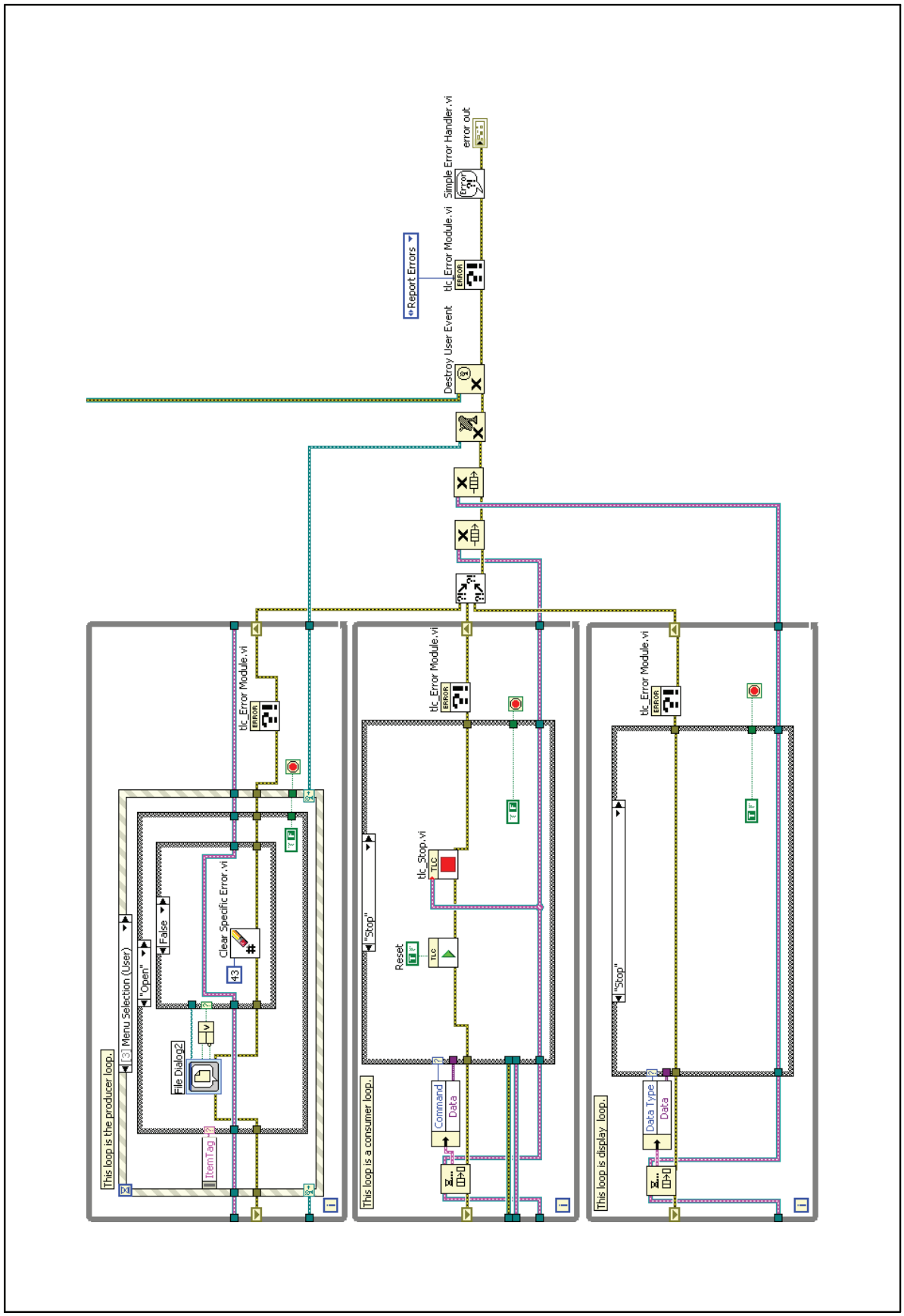


Figure 7-20. Producer/Consumer Design Pattern with Error Module

Testing

1. Test the error handling capability by placing a Diagram Disable structure over the Clear Specific Error VI in the producer loop.

   You can use the Diagram Disable structure to comment out code when you want to test functionality or isolate problem areas.

   ❑ In the producer loop, navigate to the False case in the Open case of the Menu Selection User Event structure, as shown in Figure 7-21.

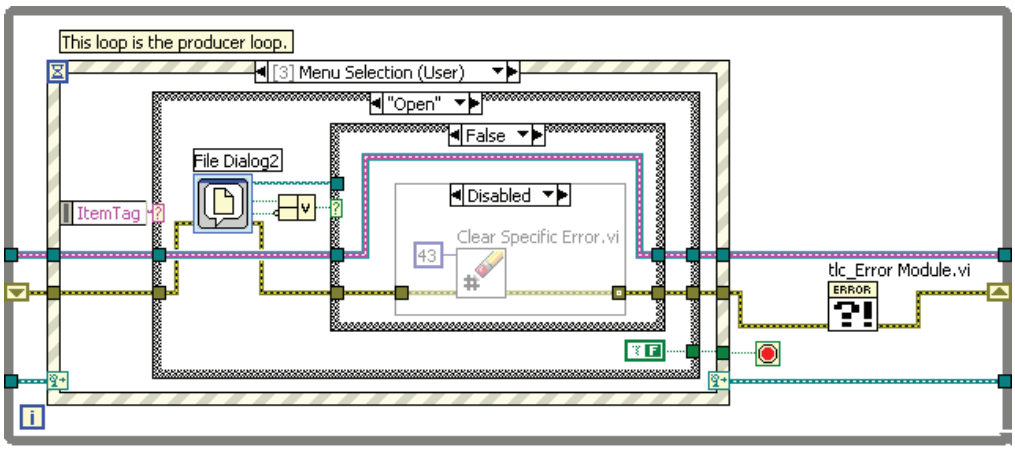


Figure 7-21. Producer Loop with Diagram Disable Structure

   ❑ Add the Diagram Disable structure from the **Structures** palette to enclose the Clear Specific Error VI.

   ❑ Switch to the Enabled case in the Diagram Disable structure, and wire the error cluster through the case as shown in Figure 7-22.

