Advanced Architectures in LabVIEW™ Exercises

Course Software Version 2011
February 2012 Edition
Part Number 325183B-01

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

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

National Instruments Corporate Headquarters

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

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

Contents

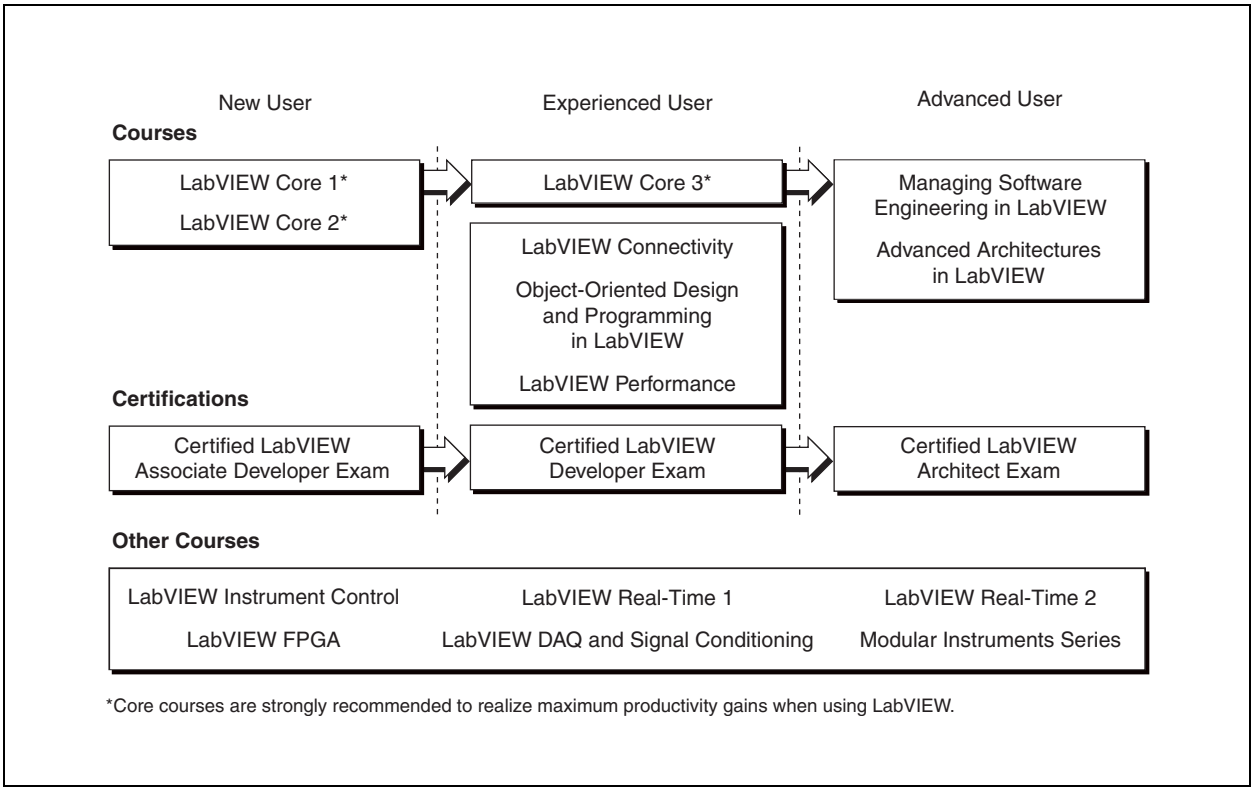
Student Guide

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

A. NI Certification

The *Advanced Architectures in LabVIEW* 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

In the *Advanced Architectures in LabVIEW* course you participate in discussions and work independently and collaboratively to learn how to architect an application and then design the components to support the architecture.

This course assumes you have taken the *LabVIEW Core 3* course or have equivalent experience.

The course is divided into lessons, each covering a topic or a set of topics. Each lesson consists of the following parts:

- An introduction that describes what you will learn.
- A discussion of the topics.
- A set of exercises that reinforces the topics presented in the discussion.

  Some lessons include optional exercises or challenge steps to complete if time permits.
- A summary that outlines important concepts and skills taught in the lesson.

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

C. What You Need to Get Started

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

- ❑ Computer running Windows 7/Vista/XP
- ❑ NI LabVIEW 2011
- ❑ Advanced Architectures in LabVIEW CD, which contains the following files:

Directory	Description
`Exercises`	Folder for saving VIs created during the course and for completing certain course exercises
`Solutions`	Folder containing the solutions to all the course exercises
`Advanced Architectures in LabVIEW 2011 - Course Manual.pdf`	*Advanced Architectures in LabVIEW* Course manual.

D. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer.
2. Install the Exercises and Solutions files to the desired location.

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

E. Course Goals

After completing this course you will be able to:

- Design a software architecture to be implemented in LabVIEW
- Design a consistent, organized, and usable API
- Analyze and evaluate several solutions to a problem
- Use advanced design patterns and techniques to build the components or subsystems of an architecture
- Understand the design trade-offs when selecting an advanced design pattern or technique
- Analyze, critique, and improve the architecture of a LabVIEW application

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 **Options»Settings»General** directs you to pull down the **Options** menu, select the **Settings** item, and select **General** 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.

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.

1

Software Architecture – Introduction

Exercise 1-1 Design a Wind Farm

Goal

To draft the initial software architecture for the Wind Farm course project.

Scenario

LabVIEW is a programming language that facilitates the rapid prototyping and development of code. As such many users of LabVIEW have developed a “code and fix” model for constructing software. The proper process, for smaller as well as larger projects, is to first design a software architecture. The developer should specify the major components of the project and how those components communicate with one another. Data structures should also be defined.

This lesson provides you the opportunity to begin with a requirements document and sketch out a preliminary software architecture. You will have the opportunity to compare your architecture with your fellow students. You will also explore several implementations of the key components of the Wind Farm.

Additionally, this lesson allows you to experience a process that is similar to what you will encounter if you take the Certified LabVIEW Architect Exam.

Implementation

Refer to the next section, *Wind Farm Course Project Requirements Document*, to begin designing the software architecture for the Wind Farm course project.

Your design may include diagrams and text. You should first consider the architecture, regardless of the programming language of the implementation. The architecture should be clear to C, Java, and LabVIEW developers. Use your remaining time to consider the LabVIEW implementation. Which design patterns will you use? Which communication components might be more difficult to implement?

Wind Farm Course Project Requirements Document

V I Engineering's Chief Architect Christopher G. Relf contributed to the following requirements example. This example represents part of a deliverable from a V I Engineering led Requirements Engineering phase and shows how LabVIEW architects can assist clients in documenting requirements.

1. Introduction

1.1. Purpose

The purpose of the Systems Requirements Specification (SyRS) is to enumerate the functions of the system and define in detail how the system interfaces to both surrounding systems and its users. This document is intended for both the customer and the developers implementing the system.

1.2. Definitions of Terms

This section provides the definitions of specific terms used to construct this SyRS.

- The word SHALL denotes a mandatory requirement. Departure from such a requirement is not permissible without formal agreement.
- The word SHOULD denotes a recommendation or advice on implementing such a requirement of the document.
- The word WILL denotes a provision or service, or an intention in connection with a requirement.
- The word MAY denotes a permissible practice or action. It does not express a requirement.

1.3. Product Description

The course project has been designed to give students the opportunity to design and evaluate several potential architectures for a larger LabVIEW application. Students will begin an initial design of the Wind Farm at the beginning of the course and then consider the implementation, utilizing programming techniques that they currently understand. Then several advanced design patterns will be taught and discussed. The merits of each will be analyzed as they apply to the overall course project.

A wind farm simulation was chosen as the course project because it is a relatively straight forward concept to understand. Key parameters that must be monitored are obvious. The number of wind turbines in operation is a variable that is only known at runtime. More over, multiple processes must run asynchronously and this can pose a challenge when designing a software architecture, even in LabVIEW.

The Wind Farm Course Project is simply an application with which to illustrate concepts that are taught in the course. Each and every concept can easily be applied to your application that you are developing at your workplace. In every case, either the results of the exercise or the process utilized in the exercise is one which you can leverage outside of the course immediately.

1.4. Assumptions

1.4.1. Simplifying a Complex System

Each module or system that will be coded or used in the course project may appear to be quite simple. Students should assume that each of these systems actually represents a complex process that might take weeks or months to develop. For the purposes of this course, a very complex system was significantly simplified.

1.4.2. Vague User Interface Requirements

A clearly defined user interface is not included in the wind farm software requirements. Only general features have been specified. This was crafted intentionally to allow the students flexibility in assessing several different designs that have different ramifications for the user interface

1.4.3. Additional Requirements

A typical requirements specification would include much more information than is noted below. The wind farm requirements constitute a subset that might be classified as interface requirements, functional requirements and design constraints. A well researched and carefully crafted requirements specification would include many additional categories such as: hardware interfaces, communication interfaces, networking with other systems, safety issues, constraints, documentation, and much more. Additionally, each requirement should have a unique ID, a priority, and an identified user. For the purpose of this exercise, you may assume that all requirements must be implemented and the user is the operator. For more information on writing requirements refer to the following:

- IEEE Std 1233, 1998 and 1233-1996 aIEEE Guide for Developing System Requirements Specifications -Description
- *Managing Software Engineering in LabVIEW*
- *Large LabVIEW Application Development* Community Page on `ni.com`
- *Writing Better Requirements* by Alexander and Stevens

1.5.Product Description

A wind farm is comprised of many wind turbines. Each wind turbine is a sophisticated system that measures wind speed, wind direction and many operational parameters related to rotating machinery. The optimum energy production occurs when the turbine rotates at a specified speed. The angle of attack and the direction of the turbine are adjusted by the control system to take advantage of the current wind conditions. This course project does not include an actual simulation or the complexities of a typical control system. Additionally, the course project runs on one target. An actual wind farm is a sophisticated, networked system.

2. Product Requirements

10-0000 Interface Requirements

11-0000 Operator Interfaces

11-1000 General

11-1001 The user interface shall include a simple display of the operation of the wind turbines.

11-1002 The user interface will be intuitive and require minimal training.

11-1003 The user interface shall not contain tabs.

11-2000 Wind Turbine Control and Data Visualization

11-2001 The user interface shall provide the capability for an operator to add a wind turbine to the display.

11-2002 The user interface shall provide the capability for an operator to display the turbine rotation speed from each wind turbine.

11-2003 The user interface may provide the capability for an operator to stop a wind turbine.

11-2004 The user interface may provide the capability for an operator to restart a wind turbine.

11-2005 The user interface may provide the capability for an operator to remove a wind turbine from the display.

11-2006 The user interface may provide the capability for an operator to stop all displayed wind turbines.

11-3000 Software Shutdown

11-3001 The user interface shall provide the capability for an operator to shut the system down.

11-3002 The user interface may display a message on successful shutdown.

20-0000 Functional Requirements

21-0000 Main Software

21-1000 Data Acquisition

21-1001 The system shall acquire each wind turbine’s rotational speed in RPM.

21-1002 The system may acquire each wind turbine's blade angle of attack in degrees relative to the support shaft.

21-1003 The system may acquire each wind turbine’s generated power in Watts.

21-1004 The system may acquire each wind turbine’s energy generation.

21-2000 Data Logging

21-2001 The test software shall create a data log for each running wind turbine.

21-2002 The data logs shall contain the name of the wind turbine tested.

21-2003 The data logs shall contain the time and date when each turbine was started.

21-2004 The data logs shall contain the rotation speed of each wind turbine at 250 msec intervals.

21-2005 The data logs shall be saved to disk in one of the following user-specified formats:

- XML
- Tab-Delimited ASCII Text

21-2006 The data logs may also need to be saved to disk in additional formats, such as TDMS, in the future.

21-3000 Wind Turbine Simulation

21-3001 A wind turbine shall receive the wind speed input.

21-3002 A wind turbine shall output the corresponding rotation speed.

30-0000 Design Constraints

31-0000 Operator Interfaces

31-1000 General

31-1001 The test system software shall be limited to one physical screen.

31-1002 The user interface shall be of a maximum resolution of 1280 x 1024.

End of Exercise 1-1

Exercise 1-2 Analyzing a Design Pattern

Goal

To analyze the Producer/Consumer Design Pattern (Events).

Scenario

The Producer/Consumer Design Pattern (Events) is a foundational starting point for many subcomponents for any software architecture. However, several changes should be made to the template before it becomes a robust starting point for component development. National Instruments provides a basic pattern that is generic to any application or use case. Further customization of the template to handle unique requirements for each programming team saves considerable time and effort during software development. The architect, with assistance from a larger team when available, must determine the needed changes.

Discussion Questions

Open the Producer/Consumer Design Pattern (Events) from the template browser and answer the following questions:

- ❑ When might you use this design pattern?

- ❑ What is missing from this design pattern? What changes must be made every single time the pattern is used in any application?

- ❑ What common tasks used by most LabVIEW developers should be included in a template based on this design pattern?

- ❑ What unique tasks for your application should be included in a template based on this design pattern?

Discussion Answers

- ❑ When might you use this design pattern?

  You might use this design pattern when you need a producer loop to handle events and maintain a responsive user interface and a parallel consumer loop to asynchronously handle the processing of the event and its data.

- ❑ What is missing from this design pattern? What changes must be made every single time the pattern is used in any application?

  The producer loop will not know if the consumer loop stopped or generated an error and stopped dequeueing elements.

  The design pattern lacks a safe way to stop the consumer loop. Currently, the design pattern stops the consumer loop by releasing the queue, which causes the consumer loop to stop even if there are still elements left in the queue.

  The queue data should be made into a typedef to make the code more maintainable.

- ❑ What common tasks used by most LabVIEW developers should be included in a template based on this design pattern?

  You should include common tasks like Initialize, Shutdown, Panel Close event that stops the application in the development environment and exits the application in the run-time environment, Error Handling, Handle the dequeued data, and so on.

End of Exercise 1-2

Exercise 1-3 Identifying Design Challenges

Goal

To list three design challenges in an application that you are currently working on or that you have worked on in the past (not the Wind Farm Course Project).

Scenario

In this course, you learn several advanced programming techniques as well as some tips and tricks. In order to facilitate integrating the ideas into your applications, you must first consider key challenges and periodically return to this list to determine how the technique could apply.

Implementation

Take a moment to think about your current application or a recent project. Consider data structures, modules, processes, and behavior. Determine which aspects pose the greatest challenge when designing an architecture. What were or are the greatest coding challenges? Identify and document the top three issues in that application.

Note Common challenges include deciding what will run synchronously and what will be asynchronous, designing the asynchronous communication mechanism, or crafting cohesive modules such that changes have limited affects on the rest of the code and minimal coding is needed to make those changes.

1.

2.

3.

End of Exercise 1-3

Notes

2

Designing an API

Exercise 2-1 Evaluating an API Design

Goal

To review the design decisions behind the STM API.

Implementation

The installer for the Simple Messaging Reference Library (STM) is included on your course CD. The installer places the VIs in a subpalette under `user.lib`.

1. Install the STM Reference Library.

   - ❑ Navigate to the `<Exercises>\Advanced Architectures in LabVIEW\Adv Design Patterns and Tools\Installer for STM` and run `setup.exe`.

Note The VIs used in this exercise are automatically installed to `<Program Files>\National Instruments\LabVIEW 2011\user.lib`.

2. If LabVIEW is open, you must close and relaunch it in order to access the STM Reference Library.

3. Use and examine the STM Reference Library.

   - ❑ Open a blank VI and view the block diagram.

   - ❑ From the palette, place the **User Library»STM»Examples» Basic Client-Server»STM Basic Client Example.vi** on the block diagram.

   - ❑ From the palette, place the **User Library»STM»Examples» Basic Client-Server»STM Basic Server Example.vi** on the block diagram.

   - ❑ View the front panel for both VIs.

   - ❑ Run the STM Basic Server Example VI. You must run the server first.

- ❑ Run the STM Basic Client Example VI. Communication is on the local host. If you are in a classroom with networked computers, you can run this example on two computers.

Note To run this example on two computers, you must determine the IP address of the computer running `STM Basic Server Example.vi`. From the Windows Start menu, open a Windows Command Prompt and enter `ipconfig`. Press <Enter> to run the command which displays the IP address.