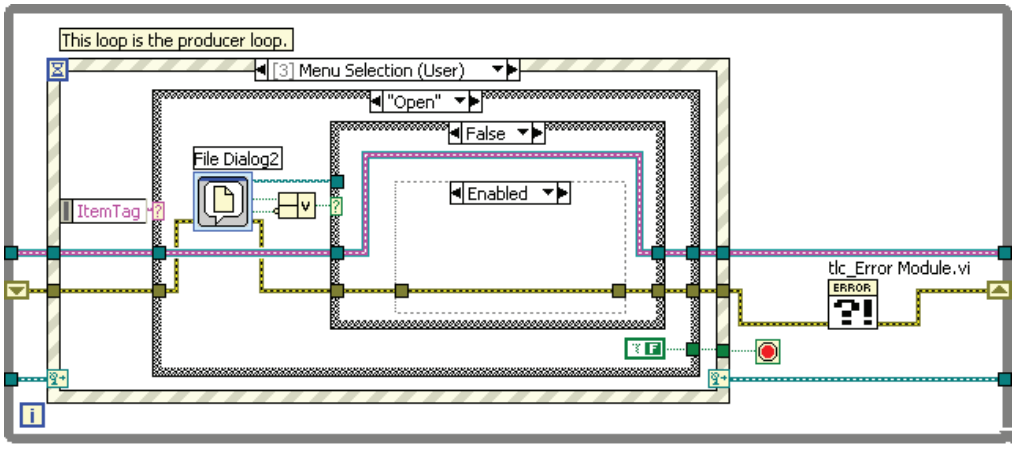


Figure 7-22. Enabled Case of Producer Loop with Diagram Disable Structure

2. Save the VI.

3. Run the VI.
   - ❑ Select **File»Open** from the menu.
   - ❑ Click **Cancel** in the **File** dialog box.
   - ❑ If the error module works correctly, the Simple Error Handler VI displays a dialog box indicating that Error 43 occurred, and the application stops because the error module stops each loop in the application.
4. Enable the case of the Diagram Disable structure that contains the Clear Specific Error VI.
   - ❑ Switch to the block diagram and navigate to the Disabled case of the Diagram Disable structure.
   - ❑ Right-click the border of the Diagram Disable structure and select **Enable This Subdiagram**.
5. Save the VI and the project.

Challenge

Integrate the Select Cue function to update the user interface with the values in a selected recorded cue. There are two primary tasks that must be implemented to build the Select Cue functionality. The Select Cue function must acquire the cue values for the selected cue and update the front panel with the cue values.

1. Modify `tlc_Functions.ctl` to contain a `Select Cue` item.
2. Change the Cue List indicator to a control.
3. Modify the Event structure in the producer loop to contain a `"Cue List":Value Change` event case.
4. Modify the producer loop to pass the index of the selected cue to the consumer loop.
5. Modify the consumer loop to get the selected cue value and update the front panel.
6. To test your changes, record several cues. Click each row in the cue list. The Cue Information cluster should update with the information for each cue that you select.

End of Exercise 7-5

Exercise 7-6 Stress and Load Testing

Goal

Perform stress and load testing on the application.

Scenario

Before releasing an application, you must perform a set of system level tests. These tests can consist of usability testing, performance testing, stress, and load testing.

Design

Create a large set of cues stored in a file that contain random wait times, fade times, follow times, channel colors, and intensities. The VI that creates the large set of cues requests maximum values to use for the wait, fade, and follow times so that it is easier to analyze the functionality of the application.

Implementation

1. Open `System Testing Driver.vi` from the `<Exercises>\LabVIEW Core 3\Course Project\System Testing` directory.
2. Open the block diagram and examine how this VI uses the modules that you implemented to create a large set of Cues.
3. Set the following front panel controls:
   - **Number of Cues** = `2000`
   - **Maximum Wait Time** = `0`
   - **Maximum Fade Time** = `5`
   - **Maximum Follow Time** = `0`
4. Run the VI.
5. When prompted, save the file as `Stress Load Test.dat` in the `<Exercises>\LabVIEW Core 3\Course Project\System Testing` folder.
6. Run the TLC Main VI.
7. Select **File»Open** to load `Stress Load Test.dat` from the `<Exercises>\LabVIEW Core 3\Course Project\System Testing` folder.

8. Click **Play**.

9. Open the Windows Task Manager and monitor the memory usage and performance of the Theatre Light Controller. Running TLC Main VI with a large number of cues can indicate if there is a memory or performance issue with the application.

10. Stop the TLC Main VI.

11. Select **File»Exit** to stop the VI.

End of Exercise 7-6

Exercise 7-7 Self Study: Integrate Save and Load Functions

Goal

Use a scalable data type to pass data from the consumer to the producer.

Scenario

The file save and load functionality are important for the lighting designers who use the software. Most theatres orchestrate the lighting for a production during dress rehearsals and the lighting designer uses the same lighting cues from dress rehearsal for opening night and beyond. Save and load functionality is important for any large scale application.

Design

Modify the producer loop in the TLC Main VI to prompt the user for the file name. Several checks need to occur to determine if the user cancels the file operation or tries to open a file that does not exist.

Modify the Save case in the consumer loop to get the values stored in the cue module and pass the recorded Cues to the file module.

Modify the Load case in the consumer loop to open the file and extract the saved cues, while populating the cue module with the saved cues.

Implementation

Note The code changes described in this exercise are integrated into the TLC Main VI as step 1 of Exercise 7-5. The purpose of this exercise is to describe the changes that were made to integrate the save and load functions into the application. If you are working through all the Lesson 7 exercises, you should complete this exercise after Exercise 7-3.

Modify the Save case in the consumer loop to get the values stored in the cue module and pass the recorded Cues to the file module.

Save

1. Add the shared files and **Shared** virtual folder to the project.

   - ❑ Right-click **My Computer** in the project tree and select **Add» Folder (Snapshot)** from the shortcut menu.

   - ❑ Navigate to the `<Exercises>\LabVIEW Core 3\Course Project\Shared` directory and click **Current Folder** to add the folder and its contents to the project tree.