- ❑ Stop the VIs.
- ❑ Open the block diagram for both VIs and notice the design decisions that were made.

Discussion Questions

1. In what ways are the icons clear and meaningful?

2. How has connector pane selection facilitated the use of this API?

3. How does the choice in label names make the VIs easy to use?

4. How has the use of polymorphic VIs made this library easy to use?

5. For any of the previous questions, what improvements could be made?

6. How might this library change in the future?

Discussion Answers

1. In what ways are the icons clear and meaningful?

   The VI icons have a common STM banner to show that they are part of the same API.

   The icons have meaningful graphics, such as arrows to indicate the direction of the communication. The text in the icons indicates whether the polymorphic VI is using the TCP, UDP, or Serial instance.

2. How has connector pane selection facilitated the use of this API?

   The connector panes consistently use the lower terminals for error wires and the upper terminals for the connection information

3. How does the choice in label names make the VIs easy to use?

   The label names all have a common STM or TCP prefix. The label names also very clearly describe the VI's functionality (i.e. STM Read Message, STM Read Meta Data).

4. How has the use of polymorphic VIs made this library easy to use?

   You can use the same polymorphic VIs to implement TCP, UDP, or serial communication. The polymorphic VIs automatically adapt based on whether you wire a TCP connection ID, UDP connection ID, VISA resource, or an existing Connection Info cluster to the top-left input terminal.

5. For any of the previous questions, what improvements could be made?

   You can add a block diagram comment to the example VIs explaining how to use the polymorphic VIs. It is not obvious that the STM API contains polymorphic VIs and UDP, TCP, and serial implementations. You can also add a separate example for a UDP, TCP, and serial implementation.

6. How might this library change in the future?

   You might add more protocols in addition to UDP, TCP, and serial.

End of Exercise 2-1

Notes

Notes

3

Multiple Processes and Inter-Process Communication

Exercise 3-1 Queue-Driven Message Handler

Goal

To become familiar with a typical queue driven message handler template.

Scenario

Some type of queue driven message handler (QDMH) or state machine is a fundamental starting point for successful LabVIEW teams and individual programmers. In existence for over a decade, this advanced design pattern is well-established and widely-used.

The QDMH used in this course is not intended to be the solution for every student. Rather, it is an example of what a development team could build from the Producer/Consumer design pattern. It does include key features commonly found in many development teams' applications. Consider this as a starting point and continue to expand or modify as needed. As is, the QDMH is sufficient for use in any of your applications.

Implementation

1. Create a new VI from a template.
   - ❑ Open `<Exercises>\Advanced Architectures in LabVIEW\Shared\QDMH Main.vit`.
   - ❑ Explore the block diagram.

Note Support VIs are located in the `<Exercises>\Advanced Architectures in LabVIEW\Shared` directory.

Tip Because this is an advanced course, some exercises do not specify the details for every step.

- ❑ Save the main VI and the QDMH Main Cluster VI in the `<Exercises>\Advanced Architectures in LabVIEW` directory.

Note Because the main VI includes a sub VI that is also a template, you must also save the sub VI in order to save the main VI.

2. Add a few Boolean controls to the main VI.
3. Modify the QDMH so that the message display updates when the control value changes.

End of Exercise 3-1

Exercise 3-2 JKI State Machine (Optional)

Goal

Explore a state machine in which the architect made different decisions on the design.

Scenario

Members of JKI Software have been extraordinarily involved in the LabVIEW community and in the development and promotion of open source tools. Additionally they engage in LabVIEW integration projects and work with teams that need assistance in software architecture and the deployment and maintenance of LabVIEW tools across an enterprise. LabVIEW architects should become informed about the tools that this team provides (`www.jkisoft.com`).

JKI Software provides a very flexible and well thought-out state machine to the LabVIEW community at no cost. JKI made several design decision that are different from what most students have seen in previous LabVIEW courses and in the broader LabVIEW community. First, the goal was to craft a very thin template with minimal support VIs. Secondly, JKI has defined a scripting methodology that allows for the rapid development state definitions and transitions. However, you may notice that a certain component is missing from the template. JKI purposefully left this feature out, thus allowing the user of the template to determine the preferred mechanism for peer to peer communication between parallel loops.

Implementation

Install and use the JKI State Machine.

1. Download and install VI Package Manager.
   - ❑ Open a Web browser and navigate to `www.jkisoft.com`.
   - ❑ Install VI Package Manager.
   - ❑ When prompted for an activation key, select **Cancel**.
   - ❑ Run the VI Package Manager, which allows you to install JKI toolkits, including the JKI State Machine.
2. Explore and modify the JKI state machine.
   - ❑ Open a blank VI.
   - ❑ Place the **JKI State Machine** on the block diagram. You can now find it in the Functions palette.

- ❑ On the front panel, add two Boolean controls and rename them as `Test 1` and `Test 2`.
- ❑ Add a string indicator labeled `Message`.
- ❑ Modify the block diagram so that the label of the control appears in the message indicator when either of the Boolean controls is pressed.
- ❑ Further explore the block diagram. What one feature is missing?

Challenge

A mechanism must be included to move data into and out of the state machine from other places in the application. What are the various ways to implement this? Why would you choose one over the other?

End of Exercise 3-2

Exercise 3-3 Spawning Multiple Instances of an Asynchronous Independent VI

Goal

Create a subVI that you can use to spawn multiple instances of an asynchronous independent VI.

Scenario

In this exercise, you begin with a reentrant VI. You must create a subVI to asynchronously launch the VI to run independently. You must then use this subVI in a main VI to launch multiple instances of the asynchronous independent VI.

Implementation

Explore the Asynchronous Independent VI

1. Open `<Exercises>\Advanced Architectures in LabVIEW\Asynchronous Independent VI\Spawning Multiple Instances.lvproj`.

2. From the Project Explorer, open **Asynchronous Independent VI»Spawned VI (reentrant).vi**.

   This is the asynchronous independent VI in this exercise.

3. Observe the VI Properties of the Spawned VI (reentrant) VI.

   - ❑ Select **File»VI Properties** to open the VI Properties window.
   - ❑ Set the Category to **Execution**.
   - ❑ Notice that this VI is configured to be reentrant.
   - ❑ Close the VI Properties window.

4. Explore the block diagram.

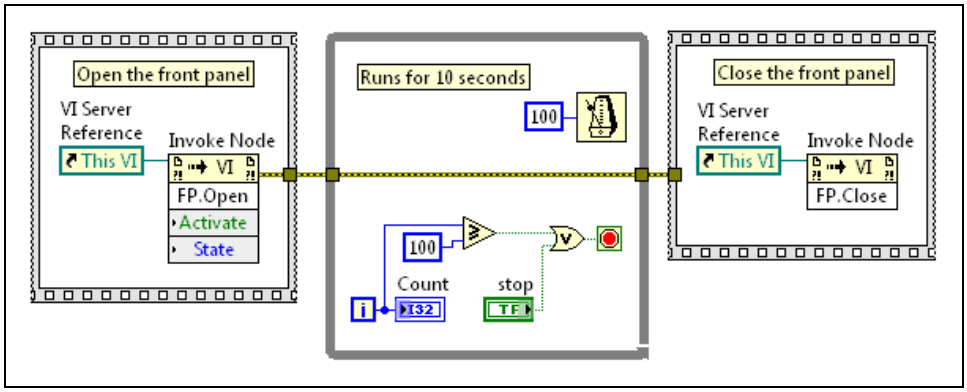


Figure 3-1. Spawned VI (reentrant) VI Block Diagram

- ❑ Notice that this VI programmatically opens its front panel when it start running and closes its front panel immediately before it finishes execution.

  Each instance of this asynchronous independent VI displays its front panel when launched.

5. Close the Spawned VI (reentrant) VI.

Create the Launcher SubVI

1. From the Project Explorer, open **Launcher subVI»Launch Asynchronous VI.vi**.
2. Modify the block diagram to launch an asynchronous independent VI, as shown in Figure 3-2, using the following items:

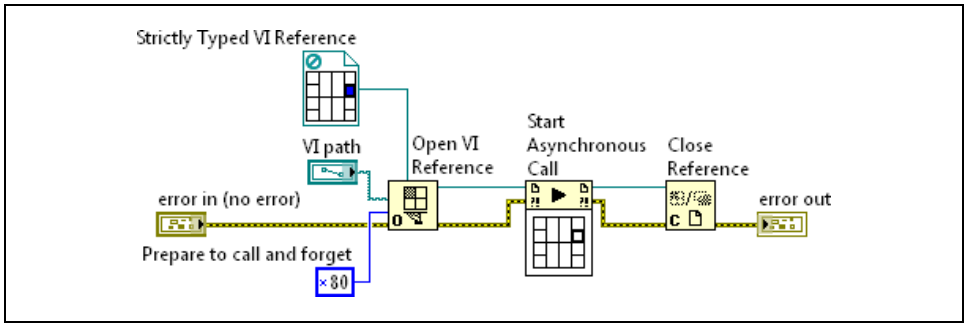


Figure 3-2. Launch Asynchronous VI Block Diagram

- ❑ Open VI Reference function—Configure this function to create a call-and-forget VI reference to be called asynchronously without returning its results.
  - – Right-click the type specifier VI Refnum input and select **Create»Constant**. Right-click the constant and select **Select VI Server Class»Browse**. Navigate to `<Exercises>\Advanced Architectures in LabVIEW\Asynchronous Independent VI\Spawned VI (reentrant).vi` and click **OK**.
  - – Right-click the options input and select **Create»Constant**. Right-click the constant and select **Visible Items»Radix**. Click the radix and select **Hex**, as shown in Figure 3-2. Set the constant to `80`.
- ❑ Start Asynchronous Call node—Starts an asynchronous call to the VI indicated by the reference input.
- ❑ Close Reference function—Closing a call-and-forget VI Reference allows the asynchronous VI to manage its own lifetime. The asynchronous independent VI leaves memory when it stops running and closes front panel.
- ❑ Wire the block diagram as shown in Figure 3-2.

3. Test the Launch Asynchronous VI.
   - ❑ Go to the front panel.
   - ❑ Set VI path to launch `Spawned VI (reentrant).vi`.
   - ❑ Run the VI.

     The Launch Asynchronous VI should launch the Spawned VI (reentrant) VI and immediately stop running. Meanwhile, the Spawned VI (reentrant) VI should keep running independently until it finishes execution. After it finishes execution, it will close its front panel and leave memory.
   - ❑ Try running the Launch Asynchronous VI multiple times to launch multiple instances of the reentrant asynchronous independent VI.
4. Close the Launch Asynchronous VI when finished.

Call the Launcher SubVI from a Main VI

1. From the Project Explorer, open TEST Launch from Main.vi.
2. Explore and run TEST Launch from Main.vi.
3. Click the Launch button multiple times to launch multiple instances of the asynchronous independent VI.
4. Notice when `Spawned VI (reentrant).vi` enters and leaves memory.
5. Close the project and VIs when finished.

End of Exercise 3-3

Exercise 3-4 Fixing the Run VI Method

Goal

To fix a VI in which VI references are not handled correctly when using the Run VI method.

Scenario

As the architect, you must understand and be able to explain to your team how LabVIEW handles VI references. In this exercise, the goal is to understand how references work using the Run VI method to launch an asynchronous VI with no open front panel.

Note Opening the front panel for the spawned VI is not the solution for the exercise.

Implementation

1. Open `<Exercises>\Advanced Architectures in LabVIEW\Fixing Run VI Method\Run VI Method.lvproj`.

   The dynamically spawned VI is `Log Time to File.vi`.

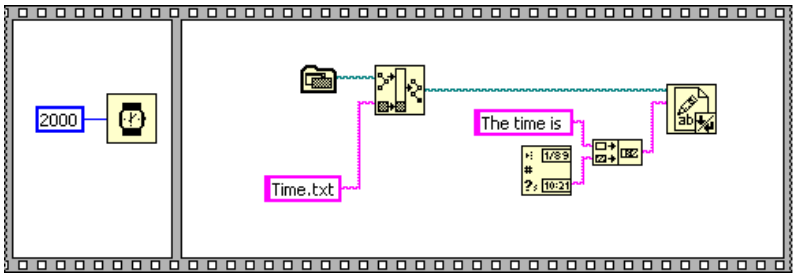


Figure 3-3. Log Time to File VI

   - ❑ Run `VI Launcher - no GUI.vi` and confirm whether `Time.txt` was created correctly in the `Fixing Run VI Method` directory.
   - ❑ Fix the Launcher so that the `Time.txt` file is generated correctly.

Why did your changes fix the VI?

End of Exercise 3-4

Exercise 3-5 Using Single Element Queues (SEQ)

Goal

To build a Wind Turbine Class with a Single Element Queue (SEQ) architecture.

Scenario

In order to pass critical information from the Wind Farm UI to the individual Wind Turbine code, you create a Wind Turbine Object. This is an object that you can instantiate an infinite number of times at runtime. Additionally, the data is protected so that while one VI modifies the data, other VIs are blocked. This class includes a subset of methods from the original requirements specification. Though this class is unique to a Wind Turbine object, you can use the templates and the process immediately in your application.

Design

Because the process for creating the class is somewhat unique and a portion of it uses scripting, follow the detailed directions in the Implementation section.

Implementation

Using Single Element Queue templates, a replicate hierarchy tool, and VI scripting, create a by Reference Wind Turbine Object. The purpose of the object is not to run the wind turbine process, but to communicate between a host and multiple wind turbines.

1. Duplicate the hierarchy.

   - ❑ Open the `By Reference Object.lvproj` located in the `<Exercises>\Advanced Architectures in LabVIEW\By Reference Objects` directory. You will use code in the `SEQueue-Templates` folder and the Scripting for Get Set folder to create your Wind Turbine Class. Code that you create will be saved in the `SEQ Wind Turbine Object` folder.

   - ❑ Open the `Duplicate Folder with New Keyword.vi` located in the **Duplicate Hierarchy SubVIs** directory of the project.

Note This is a generic tool you can use to duplicate any template hierarchy and replace keywords that are in the template with new keywords.

❑ The root path should point to the templates directory. The remaining controls should resemble Figure 3-4.

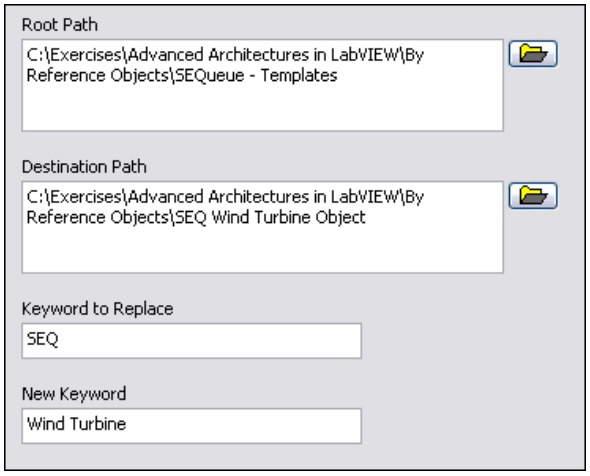


Figure 3-4. Duplicate Folder with New Keyword VI Front Panel

❑ Run the VI and click **OK** after you have confirmed that all inputs are correct. The VI may take several seconds to run.

❑ In Windows Explorer, navigate to the SEQ Wind Turbine Object subfolder to see the files created by the tool.

2. Modify controls and the Wind Turbine Create VI, Wind Turbine Destroy VI, and the Wind Turbine Object Snooper VI.

❑ Open `Wind Turbine Data.ctl` from the `<Exercises>\Advanced Architectures in LabVIEW\By Reference Objects\SEQ Wind Turbine Object\Controls` directory.

❑ Remove or modify the numeric in the cluster.

❑ Add five numeric controls to the cluster. Each should remain double precision. Use the following labels:

– Wind Speed (m/s)

– Turbine Speed (RPM)

– Angle Attack (Degs)

– Current Power (kW)

– Energy Generation (kW*hr)

❑ Your control should resemble Figure 3-5.