2. Open the TLC Main VI.

3. Modify the Save case in the Menu Selection event of the producer loop, as shown in Figure 7-23. Use the following items:

   - ❑ File Dialog Express VI
     - – Launches a dialog box to select a file to load. Verify that **Limit selection to single item** is selected. Select **File** and **New**. Click **OK**.
     - – If the user selects an existing file, the File Dialog Express VI prompts the user to replace the file. The File Dialog Express VI returns True for the **exists** output if the user replaced the file. If the user cancels a save operation, the **cancelled** output returns True.
     - – Right-click the File Dialog Express VI and select **View As Icon**.
   - ❑ Compound Arithmetic function
     - – Set the Mode to OR.
     - – Invert the bottom terminal.

       The Compound Arithmetic function returns True if the user replaces the file or does not cancel the file save operation.
   - ❑ Case structure
   - ❑ Enqueue Element function—Add this function to the True case of the Case structure.
   - ❑ `tlc_Consumer Control.ctl` constant—Add this constant to the True case and set it to **Save**.
   - ❑ To Variant function—Wire the **Path** output of the File Dialog Express VI to the To Variant function.
   - ❑ Bundle By Name function—Wire the `tlc_Consumer Control.ctl` constant to the Bundle By Name function. Wire the output of the To Variant function to the Bundle By Name function.

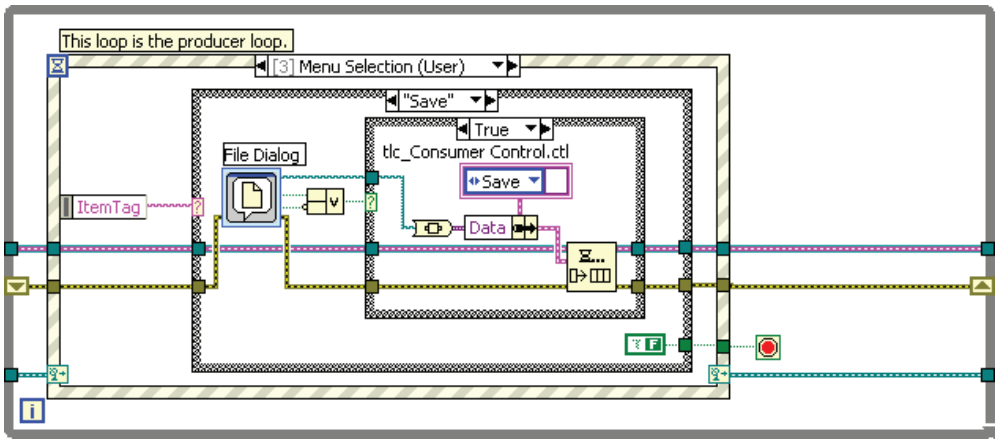


Figure 7-23. Producer Save Case True Case

- ❑ Clear Specific Error VI
  - – Add this VI to the False case of the Case structure.
  - – Create a numeric constant with the value of `43` and wire it to the **code** input of the Clear Specific Error VI. The File Dialog Express VI generates Error 43 if the user selects **Cancel** in the **File** dialog box.
- ❑ Wire the queue reference through the False case.

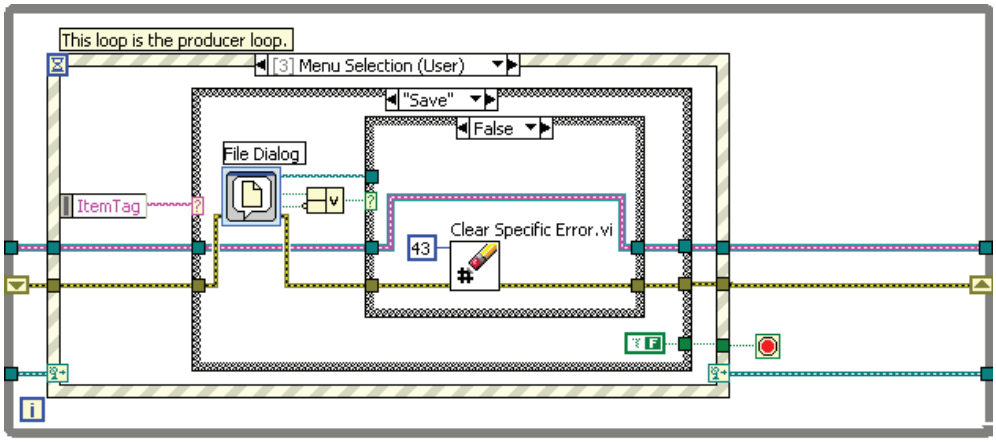


Figure 7-24. Producer Save Case False Case

4. Save the VI.
5. Modify the Save Consumer case to perform the Save function as shown in Figure 7-25. Use the following items:
   - ❑ Variant to Data function

- ❑ For Loop—This loop iterates through the cues in the Cue List.
- ❑ `tlc_Cue Module.vi`
  - – Add the first instance of this VI outside the For Loop and set the **command** input to **Get Number of Cues**.
  - – Wire the **Number of cues** output to the count terminal of the For Loop to determine the number of iterations needed to save the Cue List data.
- ❑ `tlc_Cue Module.vi`
  - – Add the second instance inside the For Loop and set the **command** input to **Get Cue Values**.
  - – Wire the iteration terminal to the **Cue Index** input of the cue module to retrieve each cue in the list.
  - – Replace the For Loop error tunnels with shift registers.
- ❑ `tlc_ File Module.vi`—Set the **command** input to **Save Cues**.
- ❑ Wire the **Cue Output** output of the cue module to the **Cue Array Input** of the file module. Verify that indexing is enabled on the For Loop tunnel.