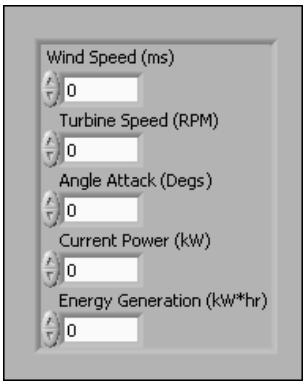


Figure 3-5. Wind Turbine Data Control

❑ Save and close the control.

❑ Open `Wind Turbine DataRef.ctl`.

❑ Open `Wind Turbine Create.vi`.

❑ Open `Wind Turbine Destroy.vi`.

❑ Open `Wind Turbine Object Snooper.vi`.

Note Do not open `Wind Turbine Get.vi` or `Wind Turbine Set.vi` that are located in the **Not Used** subfolder. You will use the new VI Scripting to create the get and set methods for the class.

- ❑ You may need to clean up the front panels. Your VIs should resemble Figure 3-6, Figure 3-7, and Figure 3-8.

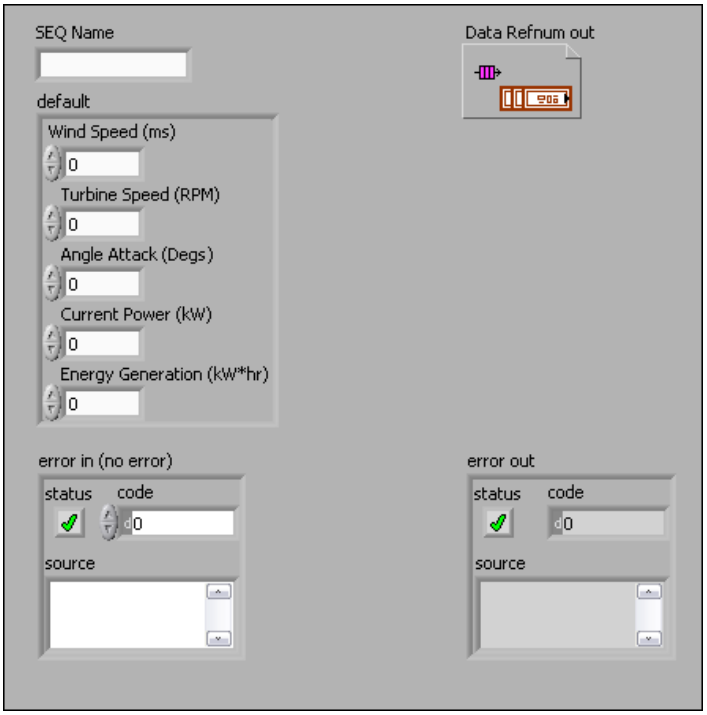


Figure 3-6. Wind Turbine Create VI Front Panel

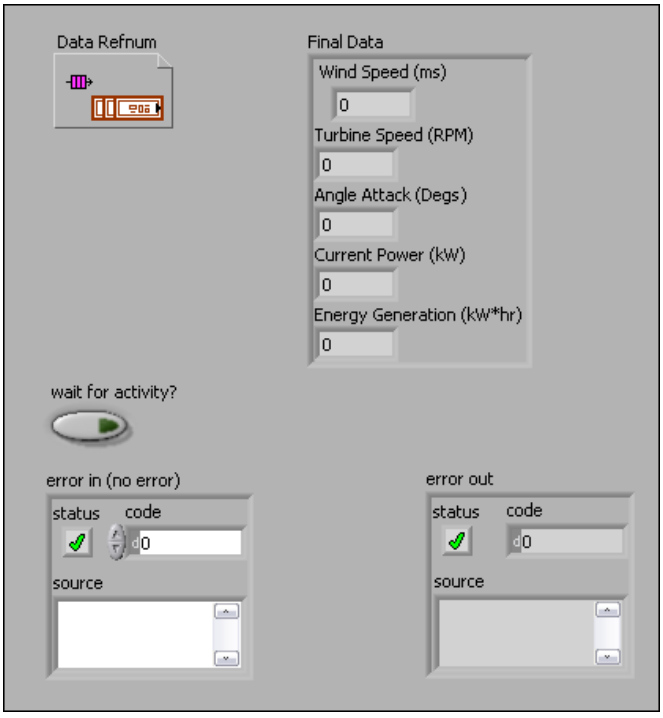


Figure 3-7. Wind Turbine Destroy VI Front Panel

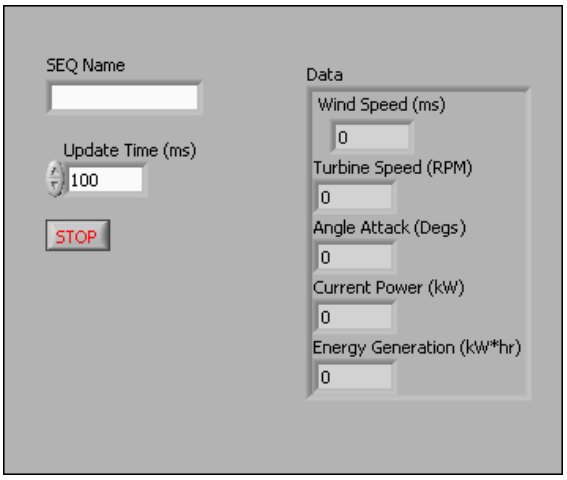


Figure 3-8. Wind Turbine Object Snooper VI Front Panel

3. Save and close all open VIs.

4. Add the following VIs to the SEQ Wind Turbine Object virtual folder in the Project Explorer.

   - ❑ `Wind Turbine Create.vi`
   - ❑ `Wind Turbine Destroy.vi`
   - ❑ `Wind Turbine Object Snooper.vi`

5. Use VI Scripting to create the Get and Set methods.

Note VI Scripting allows the programmer to use LabVIEW to create and modify LabVIEW code. Refer to Lesson 7, *Tips, Tricks, & Other Techniques*, for more information about VI Scripting.

In this exercise, two templates are provided: `niSYS Export GetTemplate.vit` and `niSYS Util Export Set Template.vit`. The scripting code is niSYS UTIL Create Exports. These VIs have been internally used at National Instruments by the System Engineering group.

Before running the VI, you must select the path to the Queue reference. The scripting VI then extracts the cluster elements from the reference and allows the user to specify which one will be get or set. Additionally the template VIs have been modified to include the icon for this exercise.

- ❑ Open `niSYS UTIL Create Exports.vi` located in the `<Exercises>\Advanced Architectures in LabVIEW\Scripting\Scripting for Get Set` directory.

- ❑ Verify that the Set/Get switch is in the **Set** position.
- ❑ In the Object DataRef Path control, browse to the `Wind Turbine DataRef.ctl`. The front panel should resemble the Figure 3-9.

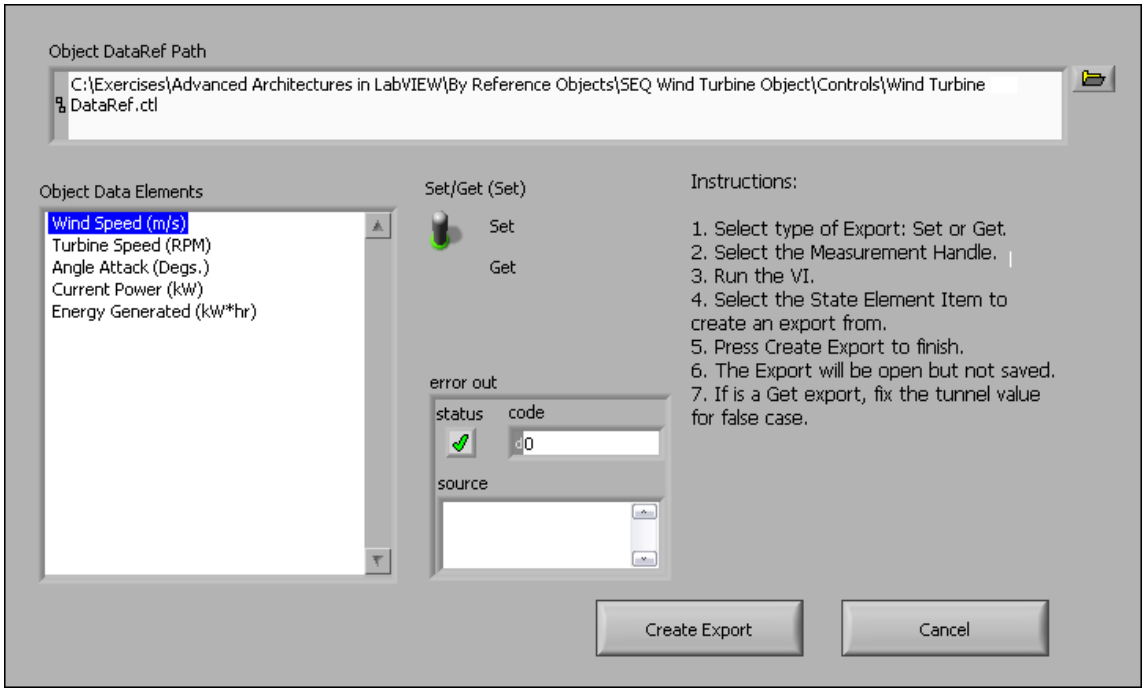


Figure 3-9. niSYS UTIL Create Exports VI Front Panel

- ❑ Run the VI.
- ❑ Move the VI so you can see the entire front panel of the newly-created VI.
- ❑ Select **Wind Speed** in the Object Data Elements listbox control.
- ❑ Select **Create Export**. Observe that a control is placed on the newly-created VI.
- ❑ Navigate to the newly-created Set VI.

- ❑ Additionally you should add the text `Wind Speed` to the VI icon.
- ❑ Your VI should resemble Figure 3-10.

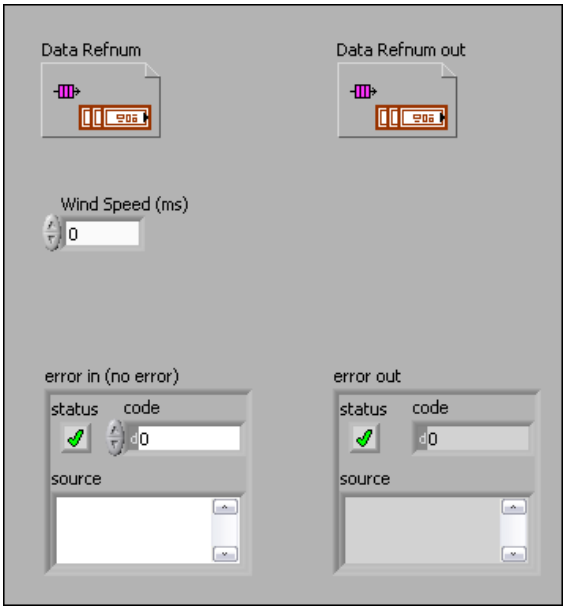


Figure 3-10. Wind Turbine Set Wind Speed VI Front Panel

- ❑ Save the VI as `<Exercises>\Advanced Architectures in LabVIEW\By Reference Object\SEQ Wind Turbine Object\VIs\Wind Turbine Set Wind Speed.vi`.
- ❑ Repeat the process and use the Set/Get Boolean control and Object Data Elements listbox control to create the following VIs:
  - `Wind Turbine Get Wind Speed.vi`
  - `Wind Turbine Set Turbine Speed.vi`
  - `Wind Turbine Get Turbine Speed.vi`
- ❑ Using the icon banner for the VIs that you created, modify the icons for the Create, Destroy, and Object Snooper that you previously created.
- ❑ Save and close all VIs.

6. Add the following VIs to the SEQ Wind Turbine Object virtual folder in the Project Explorer.
   - ❑ Wind Turbine Set Wind Speed.vi
   - ❑ Wind Turbine Get Wind Speed.vi
   - ❑ Wind Turbine Set Turbine Speed.vi
   - ❑ Wind Turbine Get Turbine Speed.vi
7. Save the project.

Testing

Now, you will test the Wind Turbine Class that you created. You will run two separate VIs that have been set up to run three Wind Turbines. This is a very simple test VI.

1. Open the block diagram for `TEST Set Wind Speeds.vi`. The test code is written for you. Add the following VIs in the appropriate locations and wire inputs and outputs.
   - ❑ `Wind Turbine Create.vi`
   - ❑ `Wind Turbine Set Wind Speed.vi`
   - ❑ `Wind Turbine Destroy.vi`
2. Open the block diagram for `TEST Get Wind Speeds.vi`. The test code is written for you. Add the following VIs in the appropriate locations and wire inputs and outputs.
   - ❑ `Wind Turbine Create.vi`
   - ❑ `Wind Turbine Get Wind Speed.vi`
   - ❑ `Wind Turbine Destroy.vi`
3. Navigate back to the front panels of both VIs.
4. Run both VIs.
5. Set the Wind Speeds in the TEST Set Wind Speeds VI.
6. Confirm that the TEST Get Wind Speeds VI reads the Wind Speeds correctly.

7. Debug as needed.

8. Save and close.

Challenge

Review the list of challenges you created in Exercise 1-3. Determine how you might integrate this into your current application.

What features from the scripting VI are missing that you might want to incorporate?

End of Exercise 3-5

Exercise 3-6 Using Data Value References

Goal

To examine the design of a Timer class that was created with the Data Value Reference (DVR) feature in LabVIEW.

Scenario

You will inevitably encounter a scenario in which you need to instantiate more than one instance of an object. Additionally, you might need a fast and efficient way to access data from more than one location in your block diagram. One way to design the component is using Single Element Queues. Another option is to use the DVR feature.

Implementation

1. Explore the DVR Timer class that was demonstrated during the Functional Global Variables section of the course.

   ❑ Open `<Exercises>\Advanced Architectures in LabVIEW\Demonstrations\FGV vs DVR\Data Value Reference\DVR Object.lvproj`.

Note The Demonstrations folder contains valuable instructor demos you may find useful in developing your applications.

   ❑ Expand the DVR Object project, as shown in Figure 3-11.

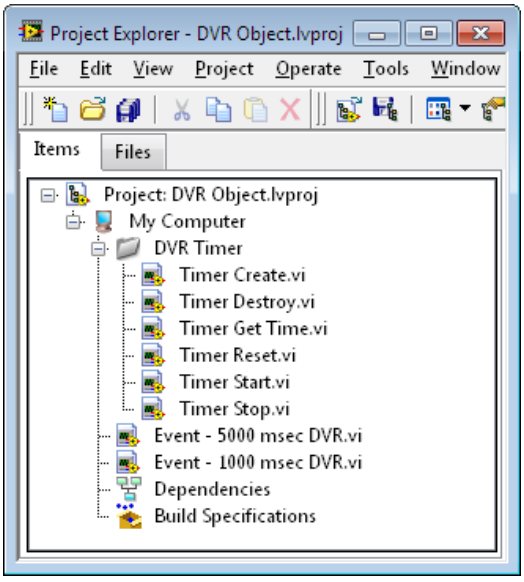


Figure 3-11. DVR Object Project

- ❑ Open and run `Event 1000 msec DVR.vi`.
- ❑ Open and run `Event 5000 msec DVR.vi`.
- ❑ Notice that each VI uses a separate instance of a timer.
- ❑ Explore the block diagram.
- ❑ Explore the organization of the DVR Timer class.

2. Close the VIs without saving.

Using a By Reference solution greatly enhances the ability to create multiple instances of objects and access them globally throughout the LabVIEW block diagram.

Challenge

Review the challenges you listed in Exercise 1-3. Determine which challenge might be solved with the DVR. Code a simple prototype. The screenshots in the slides and the block diagram of the DVR Timer demonstrate the basics for creating a DVR class.

Using the DVR, create a simple wind turbine object that gets and sets wind speed.

End of Exercise 3-6

Exercise 3-7 Functional Global Variables (Optional)

Goal

To create a scalable FGV that manages User Event References using a name-value pair template for a Functional Global Variable (FGV).

Scenario

As the architect, you must determine the extent to which your team uses FGVs in your application. A name-value pair FGV is useful for storing and retrieving data items across an application. If you plan to use FGVs like this, you should construct a template for your developers. In this exercise, you use a name-value template to build a User Event Reference manager. You can use this template as-is for your development team or you can customize it further.