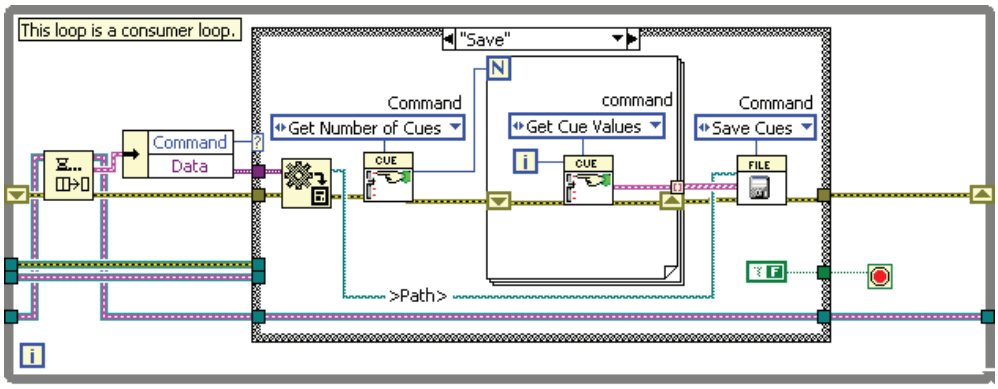


Figure 7-25. Consumer Save Case

6. Save the VI.

Load

Modify the Load case in the consumer loop to open the file, extract the saved cues, and populate the cue module with the saved cues.

1. Create the Open case in the Menu Selection (User) event case of the producer loop, as shown in Figure 7-26. It is easiest to create the Open

case by duplicating the Save case and modifying the File Dialog Express VI.

- ❑ Delete the existing Open case of the Case structure inside the Menu Selection (User) event case.
- ❑ Right-click the Save case and select **Duplicate Case** from the shortcut menu.
- ❑ Enter `Open` in the case selector label.
- ❑ Double-click the File Dialog2 Express VI to configure it. Select **File** and **Existing**. Click **OK**.
- ❑ In the True case, change the `tlc_Consumer Control.ctl` enum constant to **Load**.

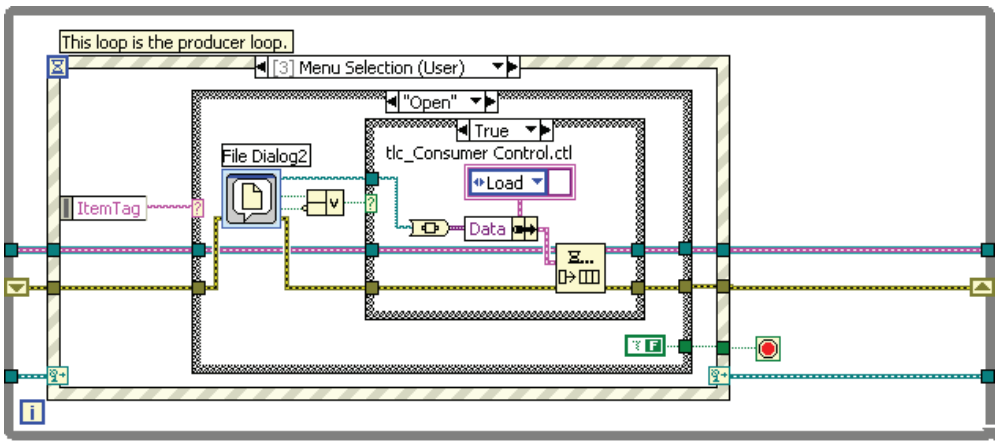


Figure 7-26. Producer Open Case True Case

2. Build the load functionality into the consumer loop.
   - ❑ Open the tlc_Load VI from the **Integration** virtual folder and examine how the VI operates. First, the VI loads the cues from the specified file. Then it initializes the front panel and adds the cues from the file to the cue module. Next, it updates the cue list and deselects any selection in the cue list.
   - ❑ Close the VI.
3. Complete the Load case of the consumer loop in the TLC Main VI, as shown in Figure 7-27. Use the following items:
   - ❑ Add the Variant to Data function to the Load case.
   - ❑ Add the tlc_Load VI to the Load case.

- ❑ Wire the **data** output of the Variant to Data function to the **Path** input of the tlc_Load VI.
- ❑ Right-click the **type** input of the Variant to Data function and select **Create»Constant** to create the path constant.

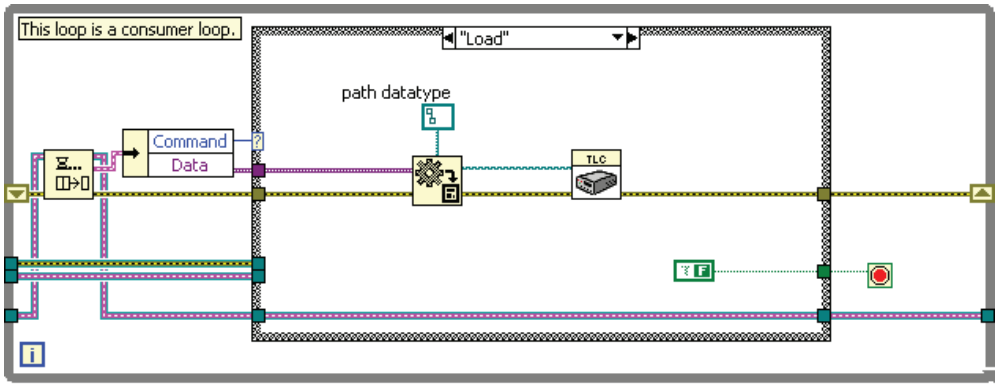


Figure 7-27. Load Function

4. Save the VI.

Testing

1. Run the TLC Main VI.
2. Record several cues.
3. Select **File»Save** from the menu.
4. Save the data file as `File Test.dat` in the `<Exercises>\LabVIEW Core 3` directory.
5. Select **File»Exit** to stop the VI.
6. Run the TLC Main VI.
7. Load `File Test.dat` by selecting **File»Open** and navigating to the `<Exercises>\LabVIEW Core 3` directory.
8. Select **File»Exit** to stop the VI.

Note If you are working through all the Lesson 7 exercises, proceed to Exercise 7-4.

End of Exercise 7-7

Exercise 7-8 Self Study: Integrate Stop Function

Goal

Flush a queue to guarantee when a command is sent.

Scenario

The requirements for the application state that the user can stop a playing cue by clicking the **Stop** button. This exercise implements that capability.

Design

When the user clicks the **Stop** button, the Event structure generates an event to process the **Stop** button. To guarantee that the application responds to the **Stop** button, flush the queue that controls the consumer loop to remove any messages stored in the queue.

Make the following modifications to the TLC Main VI:

- In the producer loop, modify the Stop event case to flush the queue and call the tlc_Timing Stop Module VIs.
- In the consumer loop, modify the Stop case to enable the front panel controls.

Implementation

Note The code changes described in this exercise are integrated into the TLC Main VI as step 1 of Exercise 7-5. This exercise describes the changes that were made to integrate the stop function into the application. If you are working through all the Lesson 7 exercises, you should complete this exercise after Exercise 7-4.

1. Open the TLC Main VI.
2. In the producer loop, add the Flush Queue function to the Stop event case to flush the queue, as shown in Figure 7-28.

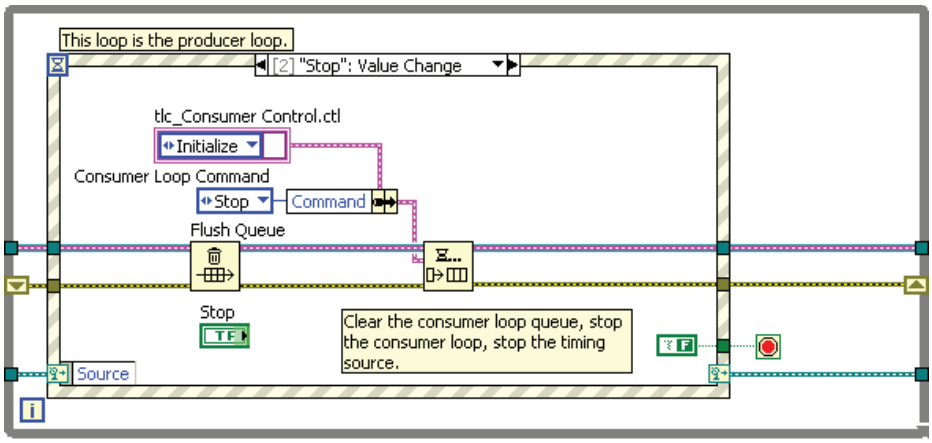


Figure 7-28. Producer Stop Event Case

3. Open `tlc_Stop.vi` from the **Integration** virtual folder and examine how the VI operates.

   - ❑ This VI clears any play commands that remain in the consumer queue after the Stop case executes.
   - ❑ Because the consumer loop runs in parallel with the producer loop, it is possible that the Play case of the consumer loop could enqueue a play command after the user clicks **Stop** and the queue is flushed in the producer loop.

     To address this, the tlc_Stop VI previews the next queue element without removing it. If the next element is Play, it removes the element from the queue. If not, execution continues normally.
   - ❑ Close the tlc_Stop VI.

4. In the consumer loop of the TLC Main VI, modify the Stop case as shown in Figure 7-29. This case should stop execution of the cue list and enable the user interface controls. Use the following items:

   - ❑ `tlc_Stop.vi`—This VI previews the next element in the consumer queue. If the next element is a play command, then it removes that element so the cue list execution stops completely.
   - ❑ `tlc_Play.vi`—Create the Reset True constant from the **Reset** input of the tlc_Play VI. This causes the tlc_Play VI to behave as if the cue list had completed execution. It resets all of its functional global variables and enables the front panel controls.

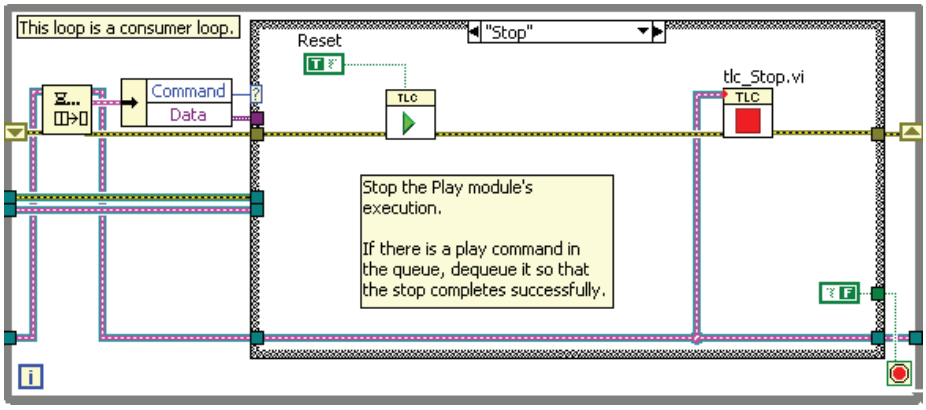


Figure 7-29. Consumer Stop Case

5. Save the VI.

Testing

1. Run the TLC Main VI.
2. Record a cue, and play the cue. Verify that the stop functionality works by clicking the **Stop** button during play.
3. Select **File»Exit** to stop the VI.

Note If you are working through all the Lesson 7 exercises, proceed to Exercise 7-5.

End of Exercise 7-8

Notes

8

Evaluating VI Performance

Exercise 8-1 Identify VI Issues with VI Metrics

Goal

Determine the complexity of the VI.

Scenario

Use VI Metrics to determine the complexity of a VI and identify issues with the application.

Design

Run the VI Metrics tool to determine the complexity of the VI.

Implementation

1. Open the TLC Main VI.
2. Select **Tools»Profile»VI Metrics**.
3. In the **Show statistics for** section, place a checkmark in each checkbox.
   - **Diagram**
   - **User interface**
   - **Globals/locals**
   - **CINs/shared lib calls**
   - **SubVI interface**
4. Examine the metrics for the VI.
   - ❑ Notice that the modular approach to developing the application results in a low **max diag depth**. This statistic indicates the deepest nesting level of block diagrams in a VI. If the VI has no structures, it has a depth of 0.
   - ❑ Notice the number of global and local variables that this application uses.
5. When you finish viewing the metrics, click **Done**.

End of Exercise 8-1

Exercise 8-2 Methods of Updating Indicators

Goal

Compare the performance of different methods for updating indicators.

Description

There are three primary methods for updating user interface values:

- Wire data directly to an indicator—Fastest and preferred method of passing data to the user interface.
- Wire data to a local variable—Good method for initializing data that is in a control.
- Wire data to a Value Property Node of the indicator—Use this method to update the control or indicator directly on the block diagram or in a subVI through a control reference.