User Events are a very powerful tool for moving data from one block diagram location to another. Subsequently, any block of code can cause an event to be fired that would be handled by the event structure. Even though complex architectures can be developed without User Events, they still serve as a very powerful tool for the architect. However, the architect must determine the method for getting the User Event Reference from one part of the block diagram to another. A name-value FGV is one way to solve that problem, and in a way that it scales to an unlimited number of User Events. Moreover, you can use this VI in any application.

Design

You create a new FGV from `FGV Resource Template.vit`, located in the templates directory.

- Create the FGV from the template.
- Create a User Event Reference to replace the existing dataset.
- The user event data type is cluster with a string and variant.

Note You must label the string and variant before creating the User Event Reference. Otherwise, the event structure can not retrieve the information that is passed via the User Event.

- Add the appropriate User Event VIs to the FGV.
- Delete the Modify case as the User Event References is never modified.
- Make appropriate changes to labels and the Method enumerated type definition.

Implementation

First, you create a new functional global variable from the template. You modify some control labels and redefine two type definitions.

1. Create a new functional global variable.

   - ❑ Open `<Exercises>\Advanced Architectures in LabVIEW\FGV\Functional Global Variable.lvproj`. Other than the initial creation of the functional global variable from the template, this will be the working project for the exercise.
   - ❑ Expand the build specifications.
   - ❑ Right-click **Install to Templates** and select **Build**.

Note Explore the properties of the build specifications. This source distribution places the FGV templates in the appropriate directory so that they are available in the template browser.

   - ❑ In the Build Status dialog, click **Done**.
   - ❑ In the project explorer, select **File»New**.
   - ❑ In the Create New list, select **VI»From Template»Frameworks» Design Patterns»FGV Resource Template.vit**.

     This is the template that you just installed.
   - ❑ Click **OK**.
   - ❑ Edit the icon to say `UE Ref Mgr`.
   - ❑ Rename and save the template components to the `Functional Global Variables` directory. The template uses two type definition templates in addition to the VI. You must save all three components.

Note The order for saving may be different from the order shown here. Carefully observe what you are saving at this step.

     – Save the VI as `FGV User Event Reference Manager.vi` to the `FGV` directory.
     – Save the DataSet type definition as `TYPE FGV User Event Ref` in the `<Exercises>\Advanced Architectures in LabVIEW\FGV` directory.

- Save the Method type definition as `TYPE FGV User Event Method` in the `<Exercises>\Advanced Architectures in LabVIEW\FGV` directory.

❑ On the front panel, change the label `DataSet Name` to `User Event Name`.

❑ Navigate to the block diagram and explore the cases.

2. Modify the type defined enumerated control, `Method (Get DataSet)`.

❑ Change the label to `Method (Get User Event Ref)`.

❑ Open the type definition by right-clicking the control and selecting **Open Type Def**.

❑ Right-click and select **Edit Items**.

- Change all instances of `DataSet` to `User Event Ref`.
- Change all instances `DataSets` to `User Event Refs`.
- Click **OK**.

❑ Select **File»Apply Changes**.

❑ Save the type definition.

3. Modify the `DataSet` type definition to contain a user event reference. Before making the actual changes create a User Event Reference.

❑ Navigate to the **Add User Event Ref** case and place the `Create User Event` function on the block diagram (you use this later in the exercise as well as here).

❑ Create a cluster constant with a string and variant constant. Label the string as `String`, label the variant `Data`, and label the cluster `User Event`, as shown in Figure 3-12.

Tip In order to access the elements within an event structure, you must include labels for cluster elements.

You must create the variant constant from an existing variant function.

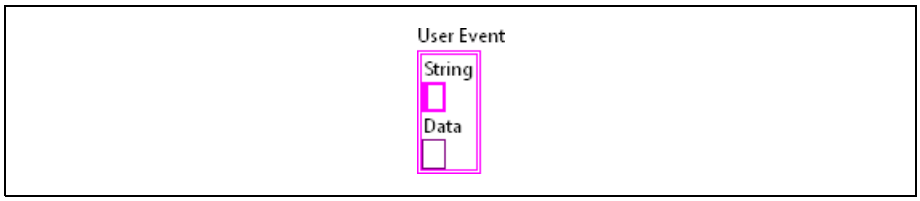


Figure 3-12. User Event cluster constant

- ❑ Wire the cluster constant to the `Create User Event` function.
- ❑ Create an indicator from the user event out output terminal of the Create User Event function. Select the indicator on the front panel and press <CTRL-X>.

  In a later step, you will paste it into the `TYPE FGV User Event Ref` type definition.
- ❑ On the front panel, open the `TYPE FGV User Event Ref` type definition by right-clicking the DataSet Out indicator and selecting Open Type Def.
- ❑ Delete the cluster.
- ❑ Select <CTRL-V> to paste the user event out indicator and rename it as `User Event`.
- ❑ Apply changes and then save and close.
- ❑ On the front panel, rename the DataSet indicators as `User Event Ref Out` and `All User Event Refs Out`.

  You do not need to rename the `DataSet` control because you will delete it later.

4. Create and destroy the User Event.
   - ❑ Navigate to the block diagram.
   - ❑ Go to the **Add User Event Ref** case.
   - ❑ Delete the DataSet control.

❑ Wire the **user event out** terminal from the Create User Event function to the build array, as shown in Figure 3-13.

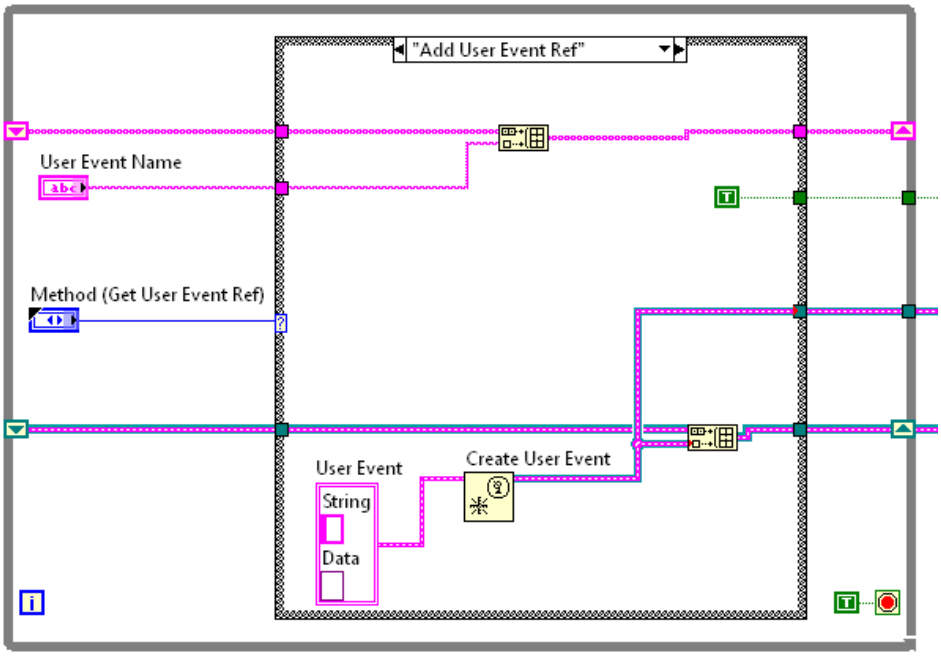


Figure 3-13. Add User Event Ref case of FGV User Event Reference Manager VI

❑ Navigate to the **Modify User Event Ref** case.

❑ Delete the case.

User Event references are only created or destroyed, never modified. Other functional global variables created from this template could have datasets that are modified.

❑ Navigate to the **Get User Event Ref** case.

❑ Explore the case. No changes are needed.

Tip One of the benefits of this advanced design pattern is that error checking code can be integrated into the functional global. The template already includes the check to ensure that the dataset or User Event Ref is indeed valid.

❑ Navigate to the **Delete User Event Ref** case.

❑ Inside the internal case structure labeled `Valid Data Set?`, add the `Destroy User Event` function to the True case to destroy the reference returned at the deleted portion terminal of the `Delete From Array` function.

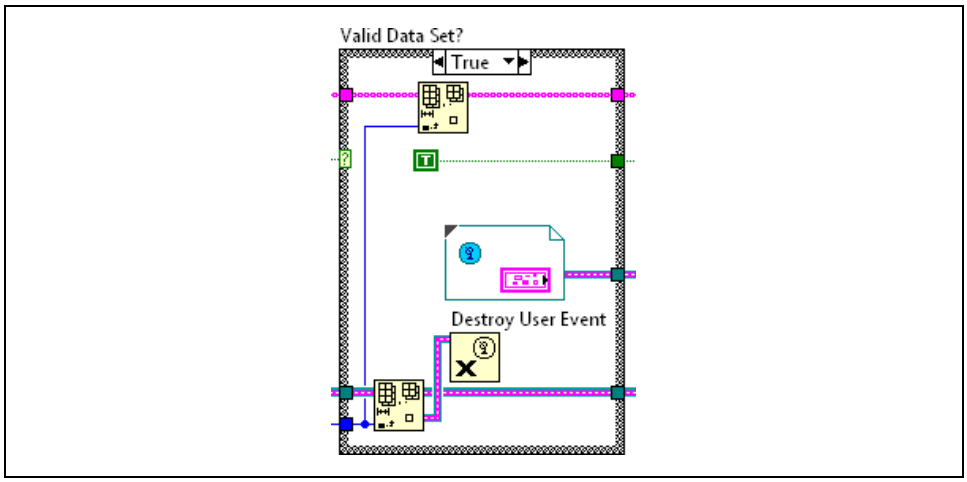


Figure 3-14. Delete User Event Ref case

- ❑ Create a constant from the tunnel to pass out an invalid user event.
- ❑ Navigate to the **Delete All User Event Refs** case.
- ❑ Place a `Destroy User Event` function inside a For Loop to destroy all the user events.

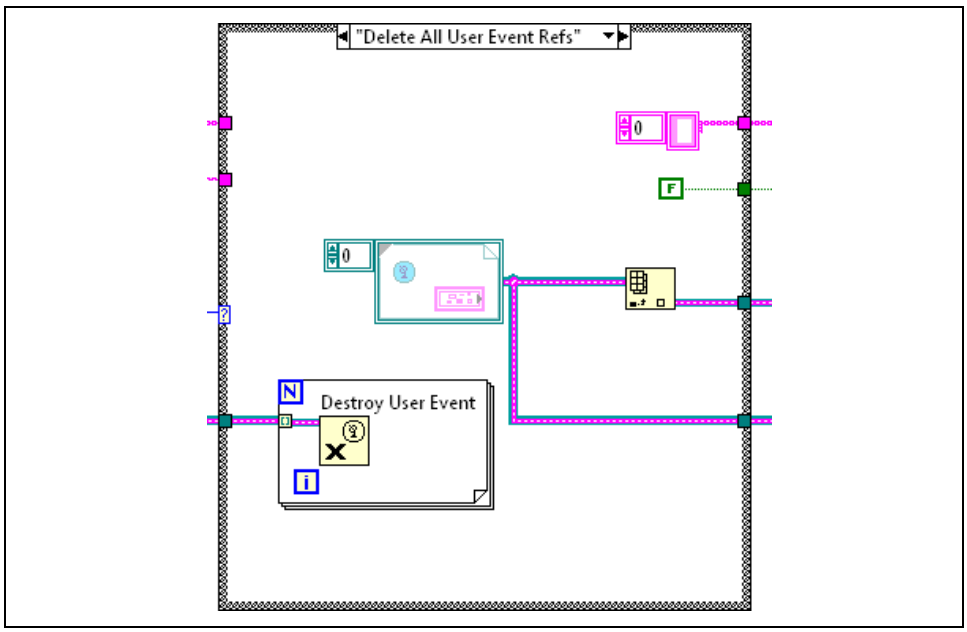


Figure 3-15. Delete All User Event Refs case

- ❑ Navigate to the **Get All User Event Refs** case.

  No changes need to be made.

- ❑ Save and close the VI.

5. Test the new functional global variable.

   You will modify two VIs. One VI allows you to add user event references and generate (or fire) user events to those references. The other VI handles, or responds to the user events. You now have a tool to globally create and manage user events.

   - ❑ Open `TEST FGV User Event Reference Manager.vi`.
   - ❑ Navigate to the following cases. Add the functional global variable with the method constant set according to the directions. Wire inputs and outputs as needed.
     - – Initialize
     - – Fire User Event (Which terminals must be wired for correct operation?)
     - – Add User Event (Which terminals must be wired for correct operation?)
     - – Delete User Event (Which terminals must be wired for correct operation?)
     - – Exit
   - ❑ Save the VI.
   - ❑ Open `TEST Handle FGV User Event Reference Manager.vi`.
   - ❑ Navigate to the block diagram and add the functional global variables as noted outside the While Loop and inside the Timeout case of the Event structure.

     Choose the correct selection for the Method input of the FGVs and wire the correct output of the FGVs to the Register for Events function.
   - ❑ Save the VI.
   - ❑ Navigate to the event case that handles the user event and examine it.

6. Test the functional global variable.

   - ❑ Run both test VIs.
   - ❑ Add user events from `TEST FGV User Event Reference Manager.vi`.
   - ❑ Fire various user events from `TEST FGV User Event Reference Manager.vi`.

     Ensure the user events are handled in `TEST Handle FGV User Event Reference Manager.vi`.

Tip If you need help, see the corresponding cases in `TEST FGV User Event Reference Manager.vi` and `TEST Handle FGV User Event Reference Manager.vi` in the `<Solutions>\Exercise 3-7` directory.

Challenge

Add instructions to the `.vit` so your development team can use it for other use cases.

List potential uses for this advanced design pattern. Review your answers on Exercise 1-3 to see if the functional global variable solves a challenge that you listed.

If you are already familiar with VI scripting, you may consider further automating this process.

End of Exercise 3-7

Notes

Notes

4

Advanced User Interface Techniques

Exercise 4-1 Creating an XControl (Optional)

Goal

To create an XControl that functions as a scalable radio control button.

Scenario

One benefit of XControls is that the architect can blend the functionality of several controls into one. In this exercise, you modify a Listbox control to have the look and feel of a radio button. Unlike the static radio button, this XControl scales dynamically while the program runs. The benefit is that the operator can see all of the items that are available (unlike a ring control) and which item is currently selected (unlike a plain Listbox). Moreover, this is a control that you can utilize in any application.

Design

Create and save the XControl with the names `X Listbox.xctl`, `X Listbox State.ctl`, `X Listbox Facade.vi`, and `X Listbox Init.vi`.

Implementation

1. Create the XControl and save the control and the abilities.

   - ❑ Open the `<Exercises>\Advanced Architectures in LabVIEW\X Controls\X Listbox\X Listbox.lvproj` directory.
   - ❑ In the `<Exercises>\Advanced Architectures in LabVIEW\X Controls\X Listbox\` folder, create a new subfolder named `Abilities`.
   - ❑ In the Project Explorer, right-click **My Computer** and select **New» XControl**.

Note You can also create an XControl from the template browser.

   - ❑ Right-click the XControl and select **Save»Save**.

- ❑ When prompted, click **Yes** to save the new unsaved files of the XControl.
- ❑ Save each of the following XControl files as follows:

Table 4-1. XControl File Names and Locations

XControl File	New Name	Folder
Facade	`X Listbox Facade.vi`	`...\Abilities`
State	`X Listbox State.ctl`	`...\Abilities`
Data	`X Listbox Data.ctl`	`...\Abilities`
Init	`X Listbox Init.vi`	`...\Abilities`
XControl	`X Listbox.xctl`	`...\X Listbox`

Note An XControl is a special kind of project library. If you must rename or move folders, do so within the XControl project. Renaming or moving files in Windows Explorer breaks the XControl.

End of Exercise 4-1

Exercise 4-2 Modifying X Listbox Abilities (Optional)

Goal

To modify the Data, State, and Init abilities of the X Listbox control you created in Exercise 4-1.

Design

Modify X Listbox Data.ctl

- This component specifies the data type of the XControl, in this case, the string data type.

Modify X Listbox State.ctl

- This component specifies any information other than the Data of an XControl that affects the appearance of the control. In this exercise, the component includes an I32 labeled as `# Items` and an array of strings labeled as `Names`.

Modify X Listbox Init.vi

- LabVIEW calls the Init ability when the XControl is first placed on a front panel or when a VI that contains the XControl is loaded into memory. You can use this ability to initialize the display state before the XControl is displayed. In this exercise, you rearrange the front panel to display only the Default Control State and Default Indicator State.

Implementation

Modify the Data, State, and Init abilities of the XControl.

1. Modify the Data ability. This component specifies the data type of the XControl, in this case, the string data type.

   ❑ Open `X Listbox Data.ctl`.

   ❑ Delete the control.

   ❑ Add a string control.

   ❑ Label the string control as `Item`.

Your control should resemble Figure 4-1.

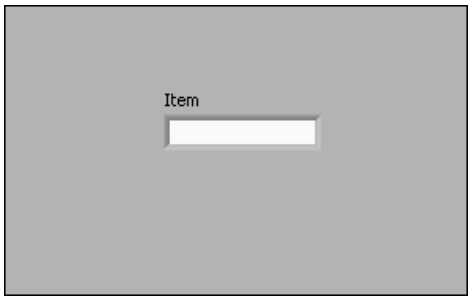


Figure 4-1. X Listbox.xctl

❑ Save and close the control.

2. Modify the State ability. This component specifies any information other than the Data of an XControl that affects the appearance of the control.

   ❑ Open `X Listbox State.ctl`

   ❑ Relabel the numeric control inside the cluster to `# Items`.

   ❑ Set the representation to I32.

   ❑ Add an array of strings.

   ❑ Label it `Names`.

   Your control should resemble Figure 4-2.

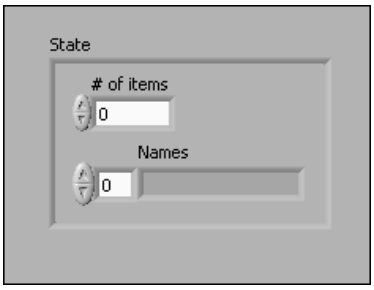


Figure 4-2. X Listbox State Control

   ❑ Save and close the control.

3. Modify the Init ability. LabVIEW calls the Init ability when the XControl is first placed on a front panel or when a VI that contains the XControl is loaded into memory.

   ❑ Open `X Listbox Init.vi`.

- ❑ Scroll down the front panel until you find the Default Indicator State and the Default Control State.

Tip Some controls on the front panel window are locked to prevent accidental deletion, which would destroy the VI. You can unlock and move the controls if you want.

- ❑ Move the controls and resize the front panel to display only the items shown in Figure 4-3.

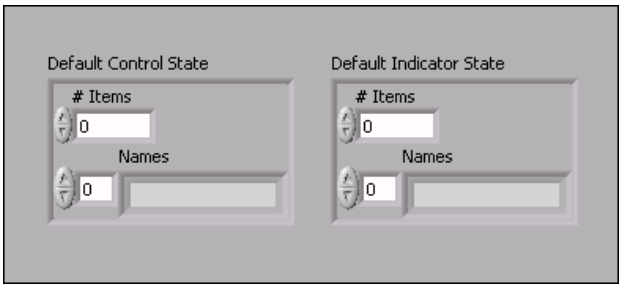


Figure 4-3.

- ❑ Save and close the VI.

End of Exercise 4-2

Exercise 4-3 Creating X Listbox Properties and Methods (Optional)

Goal

To create XControl properties and methods.

Scenario

You can create properties for an XControl in which one attribute or element of the display state is modified. In such cases, the property you create has one parameter associated with it. You use methods in cases when no parameters are needed (as in the Delete All method) or when more than one parameter is needed. Additionally, invoking a method implies that an action is taken, such as adding an item and deleting an item.

Design

- Create the Delete All Items method.
- Create the Delete Item from Listbox method.
- Create the Add Item to Listbox method.

Implementation

In this exercise, the data type of the X Listbox control is a string, specifically the item name of the active row. Users of the XControl must be able to add an item, delete an item, and delete all items. When complete, the front panel for the test VI resembles Figure 4-4.

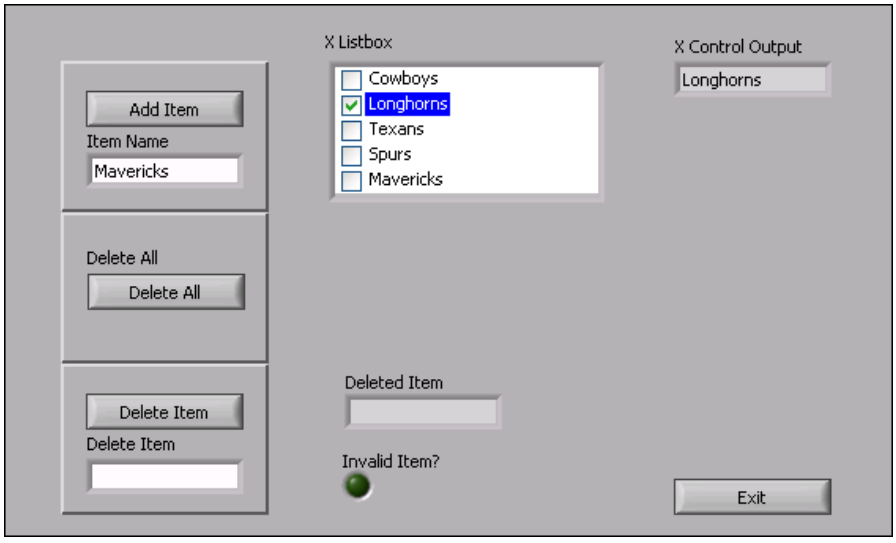


Figure 4-4. TEST X Listbox Front Panel

Remember that the X Listbox State contains the elements that you modify with the methods you create. When the parent VI that holds the XControl calls a method or property, the corresponding Facade VI executes the Display State Change event. After you create these methods, proceed to Exercise 4-4 to modify the Display State Change event case to handle the data modified by the methods.

1. Create the Delete All Items method.

   - ❑ Right-click `X Listbox.xctl` and select **New»Method**.
   - ❑ Right-click the new method and select **Save**.
   - ❑ Save the VI as `Delete All Items.vi` in the `<Exercises>\Advanced Architectures in LabVIEW\X Controls\X Listbox\Methods` directory.
   - ❑ Right-click the `Delete All Items.vi` and select **Configure Method**.
   - ❑ View the settings. No changes must be made for this method because it does not have any inputs or outputs.
   - ❑ Click **OK** to close the Configure Method dialog box.
   - ❑ Open the block diagram of `Delete All Items.vi`.
   - ❑ Build the block diagram as shown in Figure 4-5.

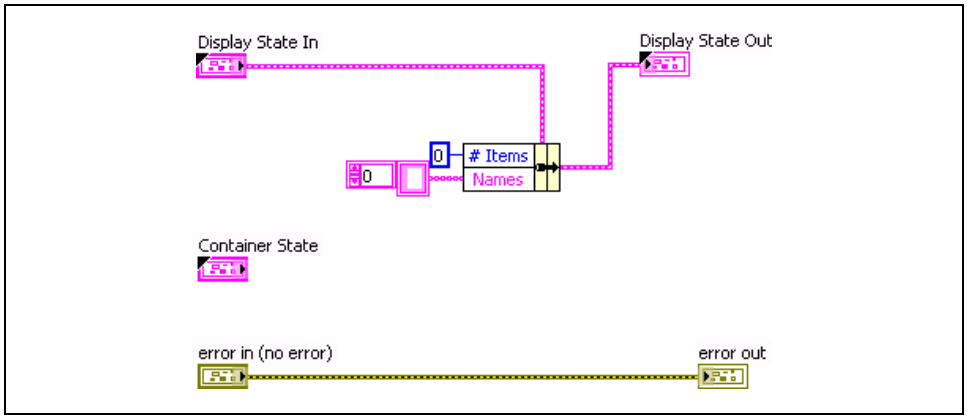


Figure 4-5. Delete All Items VI Block Diagram

   - ❑ Save and Close the VI.
   - ❑ Save `X Listbox.xctl`.

2. Create the Delete Item From Listbox method.

   ❑ Right-click `X Listbox.xctl` and select **New»Method**.

   ❑ Right-click the new method and select **Save**.

   ❑ Save the VI as `Delete Item From Listbox.vi` in the `<Exercises>\Advanced Architectures in LabVIEW\X Controls\X Listbox\Methods` directory.

   ❑ Open the block diagram of `Delete Item From Listbox.vi`.

   ❑ Build the diagram of the VI as shown in Figure 4-6.

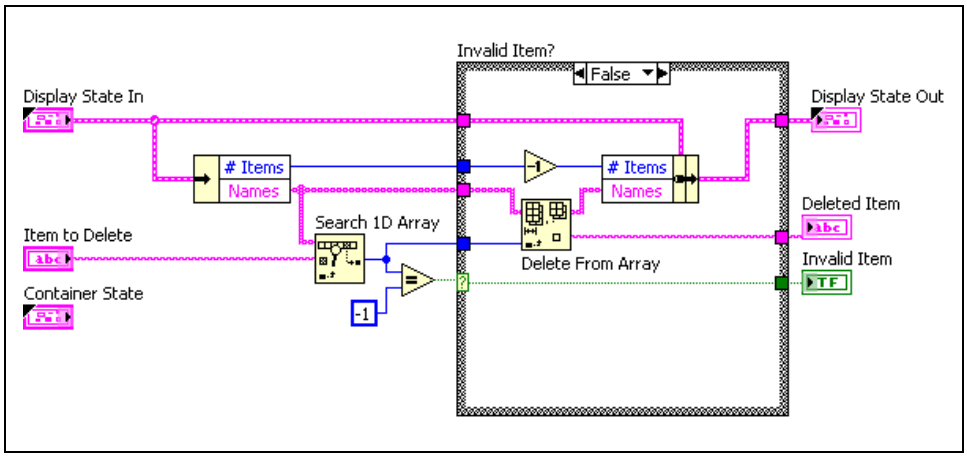


Figure 4-6. Delete Item From Listbox VI Block Diagram - False Case

❑ Wire the True case as shown in Figure 4-7

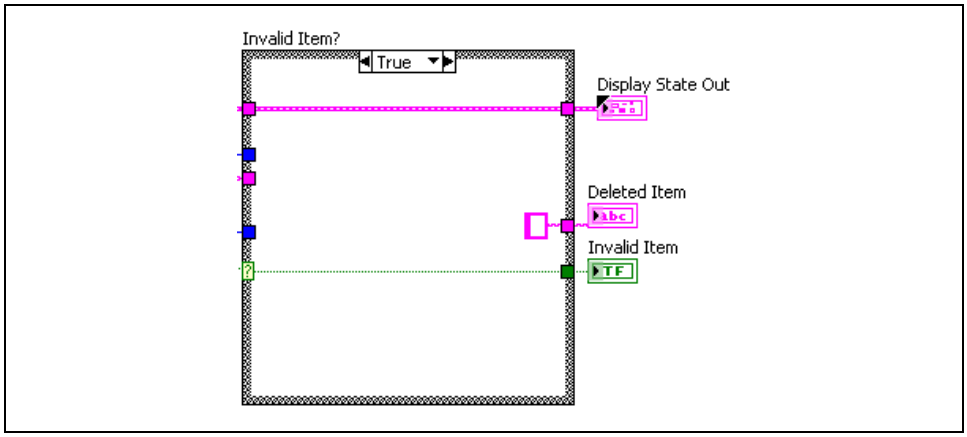


Figure 4-7. Delete Item From Listbox VI Block Diagram - True Case

- ❑ Wire **Item to Delete**, **Deleted Item**, and **Invalid Item?** to appropriate connectors on the connector pane.
- ❑ In the project explorer, right-click the `Delete Item From Listbox.vi` and select **Configure Method**.
- ❑ Add and configure the parameters for the input and each output, as shown in Figure 4-8.

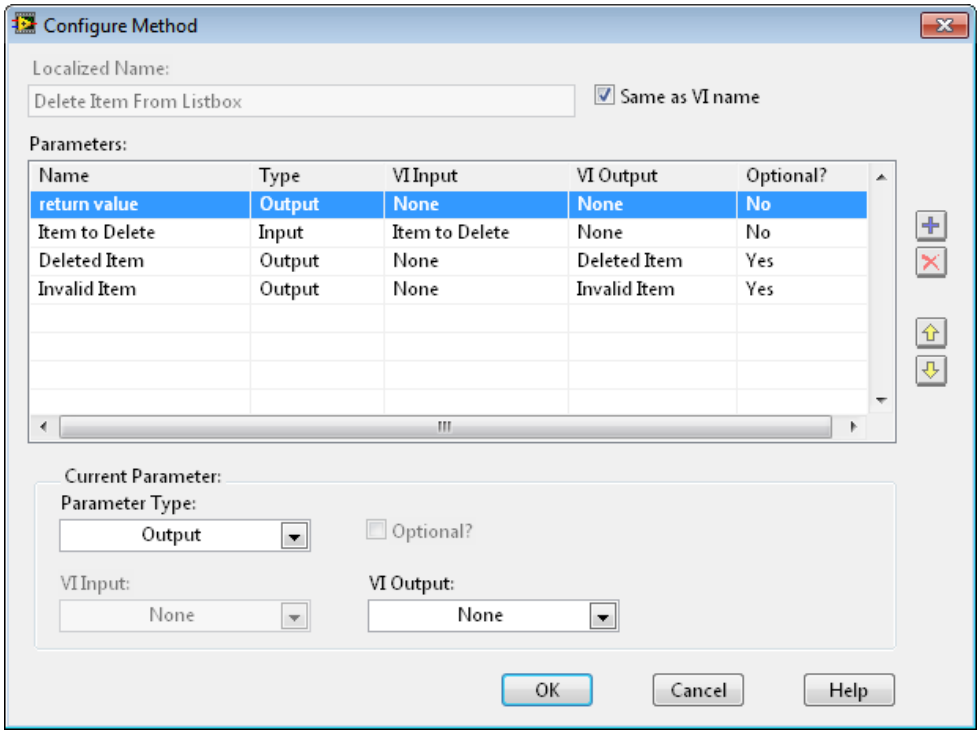


Figure 4-8. Configure Method Dialog for Delete Item From Listbox VI

Note If you did not define connector pane terminals, the above items will not be available. Each item added in the Configure method is a parameter for the XControl method.

- ❑ Select **OK** to close the Configure Method dialog box.
- ❑ Save and close the VI.
- ❑ Save `X Listbox.xctl`.

3. Create the Add Item To Listbox method.
   - ❑ Right-click `X Listbox.xctl` and select **New»Method**.
   - ❑ Right-click the new method and select **Save**.

- ❑ Save the VI as `Add Item To Listbox.vi` in the `<Exercises>\Advanced Architectures in LabVIEW\X Control\X Listbox\Methods` directory.
- ❑ Open the block diagram of `Add Item To Listbox.vi`.
- ❑ Add a string control to the front panel. Rename the control as `Item Name`.
- ❑ Build the diagram of the VI as shown in Figure 4-9.

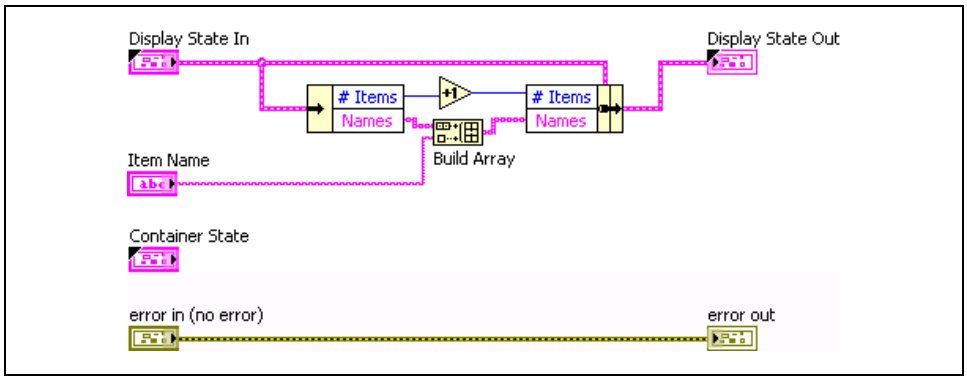


Figure 4-9. Add Item to Listbox VI Block Diagram

- ❑ Wire **Item Name** to an appropriate connector on the connector pane.
- ❑ Right-click the `Add Item To Listbox.vi` and select **Configure Method**.