Each of these methods has performance differences. This exercise demonstrates the performance of each of these methods.

Implementation

1. Open `Methods of Updating Indicators.vi` from the `<Exercises>\LabVIEW Core 3\Methods of Updating Indicators` directory.
2. Open the block diagram and examine how this VI operates.
3. Run the VI for each of the methods by setting the **Method** enum, and running the VI. Observe how long the VI takes to run for each method.

End of Exercise 8-2

Notes

Notes

9

Implementing Documentation

Exercise 9-1 Document User Interface

Goal

Document the user interface.

Scenario

Document the user interface. You must document the front panel of every VI that you create.

Design

Document the TLC Main VI. Good documentation includes a description of the VI, and a meaningful description and tip for each control and indicator.

Implementation

1. Open the TLC Main VI.
2. Select **File»VI Properties** and select **Documentation** from the **Category** list. Add the following documentation to the **VI description** section:
   - ❑ An overview of the VI.
   - ❑ Instructions for how to use the VI.
   - ❑ Click **OK** to close the **Documentation** page.
3. Include a description for every control and indicator. Right-click the object and select **Description and Tip** from the shortcut menu to create, edit, and view object descriptions. The object descriptions appear in the **Context Help** window when you idle the cursor over the object and in any VI documentation you create.

   Every control and indicator needs a description that includes the following information:
   - ❑ Functionality
   - ❑ Data type

- ❑ Valid range (for inputs)
- ❑ Default value (for inputs)

  You also can list the default value in parentheses as part of the control or indicator label.

- ❑ Behavior for special values (0, empty array, empty string, and so on)

4. Save the VI.

End of Exercise 9-1

Exercise 9-2 Implement Documentation

Goal

Use features in LabVIEW to create professional documentation for the application.

Scenario

Documentation is an important part of developing applications that are scalable, readable, and maintainable. Also, the end user of the application requires documentation in order to use the system. LabVIEW assists in developing documentation. LabVIEW can automatically generate HTML or RTF documents that document the functionality of the application. After LabVIEW generates the documentation, the developer can edit and expand the documentation for other developers who maintain the application, and for the end user.

Design

Use the **Print** dialog box to generate an HTML document that you can link to the VI Properties of the TLC Main VI.

Implementation

1. Open the TLC Main VI.
2. Select **File»Print** to open the **Print** dialog box.
3. Select **TLC Main.vi** and click **Next**.
4. Select **VI Documentation** and click **Next**.
5. Select **Complete** for the **VI Documentation Style** and click **Next**.
6. Select **HTML file** for the **Destination** and click **Next**.
7. Click **Save**.
8. Save the documentation as `TLC_Main.html` in the `<Exercises>\LabVIEW Core 3\Course Project\Documentation` directory.
9. Select **File»VI Properties** and select **Documentation** from the **Category** pull-down menu.
10. Click **Browse**, navigate to `<Exercises>\LabVIEW Core 3\Course Project\Documentation\TLC_Main.html` and click **OK** to add the path to the **Help Path** textbox. This loads the help HTML file if the user clicks **Detailed Help** from the **Context Help** window.

11. Click **OK** to close the **VI Properties** dialog box.

12. Save the VI.

Testing

1. Open the **Context Help** window.

2. Idle your mouse over the icon/connector pane of the TLC Main VI.

3. Click the **Detailed help** link in the **Context Help** window to load the documentation for the application.

End of Exercise 9-2

Notes

Notes

10

Deploying the Application

Exercise 10-1 Implementing Code for Stand-Alone Applications

Goal

Create an **About** dialog box that you can use in your own applications.

Scenario

Most applications have an **About** dialog box that displays general information about the application and the user or company that designed it. You can create a VI that LabVIEW runs when a user selects **Help»About** to display information about the stand-alone application you create.

When creating a stand-alone application, it is important to understand the architecture of the Application Builder. A VI that is running as a stand-alone executable remains in memory when the application finishes running. It is necessary to call the Quit LabVIEW function in order to close the application when the application finishes executing. Adding the Quit LabVIEW function to the block diagram can make editing the application more difficult in the future because LabVIEW quits each time the application finishes.

You can use a Case structure and the Application:Kind property to execute different code based on the LabVIEW instance the VI is running in, including invalid application types. In this case, you want to call the Quit LabVIEW function if the application is running in the LabVIEW Run-Time Engine.

For stand-alone applications in LabVIEW, set **Window Appearance** to **Top-level application** so that the front panel opens when the VI runs.

Design

1. Implement a dialog box VI that uses an Event structure to create an **About** dialog box.
   - Create a dialog box VI that closes when the user clicks the mouse anywhere on the VI.
   - Modify the run-time menu to add the **Help»About** menu item.

2. Add a Case structure around the Quit LabVIEW function and use an App.Kind Property Node to quit LabVIEW when the code is running in the Run-Time Engine.

3. Modify the properties of the VI to prepare for building a stand-alone application.

Implementation

About Dialog Box

1. Create a dialog box VI that closes when the user clicks the mouse button anywhere on the VI.

   ❑ Create a new VI within the TLC project.

   ❑ Select **File»VI Properties** and set the **Window Appearance** of the VI to **Dialog**. Click **OK**.

2. Create a front panel for the VI.

Note The front panel window must include a National Instruments copyright notice. Refer to the *National Instruments Software License Agreement* located on the LabVIEW DVD or CD for more information about the requirements for any **About** dialog box you create for a LabVIEW application.

   ❑ Add any free text or images you want to the front panel.

   ❑ Add error clusters to the front panel.

3. Switch to the block diagram and add code to close the dialog box when the user clicks the mouse button anywhere on the VI, as shown in Figure 10-1.