❑ Add a parameter for the input, as shown in Figure 4-11.

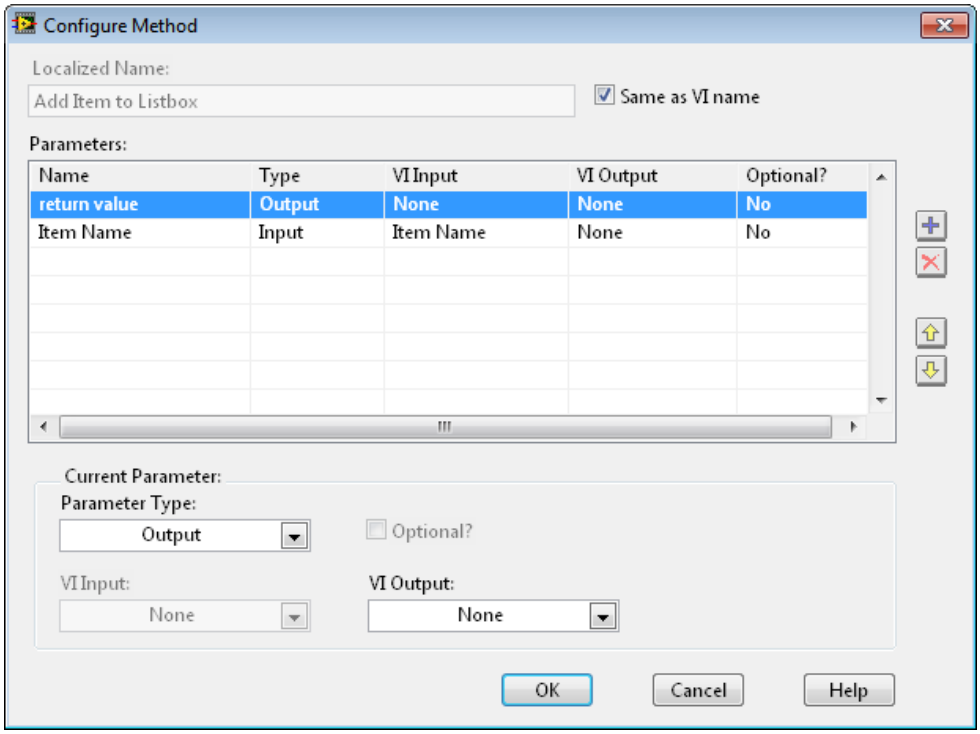


Figure 4-10. Configure Method Dialog for Add Item To Listbox VI

❑ Click **OK** to close the Configure Method dialog box.

❑ Save and close the VI.

❑ Save `X Listbox.xctl`.

End of Exercise 4-3

Exercise 4-4 Creating the X Listbox Facade VI (Optional)

Goal

To create the Facade VI.

Design

Create the front panel window

- The front panel window should include a Listbox that is sized to include many items. The Fit Control to Pane option should be enabled.

Implement the Display State Change event

- This section should update the properties of the X Listbox so that new items are added and others are deleted. The `X Listbox Update for Add and Delete.vi` can be used here.

Implement the Exec State Change event

- The number of rows and the item names of the listbox should be updated based on the display state data.

Create a Listbox: Mouse Down event

- The item symbols should be updated depending upon the active listbox item. The `X Listbox Update Symbols for Select.vi` can be used here.

Organize the X Control library.

Test the VI using the TEST X Listbox VI that has been provided.

Implementation

1. Modify the Facade front panel.
   - ❑ Open `X Listbox Facade.vi`.
   - ❑ Delete the comment on the front panel window.
   - ❑ Add a listbox to the front panel window.
   - ❑ Right-click the listbox control and select **Visible Items»Label** to hide the label.

Note The XControl has its own label when you drop it onto a front panel.

- ❑ Right-click the Listbox control and select **Visible Items»Symbols** to make them visible. This gives the Listbox a radio button feel.
- ❑ Right-click the Listbox control and deselect **Visible Items»Vertical Scrollbar** to remove the scrollbar.
- ❑ Size the Listbox control and the front panel window so that the listbox control takes up the entire window and exactly matches the window borders, as shown in Figure 4-11.
- ❑ Right-click and select **Fit Control to Pane**.

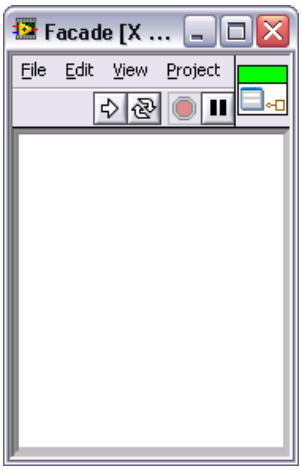


Figure 4-11. Listbox Control Window

2. Modify the Display State Change event.

Note The Display State Change event is generated on the Facade VI when the display state of the XControl changes. This can be in response to a property or method call, or from the Init ability. This event is not generated when other Facade events set the State Changed? element of action. At any given time, the appearance and behavior of the XControl should depend only on the display state, and therefore the Display State Change event must apply any changes in the state to the appearance or behavior of the XControl.

❑ Complete the Display State Change event, as shown in Figure 4-12, to update the appearance of the X Listbox XControl when its properties and methods modify the Display State.

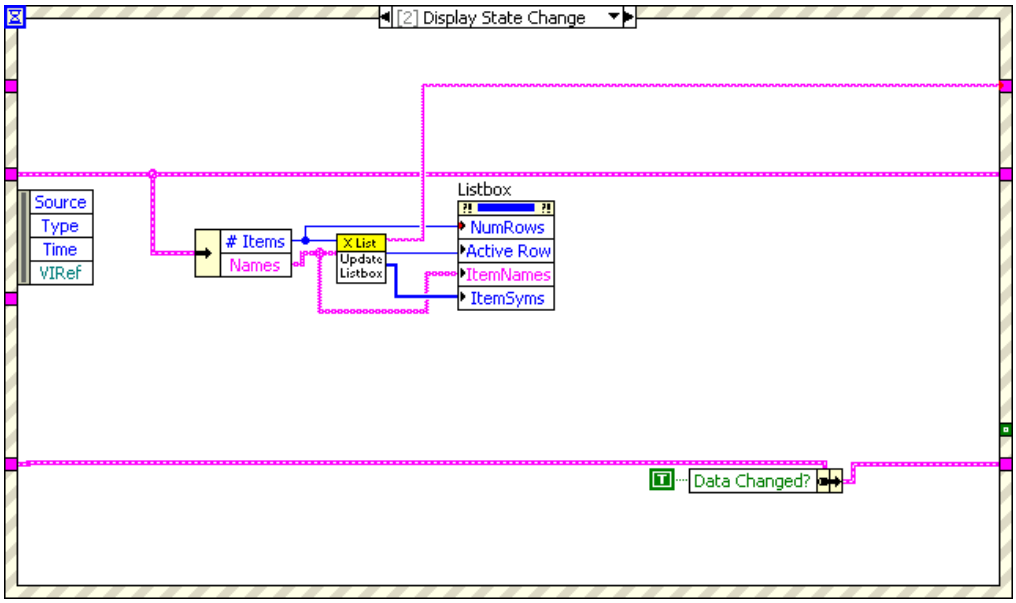


Figure 4-12. Display State Change Event Case

Note The `X Listbox Update for Add and Delete.vi` is located in the SubVIs directory of the X Listbox Control Project.

The number of the event case does not relate to the functionality of the event structure; it specifies the order of the event case when you view it.

3. Modify the Exec State Change case.

Note The Exec State Change event is generated on the Facade VI when a VI containing the XControl changes from edit mode to run mode, or vice versa. You can use this event to modify the control for any differences between its edit-time and run-time behavior.

❑ Complete the Exec State Change event, as shown in Figure 4-13.

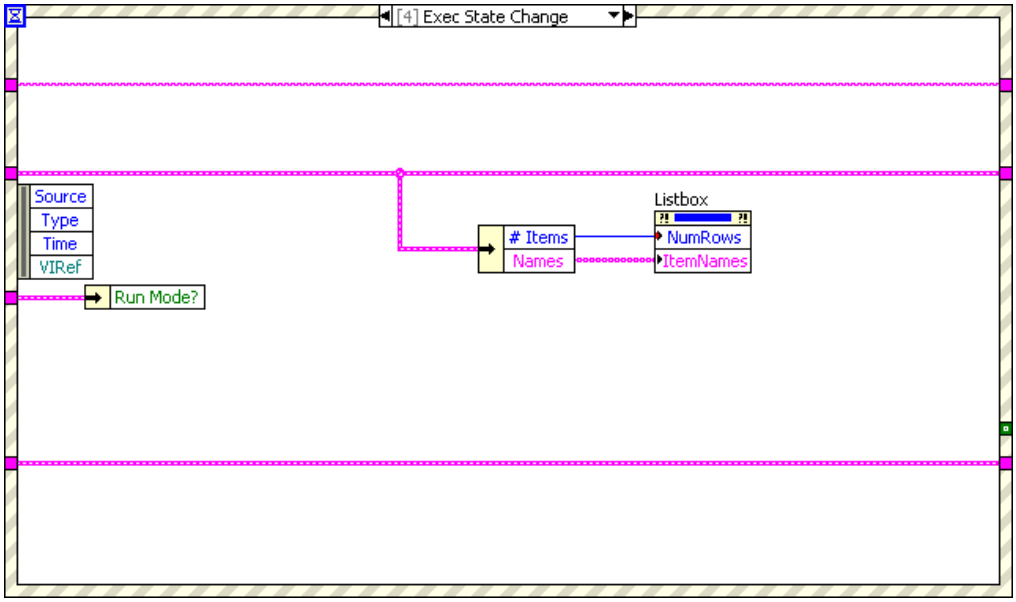


Figure 4-13. Exec State Change Event Case

4. Add the code for a mouse down event on the listbox.

   Remember that you can add any event for the controls on the Facade of your XControl.

   The XControl template adds only four new event cases to those that are already available for controls and VIs.

   ❑ Add a `"Listbox":Mouse Down` event.

❑ Complete the block diagram, as shown in Figure 4-14, to update the appearance of the X Listbox XControl when the user selects a row using the mouse.

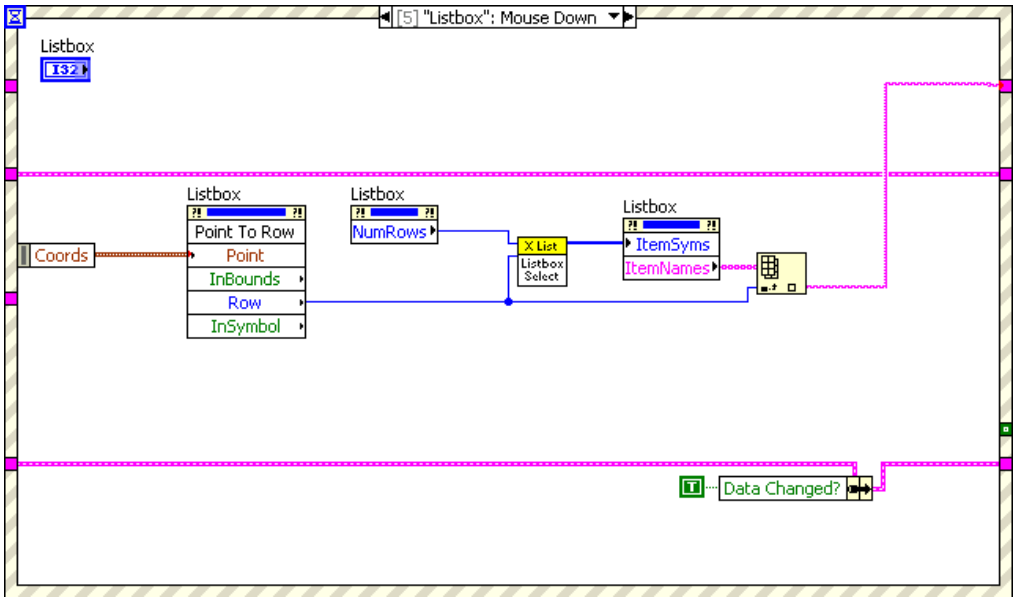


Figure 4-14. "Listbox": Mouse Down Event Case

Note The `X Listbox Update Symbols for Select.vi` is located in the SubVIs folder in the X Listbox Control.lvproj.

5. Save and close the Facade VI.

6. Organize the X Listbox library.

   ❑ Save the `X Listbox.xctl`.

   ❑ In the Project Explorer, add two virtual folders under the `X Listbox.xctl` item: `Abilities` and `Methods`.

   ❑ Drag the `X Listbox Data.ctl`, `X Listbox State.ctl`, `X Listbox Facade.vi` and `X Listbox Init.vi` into the `Abilities` virtual folder.

   ❑ Drag the `Delete All Items.vi`, `Delete Item from Listbox.vi`, and `Add Item to Listbox.vi` into the `Methods` virtual folder.

   ❑ Drag the `SubVIs` virtual folder under the `X Listbox.xctl` item.

❑ Select **File»Save All**. Your project should resemble Figure 4-15.

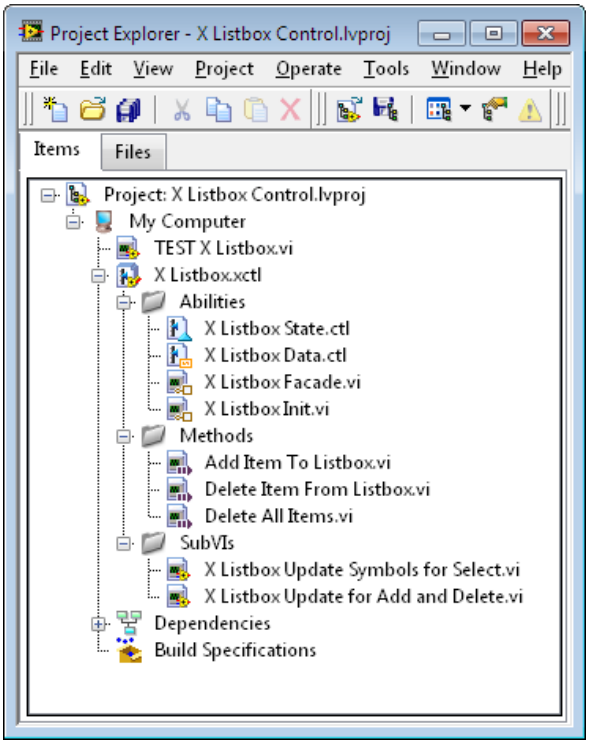


Figure 4-15. X Listbox Control Project

❑ Close the XControl.

7. Test the XControl.

   The X Listbox Control project already contains the Test VI stub.

   ❑ Open `TEST X Listbox.vi`.

   ❑ Drag the `X Listbox.xctl` from the project onto the front panel.

   ❑ Open the block diagram.

❑ Wire the X Listbox control to the X Control Output control and edit the event handled by this case, as shown in Figure 4-16.

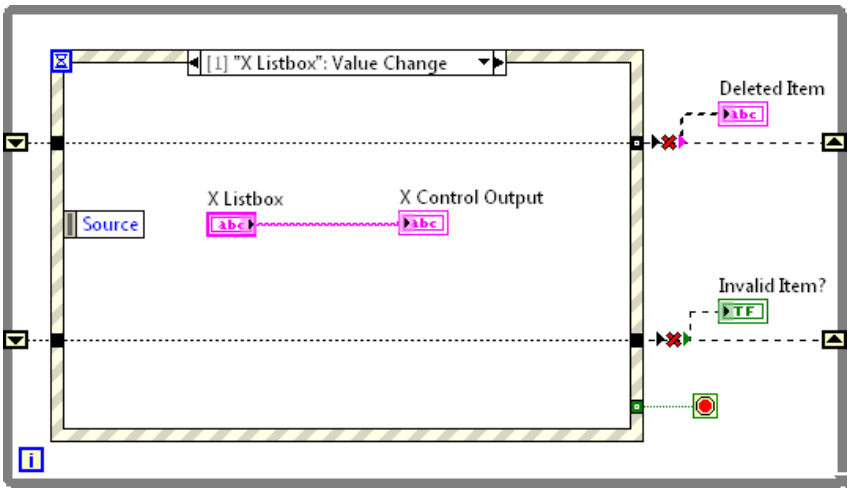


Figure 4-16. X Control Output Control

Note Wires on the block diagram appear broken until you wire the `"Delete Item": Value Change` case.