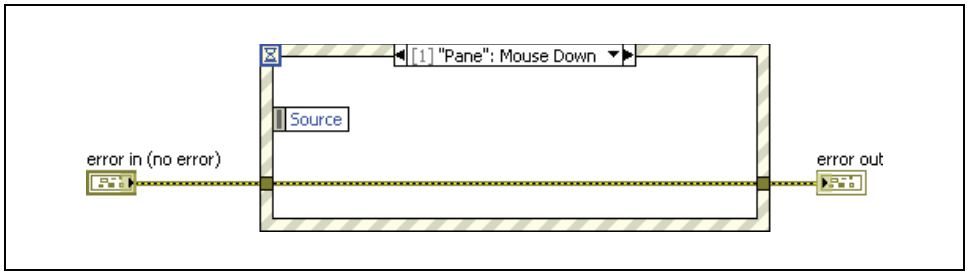


Figure 10-1. About VI Block Diagram

   ❑ Add an Event structure to the block diagram and edit an Event case for the **Mouse Down** event for the **<Pane>** event source.

   ❑ Wire the error clusters through the Event structure.

4. Save the VI as `About.vi` in the `<Exercises>\LabVIEW Core 3\Course Project` directory.

Note You must save the About VI at the same level of the file hierarchy as the top-level VI for the **About** dialog box to display when the user selects **Help»About**.

5. Close the About VI.

6. Open the TLC project. Verify that the About VI displays in the project hierarchy at the same level as the TLC Main VI.

7. Save the project.

8. Modify the TLC Main VI so the run-time menu contains **Help»About LabVIEW**. When the user selects **Help»About** in the stand-alone application, the **About** dialog box displays. Figure 10-2 shows the completed Run-Time Menu Editor.

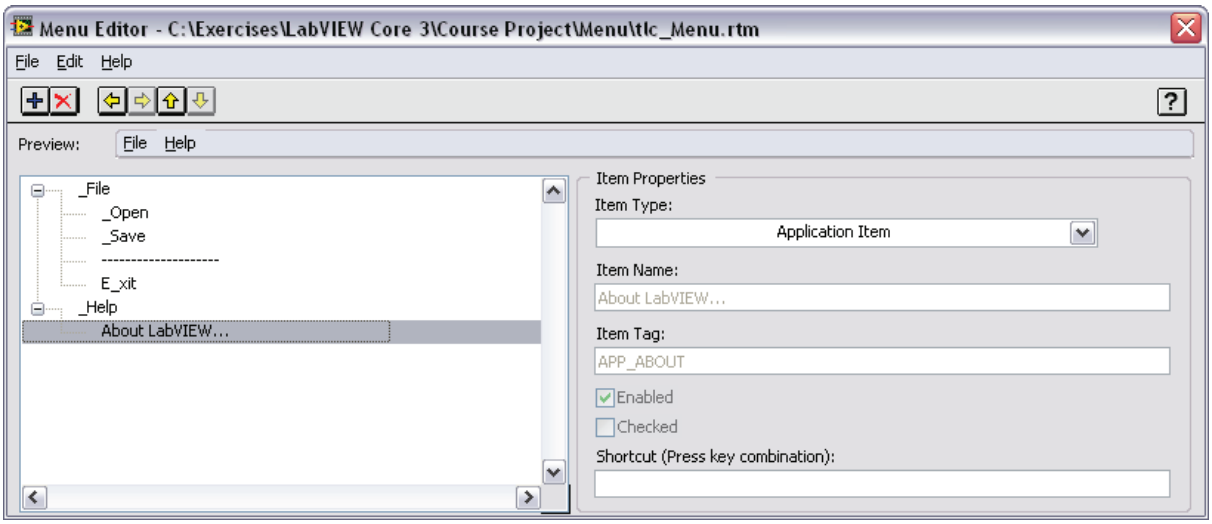


Figure 10-2. Run-Time Menu Editor

- ❑ Open the TLC Main VI. Select **Edit»Run-Time Menu** to open the Menu Editor.
- ❑ Add the Help menu item by adding a User Item for Help, with the **Item Name** set to `_Help` and the **Item Tag** set to `Help`. Use the arrow buttons to move the entry below the other entries and in line with **File**, as shown in Figure 10-2.
- ❑ Add the About LabVIEW item below the Help menu item by selecting **Edit»Insert Application Item»Help»About LabVIEW**.

9. Save the Run-Time Menu and exit the Menu Editor.

Quit LabVIEW Function

Use the Quit LabVIEW function to shutdown LabVIEW when the application completes. You can use the Application:Kind property with a Case structure to execute the Quit LabVIEW function only when the application runs in the Run-Time Engine, for example, when it runs as a stand-alone application. Implement this functionality as shown in Figure 10-3.

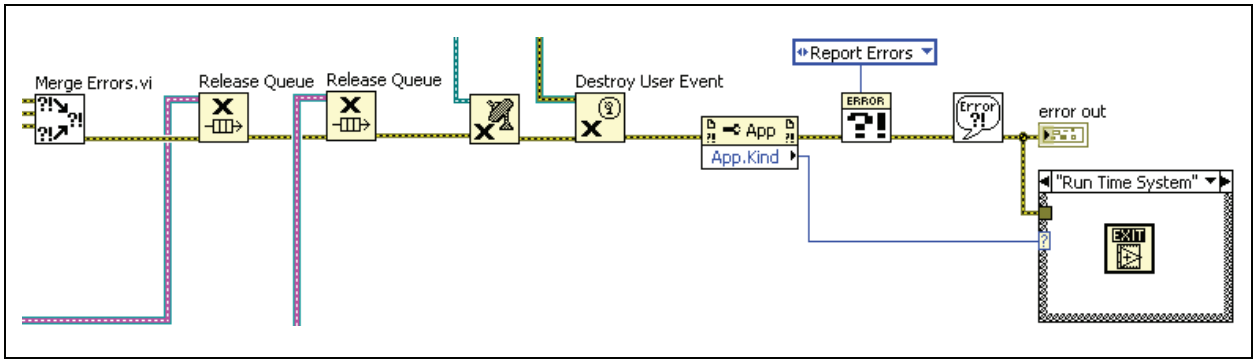


Figure 10-3. Quit LabVIEW

1. Create code to quit LabVIEW if the VI is running in the LabVIEW Run-Time Engine, for example, if the VI is called as part of a shared library or as a stand-alone executable.

   ❑ Add a Property Node to the block diagram of the TLC Main VI.

   ❑ Configure the Property Node to return the value of the **Application» Kind** property.

   ❑ Add a Case structure to the block diagram.

   ❑ Wire the **App.Kind** output of the Property Node to the case selector terminal of the Case structure.

   ❑ Right-click the Case structure and select **Add Case for Every Value**. Each case represents a different LabVIEW execution environment.

   ❑ Add the Quit LabVIEW function to the Run-Time System case.

2. Wire the diagram as shown in Figure 10-3. The Quit LabVIEW function must be the last function that executes.

Window Appearance

1. Select **File»VI Properties** and select **Window Appearance** from the **Category** list.
2. Select **Top-level application window** and click **OK**.
3. Enter a meaningful name for the VI in **Window title** or place a checkmark in the **Same as VI name** checkbox.
4. Save the VI.

End of Exercise 10-1

Exercise 10-2 Create a Stand-Alone Application

Goal

Create a build specification and build a stand-alone application (EXE) in LabVIEW.

Scenario

Creating a stand-alone application is important for distributing and deploying your application. It is also a step in the creation of a professional installer.

Design

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

Implementation

1. Create a stand-alone application of the Theatre Light Controller.
   - ❑ Open the `TLC.lvproj`.
   - ❑ Right click **Build Specifications** and select **New»Application (EXE)** from the shortcut menu to open the **Application Properties** dialog box.
   - ❑ On the **Information** page, specify a **Build specification name** and **Target filename**.
   - ❑ Set the **Destination directory** to `<Exercises>\LabVIEW Core 3\Course Project\Builds\Executable`.
   - ❑ Select the **Source Files** page and select `TLC Main.vi` in the **Project Files** listbox. Click the right arrow to add the VI to the **Startup VIs** listbox.
   - ❑ Select `About.vi` in the project hierarchy and click the right arrow to add the VI to the **Always Included** listbox.
   - ❑ Select the **Icon** page and use the Icon Editor to create a color icon for both the 32 × 32 and 16 × 16 icons. Save the new icon in the `<Exercises>\LabVIEW Core 3\Course Project\Icons` directory.
   - ❑ Deselect **Use the default LabVIEW Icon file** and browse to the `.ico` file you created.

- ❑ Select the **Preview** page and click **Generate Preview** to preview the output of the Build Specification.
- ❑ Select the **Advanced** page and add a checkmark to the **Copy error code files** checkbox to include the custom error code that you developed.
- ❑ Click **OK**.

2. Save the project.
3. Right-click the build specification in the project tree and select **Build** from the shortcut menu.

Testing

Navigate to the directory you specified for the destination directory and run the executable.

End of Exercise 10-2

Exercise 10-3 Self-Study: Create an Installer

Goal

Create a professional installer for your application.

Scenario

A professional application should always have an installer to deploy the application. Providing an installer improves the end user experience with the application.

Design

Create an installer build specification for the executable you created.

Implementation

Create an installer for the Theatre Light Controller.

1. Open the TLC project.
2. Right-click **Build Specifications** and select **New»Installer** from the shortcut menu to open the **My Installer Properties** dialog box.
3. On the **Product Information** page, specify a **Build specification name** and **Product name**.
4. Set the **Installer destination** to `<Exercises>\LabVIEW Core 3\Course Project\Builds\Installer`.
5. Select the **Source Files** page and verify that a folder that matches your project name exists under **ProgramFilesFolder** in **Destination View**. If no folder exists for your project, add a folder and provide a meaningful name for the folder.
6. In the **Project View** list, select the stand-alone application build specification. Click the arrow to add the build specification to the destination folder.
7. Select the **Shortcuts** page to edit the shortcut the installer creates in the **ProgramMenuFolder** in Windows. Change the **Name** to `Theatre Light Controller`. This places the item in the **Start»Programs** menu.
8. Select the **Additional Installers** page and verify that a checkmark appears in the checkbox for the LabVIEW Run-Time Engine installer.
9. Click **OK**.

10. Save the project.

11. Right-click the installer build specification and select **Build** from the shortcut menu.

Testing

Navigate to the installer destination directory you specified and run the installer. After the installer runs, verify the installation of the Theatre Light Controller.

Note These are the ideal conditions for testing an installer. For a more accurate test, try your installer on a computer that does not have LabVIEW installed.

End of Exercise 10-3

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 on demand training modules via the Services Resource Center. 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`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. 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

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 zone.ni.com/devzone/cda/tut/p/id/5072 for more information.

Course Evaluation

Course _____

Location _____

Instructor _____ Date _____

Student Information (optional)

Name _____

Company _____ Phone _____

Instructor

Please evaluate the instructor by checking the appropriate circle.	Unsatisfactory	Poor	Satisfactory	Good	Excellent
Instructor's ability to communicate course concepts	❍	❍	❍	❍	❍
Instructor's knowledge of the subject matter	❍	❍	❍	❍	❍
Instructor's presentation skills	❍	❍	❍	❍	❍
Instructor's sensitivity to class needs	❍	❍	❍	❍	❍
Instructor's preparation for the class	❍	❍	❍	❍	❍

Course

Training facility quality	❍	❍	❍	❍	❍
Training equipment quality	❍	❍	❍	❍	❍

Was the hardware set up correctly? ❍ Yes ❍ No

The course length was ❍ Too long ❍ Just right ❍ Too short

The detail of topics covered in the course was ❍ Too much ❍ Just right ❍ Not enough

The course material was clear and easy to follow. ❍ Yes ❍ No ❍ Sometimes

Did the course cover material as advertised? ❍ Yes ❍ No

I had the skills or knowledge I needed to attend this course. ❍ Yes ❍ No If no, how could you have been better prepared for the course? _____

_____

What were the strong points of the course? _____

_____

What topics would you add to the course? _____

_____

What part(s) of the course need to be condensed or removed? _____

_____

What needs to be added to the course to make it better? _____

_____

How did you benefit from taking this course? _____

_____

Are there others at your company who have training needs? Please list. _____

_____

_____

Do you have other training needs that we could assist you with? _____

_____

How did you hear about this course? ❍ NI Web site ❍ NI Sales Representative ❍ Mailing ❍ Co-worker ❍ Other _____