❑ Create the methods from the X Control.

– Right-click the X Control terminal and select **Create»Invoke Node»Delete All Items** to create the method. Add to the corresponding event, as shown in Figure 4-17.

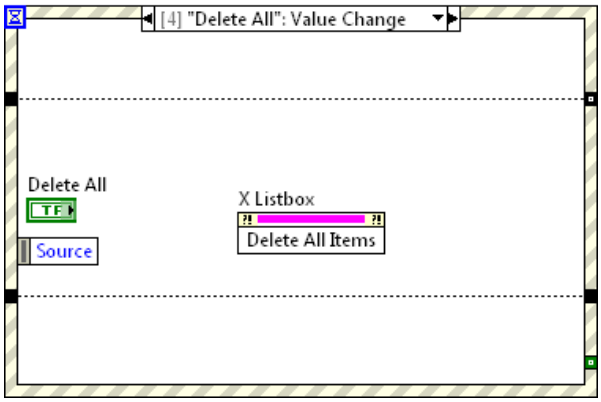


Figure 4-17. X Control Delete All Items Event

– Right-click the X Control terminal and select **Create»Invoke Node»Delete Item from Listbox** to create the method. Add to the corresponding event, as shown in Figure 4-18.

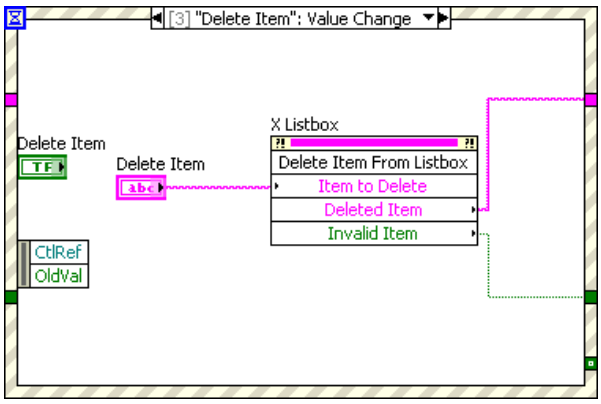


Figure 4-18. X Control Delete Item From List Box Event

– Right-click the X Control terminal and select **Create»Invoke Node»Add Item to Listbox** to create the method. Add to the corresponding event, as shown in FigureX Control Add Item to Listbox Event 4-19.

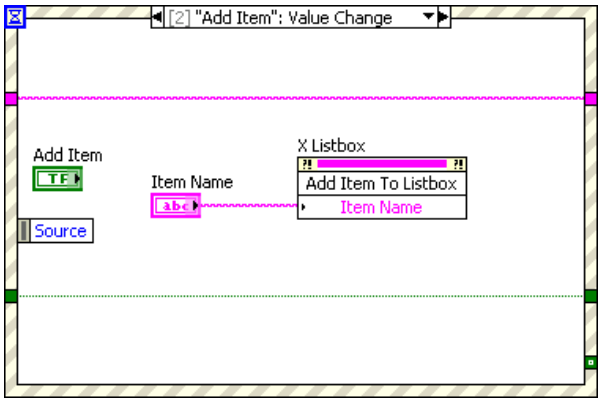


Figure 4-19. X Control Add Item to Listbox Event

❑ Save the VI.

8. Test the VI.

❑ Add items to the listbox.

❑ Delete items from the listbox.

❑ Delete all items from the listbox.

❑ Stop the VI with items still in the listbox.

- ❑ Run the VI again to ensure items remain.
- ❑ Save and close the VI.

Note In order for state data to be saved with a parent VI, a Convert for Save ability must be added to the XControl. No modifications need to be made.

- ❑ Save and close the project.

Challenges

- ❑ As it is currently coded, the X Listbox operates in the following manner. When a user adds or deletes an item, the last item becomes the active item. This may not be the cleanest solution when the user deletes an item in the middle of the list. Modify the X Listbox so that upon deleting an item, the active row remains the same as the row that was just deleted. You must modify the `X Listbox State.ctl` as well as the `X Listbox Facade.ctl`.
- ❑ Modify the `X Listbox Init.vi` so that it reads an `*.ini` to populate the initial values of the item names.
- ❑ Modify the appearance of the check boxes to be radio buttons. Hint: this will involve copying an image to the symbol table.
- ❑ Modify Add Item to Listbox to check for duplicate names.
- ❑ Add an `Add Items to Listbox.vi` to add an array of names at one time.

End of Exercise 4-4

Notes

Notes

5

Introduction to Object-Oriented Programming in LabVIEW

Exercise 5-1 LVOOP

Goal

To add a sibling class to an existing class.

Scenario

You are given an existing hierarchy of LVOOP of classes. The parent class is the report class. The existing child classes are the XML class and the TXT class. These are sibling classes. From the TXT class, you create a TDMS class and test its functionality.

Code is already written to interface with the parent and existing sibling classes. Adding a new class with the same interface as the other classes is very simple task. You do not need to fully understand native LabVIEW classes (LVOOP) to complete this exercise, but you see how classes add flexibility and needed structure in an architecture. Use this as a starting point for creating reports in any application.

Design

- From the LVOOP Report project, you create a new sibling class from the `TXT Report.lvclass`.
- Make the `Report.lvclass` the parent.
- Modify the Write to File to generate the TDMS file. The Write TDMS File VI is provided for you.
- Add the TDMS Report Init.vi to the TEST Report Class VI to test your code.

Implementation

1. Observe the class hierarchy.

   - ❑ Open the `LVOOP Report.lvproj`, located in the `<Exercises>\Advanced Architectures in LabVIEW\LVOOP` directory.
   - ❑ Select **View»LabVIEW Class Hierarchy** to observe the relationships of the classes, and then close the LV Class Hierarchy window.

2. Making a copy of an existing class in order to create a sibling class.

   - ❑ Navigate back to the `LVOOP Report.lvproj`.
   - ❑ In the project explorer window, right-click the `TXT_Report.lvclass` file located in the **Plug Ins** virtual folder, and select **Save»Save As**.
   - ❑ Select **Copy** and name the class `TDMS Report.lvclass`.
   - ❑ Verify that the **Add copy to LVOOP Report.lvproj** checkbox is enabled.
   - ❑ Click **Continue**.
   - ❑ Navigate to `<Exercises>\Advanced Architectures in LabVIEW\LVOOP\Plug Ins` and create a new directory called `TDMS Report Class`.
   - ❑ Navigate to the newly created `TDMS Report Class` directory and click **Current Folder** in the file dialog box to save the copy.
   - ❑ In the project explorer, move `TDMS Report.lvclass` to the Plug Ins virtual folder.

3. Modify the icon to associate with the TDMS Report class.

   - ❑ Right-click `TDMS Report.lvclass` and select **Properties**.
   - ❑ In the General Settings, click **Edit Icon**.
   - ❑ Change the Icon text to `TDMS` instead of `TXT`.
   - ❑ Click **OK** to close the Icon Editor dialog.

- ❑ Click **OK**.
- ❑ In the Apply Icon Changes to Member VIs? dialog box, click **Yes**.

4. Modify the TDMS Report class to generate TDMS files.
   - ❑ In the project explorer, rename the `TXT Report_Init.vi` under `TDMS Report.lvclass` to `TDMS Report_Init.vi`.
   - ❑ Open the block diagram of `TDMS Report_Init.vi`.
   - ❑ Change the label on the object out to `TDMS Report out` from `TXT Report out`.
   - ❑ Save and close the VI.
   - ❑ In the `TDMS Report.lvclass`, open the `Write to File.vi`.
   - ❑ On the labels for the objects, change `TXT` to `TDMS`.
   - ❑ On the block diagram, replace the `Write TXT File.vi` with `Write TDMS File.vi` which is located in the SubVIs directory of this exercise.
   - ❑ Save and close the VI
   - ❑ Open the block diagram of `Set Extensions.vi`.
   - ❑ Go to the No Error case and change the string constant text from `txt` to `TDMS`.
   - ❑ On the front panel, change the names of the objects to reflect the TDMS report type.
   - ❑ Save and close the VI.

Testing

1. Test the changes.
   - ❑ In the project explorer, open the `TEST Report Class.vi`.
   - ❑ On the block diagram, replace the existing `Init.vi` with the `TDMS Report_Init.vi`. Notice the automatic selection of the Write TDMS.
   - ❑ Run and confirm that the TDMS file has been created.
   - ❑ Save and close the VI.

Note To view the contents of the TDMS file, run the `Read TDMS_Report.vi` located in the SubVIs directory of this exercise.

2. Learn more about XML and objects.

   ❑ Open the XML Report class.

   ❑ Investigate the Write to File method.

   ❑ What is different about this implementation?

   ❑ How could you utilize XML in your application?

Challenge

Review the challenges you listed in Exercise 1-3. Look for potential modules in your application in which a parent-child relationship exists. Draft an initial class diagram.

Notes

Notes

6

Plug-In Architectures

Exercise 6-1 Using Plug Ins with VI Server

Goal

To evaluate the key components of a plug in architecture implemented with VI Server and to add a plug in to the existing architecture.

Scenario

Though the architect cannot foresee all potential changes to an application, some changes can easily be identified and the architect can plan in advance to adapt. Often, the change entails adding a component that is of a similar type to another component. For instance, the structure for saving data to a file can change. Data may initially be saved in a binary file and then later the application will save data to a TDMS file.

The interface for saving the file, regardless of its type, can be defined early the design stage. The application calls the interface, which in turn calls specific binary writer code. Then, any additional file format writers can be added by dropping the new plug in VI into a specified directory. The interface will recognize the plug in and as such make it available to the main application.

This example was designed with plug ins that are different waveform generation VIs. Though waveform generation may not be a feature of your application, the structure for finding and calling the plug ins can immediately be used in your application.

Implementation

Add features to a running VI without stopping the VI.

1. Evaluate the existing VI.

   - ❑ Open `Plug Ins with VI Server.lvproj` located in the `<Exercises>\Advanced Architectures in LabVIEW\Plug Ins with VI Server` directory.

   - ❑ Open `Plug Ins Main.vi` from the Plug Ins with VI Server project.

❑ Run the VI and observe the two waveform options from the Waveform Selector.

❑ Stop the VI and explore the case structure as follows:

– Initialize case

Note The control terminals are used in the event structure and, as such, the local variables must be used here. Because this is a VI in which the User Interface is visible, this is an acceptable use of local variables.

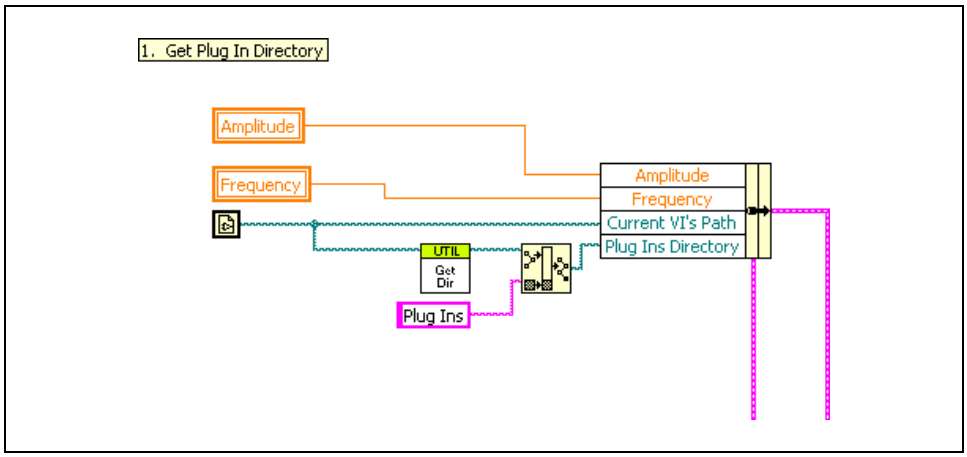


Figure 6-1. Plug Ins Main VI - Initialize Case

– Get Plug Ins case

Open the Create WFG Plug In List subVI. Observe how the list is created. The VIs that have the same connector pane as the strictly typed VI will be opened and not have an error. For those VIs, name value pairs are created from the window title and the VI path.

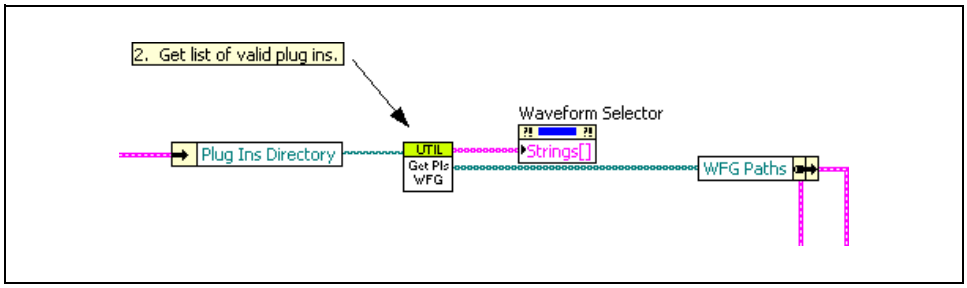


Figure 6-2. Plug Ins Main VI - Get Plug Ins Case

– "No Action" case, Event structure "Waveform Selector":Value Change event

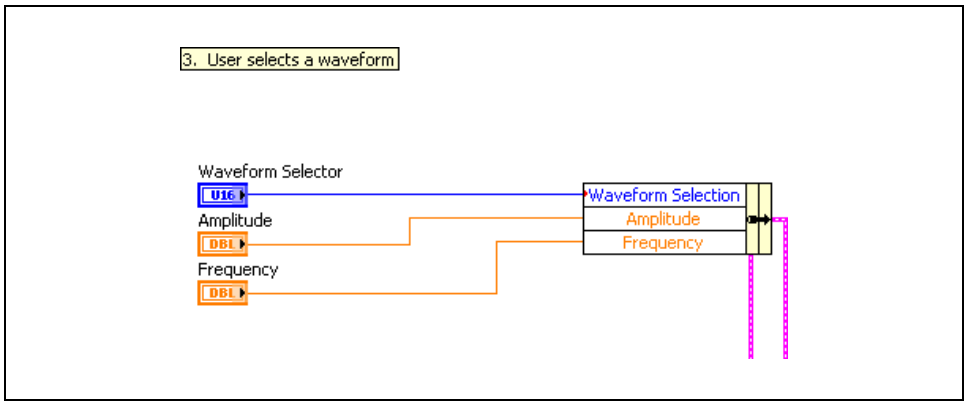


Figure 6-3. Plug Ins Main VI - No Action Case

– Waveform Selector case

Open the Call WFG Plug In subVI. Notice that the connector pane must remain the same for any plug in that will be called by this VI.

Tip When designing plug ins, you may want consider using extra terminals of the string or variant data type to facilitate modifications to the code.

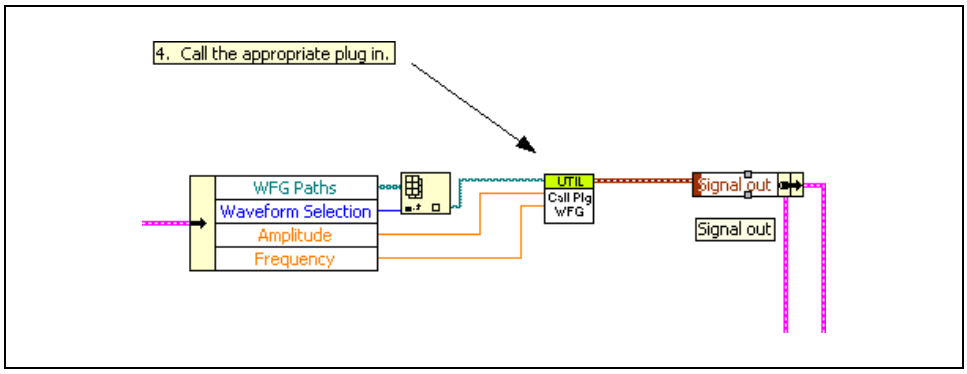


Figure 6-4. Plug Ins Main VI - Waveform Selector Case

❑ Return to the front panel.

❑ In the project explorer, explore the `Plug Ins` directory where the plug in VIs are stored. In the remainder of this exercise, you create a plug in and then save it to this directory while `Plug Ins Main.vi` is running.

2. Add a plug in while the VI is running.

   Move and size the front panel so that you can view the running VI and the project at the same time.

   ❑ Run the Plug Ins Main VI.

   ❑ From the Plug Ins with VI Server project, expand the SubVIs folder.

   ❑ Right-click `Template Plug In.vit` and select **New From Template**.

   ❑ Modify the icon to say `Square`.

   ❑ From VI Properties, select **Window Appearance**.

   ❑ Change the Window Title to `Square Waveform`.

   ❑ On the block diagram, right-click the subVI and select **Replace» Waveform Generation Palette»Square Waveform.vi**.

   ❑ Save the VI in the `<Exercises>\Advanced Architectures in LabVIEW\Plug Ins with VI Server\Plug Ins\Square Plug In.vi`.

Note The new VI must be saved in the Plug Ins folder for the example to work correctly.

   ❑ Close the VI. The VI flashes because it is already linked into the main application.

   ❑ Return to the still running Plug Ins main and notice that the user now has the option to create a Square Waveform.

Challenge

Create a build specification and stand-alone EXE that will have the same plug-in functionality as seen earlier in this exercise.

1. Create the stand-alone application.

   ❑ Create and configure a build specification. The stand-alone application should look for plug-ins inside a `Plug Ins` folder.

   To use a pre-configured build specification instead, open `<Solutions>\Advanced Architectures in LabVIEW\Exercise 6-1\Plug Ins with VI Server.lvproj`.

- ❑ Build the stand-alone EXE.
- ❑ Move the `Square Plug In.vi` to a location other than the `Plug Ins` directory.
- ❑ Run the stand-alone EXE.
- ❑ Copy the `Square Plug In.vi` to the Plug Ins directory to confirm that it gets linked into the code while the stand-alone EXE is running.

2. Evaluate the solution.

   What are the restrictions on using this Plug In architecture?

   What must be considered at design time in order to facilitate future scalability?

   What other considerations could come into play with this advanced design pattern?

End of Exercise 6-1

Exercise 6-2 Using LVOOP Plug Ins

Goal

To write the calling code to allow new LVOOP sibling classes to be plugged in at run time.

Scenario

This is an advanced feature that will likely take time to fully grasp. It is included because it could be useful for you now or at a later date. Additionally, the details for the implementation were published in the LabVIEW community at `www.expressionflow.com`.

Implementation

This exercise illustrates the ease of extending an application using native LabVIEW classes. You are given the stub VI and a list of the VIs that you need to add to the application. You must decipher how to put the VIs together and where to find them on the palettes.

1. Open the VI.
   - ❑ Open the LVOOP Report project located in the `<Exercises>\Advanced Architectures in LabVIEW\LVOOP` directory.
   - ❑ Open the `TEST LVOOP Plug Ins.vi` from the `Plug Ins` folder of the project explorer.
2. Explore the VI.
   - ❑ Navigate to the **Save to File** case.
   - ❑ Observe the block diagram and the use of the following VIs.
     - – Set Extensions
     - – Set Path Name
     - – Set Header and Data
     - – Write to File

3. Modify the VI.

   ❑ Navigate to the **Get Plug Ins** case.

   ❑ Add the following items to the block diagram:

   – Report object constant

   – Array of report object constants

   – Get LV Class Default Value VI

   – To More Specific Class function

These general steps help you complete the block diagram. Arrows and notes on the block diagram assist you in the placement of the VIs.

- Extract all files from the Plug Ins folder that have the `.lvclass` extension.
- Initialize an array of report object constants.
- Add all objects from the Plug Ins folder that are also in the report class hierarchy to the array of report objects.
- To screen for objects that are in the report class hierarchy, you can try to convert the object to the more specific class of report. If no error occurs, then the object is part of the hierarchy.

Note Objects that are in the same object class hierarchy can be bundled into arrays and can be wired to any terminal whose data type is in the same hierarchy. Thus, objects facilitate the design of flexible and scalable software architectures.

❑ Wire the items you added to the block diagram.

Testing

1. Run and test the VI.
2. Save and close the VI.

Challenge

Create the build application for the project.

End of Exercise 6-2

Notes

7

Tips, Tricks, & Other Techniques

Exercise 7-1 Using Drop Ins

Goal

To create a drop-in VI that logs all Boolean value change events to a file.

Scenario

As the project architect, part of your job is identifying coding tasks that you repeat over and over again. Additionally, you may identify a feature that you want to include with a type of control in LabVIEW. By extracting the references of controls of a certain type for a VI, you can add features that are not currently available. You will use a drop-in architecture to implement a Boolean value change logger, which is a drop-in VI that you can use in any application and expand to add functionality.

Design

- Open `Drop Ins.lvproj`
- Open `DropIn Get Button Clicks - Stub.vi`
- Add `Traverse for GObjects.vi`
- Add **Register for Events** function
- Add **Type Cast** function

Implementation

1. Open the project and drop-in VI.
   - ❑ Open `Drop Ins.lvproj` located in the `<Exercises>\Advanced Architectures in LabVIEW\Tips & Tricks\Drop Ins` directory.
   - ❑ Open `DropIn Get Button Clicks - Stub.vi`. This is the drop-in VI.

2. Enable the VI Scripting palette to access the Traverse For GObjects VI.

   ❑ Select **Tools»Options** and select the **VI Server** category.

   ❑ Enable the **Show Scripting functions, properties and methods** checkbox.

   ❑ Click OK.

3. Modify the DropIn Get Button Clicks - Stub VI block diagram to write to file each time a value change event occurs for any boolean control on the referenced VI, as shown in Figure 7-1, using the following items.

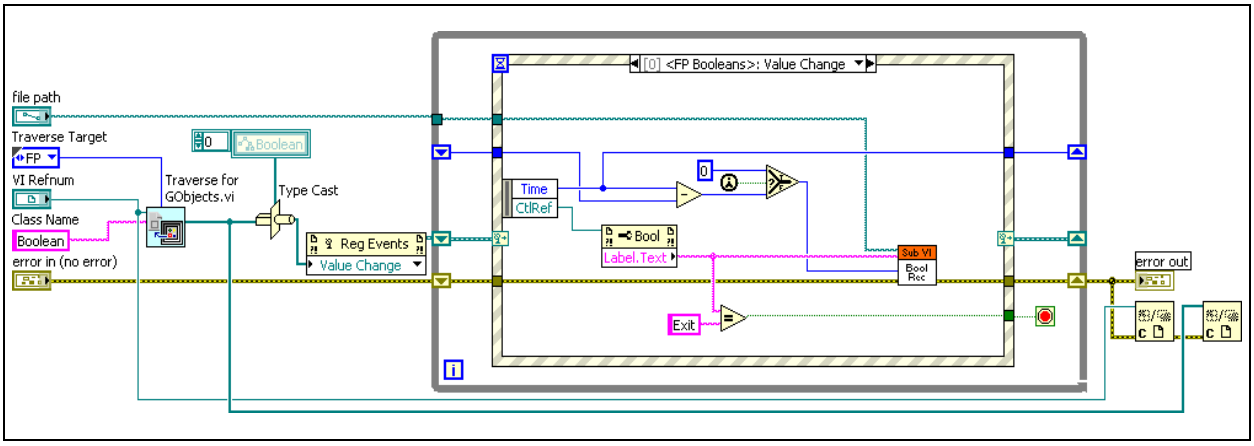


Figure 7-1. DropIn Get Button Clicks - Stub VI Block Diagram

   ❑ Traverse for GObjects VI

      – Right-click the corresponding inputs to create and configure the control and constants shown in Figure 7-1.

      – On the front panel, wire the VI refnum control to the top left connector on the connector pane.

   ❑ Type Cast function

   ❑ FP Boolean Array Constant

      – From the Application Control palette, select the Class Specifier Constant and add it to the block diagram. Click the constant and select **Generic»GObject»Control»Boolean**.

      – Add an Array Constant to the block diagram.

      – Drag the Boolean Reference Constant into the Array Constant.

- ❑ Register For Events function
  - – Wire the output of the Type Cast function to the event source input of the Register For Events function.
  - – Click the Register for Events function and select **Value Change**.
- ❑ Wire items from left to right as shown in Figure 7-1.

4. Save the VI.

Testing

Test the drop-in VI by placing it in a calling VI.

1. From the Project Explorer, open the TEST Drop In VI.
2. Modify the TEST Drop In VI.
   - ❑ Add the DropIn Get Button Clicks - Stub VI to the TEST Drop In VI block diagram.
   - ❑ On the Application Control palette, select the VI Server Reference constant and add it to the block diagram. Wire the VI Server Reference to the VI Refnum input of the DropIn Get Button Clicks - Stub VI.
   - ❑ Right-click the **file (use dialog)** input and select **Create»Constant**. Set the path to a valid location.
3. Run the TEST Drop In VI.
4. On the front panel, click the Test 1, Test 2, and Test 3 buttons a few times. Click **Exit**.
5. Navigate to the location you specified in the file (use dialog) constant and open the log file. You should see a list of the buttons that you pressed and when you pressed them.

Challenge

Modify the drop-in VI to capture all interaction with the front panel and classify the type of control in which the value change event occurred.

End of Exercise 7-1

Notes

8

Error Handling

Exercise 8-1 Designing an Error Handling System

Goal

To design and prototype an error handling system that you can use in your daily LabVIEW development.

Scenario

Apply the skills that you learned in the course and to improve code that you are currently designing. Follow the requirements generally, as specifics can be added to or deleted from what is listed. Your main consideration is the broader structure.

Design

Assume that the overall architecture of a project that uses the error handling system may include Queue Driven Message Handlers, or a design similar to the JKI State Machine for controlling all main processes.

The error system might be a separate process that runs concurrently with the main application. However, the error system does not have to be centralized. You could implement a central definition that is utilized in a distributed error handling system.

General Requirements

Note These are not specific requirements. They include broad categories and information to help you define requirements for your application.

Classification

The error handling system classifies all errors. The architect must determine the classification. For example, you can classify errors as warnings, non-critical errors, or critical errors. The classification also implies the action to be taken. Classification and action could be synonymous or differentiated. For instance, an application could flag a data acquisition error that is critical or non-critical. Additionally, a non-critical error could be appropriately handled by a user prompt or an email. Therefore, you should consider the types of actions that may need to occur. Thus you must first determine an appropriate error classification system.

Data Structure

Determine the appropriate data that is required to process various errors. For instance, in a process environment, if a fatal error occurs, information about the lot and the process parameters may be required. If the error is a transient instrumentation error, the pertinent information may be the instrument IDs and a command to reset the instrument. Regardless, the information needed to adequately process an error will exceed the capabilities of the standard error cluster. Moreover, isolating the cause may require retaining historical operational information. Therefore, you should carefully analyze all of the data that might be needed to process errors and consider the types of errors that might be added in the future.

Actions

Basic error logging may entail simply documenting the errors that occur and when they occur in a simple `.txt` file. Actions could include prompting the operator or shutting down the system. Additionally, error notification could extend beyond operator notification and include email, web, or text messages.

Error System Structure

One VI will most likely be insufficient to implement all facets of a comprehensive error handing system. You must determine the key VIs to include and the typical location of those VIs. For instance, a VI that displays a user prompt may reside in the Error case of the QDMH. An initialize case could ensure that an error handling process is indeed running. After each major state transition in the QDMH, an error VI may filter some errors and pass others to the main system. You must also consider the appropriate structure for consolidating this information and conveying it to other portions of the code. This is a critical component of a comprehensive system.

Implementation

Refine the general requirements to better suit your application needs. Considering Classification, Data Structures, and Actions helps you further clarify how the system should respond to various errors that could occur. After you sketch the requirements, draft a software architecture and prototype a system.

1. Clarify Requirements.

   - ❑ Classification
     - – What are the main classifications? Why?
     - – Can you group some of the classifications?
   - ❑ Actions
     - – Determine all potential actions that may occur.
     - – Group similar actions and consider future actions that might be incorporated.
   - ❑ Data Structures
     - – For each classification, identify the data elements required to process the error.
     - – Identify common components. Based on the common components, you may consider an object-oriented approach.

2. Draft a preliminary software requirements specification.

3. Draft an initial software architecture.

   The `QDMH Main.vit` has two VIs that related to dealing with errors. The TLC Main also has an error handling mechanism. Think through your application and determine the main modules that you will need to include and the communication mechanism amongst them.

   - ❑ Define an initial architecture that takes your refined requirements into consideration.

- ❑ Share the ideas with someone else in the class and refine.

4. Prototype

   ❑ Prototype a portion of your design. This may be the overall structure or a component.

End of Exercise 8-1

Notes

A

Course Slides

Topics

A. Software Architecture – Introduction
B. Designing an API
C. Multiple Processes and Inter-Process Communication
D. Advanced User Interface Techniques
E. Introduction to Object-Oriented Programming in LabVIEW
F. Plug-In Architectures
G. Tips, Tricks, & Other Techniques
H. Error Handling

Lesson 1
Software Architecture – Introduction

TOPICS

A. Software Architecture – Quality
B. Documenting a Software Architecture
C. Design Patterns and Scalability

A. Software Architecture – Quality

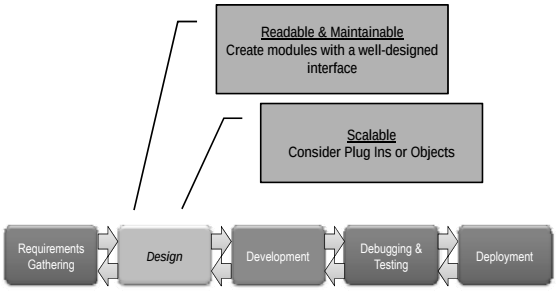
An architecture "…shouldn't look as if the problem and the architecture have been forced together with duct tape..."
– Steve McConnell *Code Complete* p. 52

... so how do we accomplish this?

Obstacles to Well-Designed Code

- "Code & Fix" model of software development is rampant
- Requirements always change
- Developer underestimates the complexity
  "It's just a small data acquisition project"
- Developer is not equipped with the design patterns needed to implement scalable code
- Design is an art and good design requires experience
- Software architecture was never designed or documented

Software Life Cycle – Design for Readability, Maintainability and Scalability



NATIONAL INSTRUMENTS | ni.com/training

What Is Your Software Design Process?

DISCUSSION

Software Life Cycle – Design & Development

- Design and development are only part of the overall process
- During development, consider time required for and the issues related to debugging, testing, deployment, and code updates

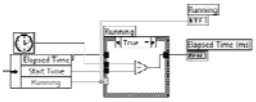
NATIONAL INSTRUMENTS | ni.com/training

Design Goals – Readable

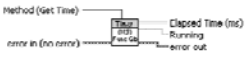
"The majority of the cost of software is incurred after the software has been first deployed. Thinking about my experience of modifying code, I see that I spend much more time reading the existing code than I do writing new code. If I want to make my code cheap, therefore I should make it easy to read." – *Implementation Patterns* by Kent Beck

Design Goals – Maintainable

- Avoid coding the same functionality in more than one location as this reduces maintainability



- Packaging data and functionality into reusable modules minimizes the effect of minor code changes to an overall application



Design Goals – Example of Readable and Maintainable

Best

Advantages
- Know the main responsibility immediately
- Updating one module updates all instances

Okay

Disadvantages
- May be in a cluttered diagram with many other clusters
- Requires remembering the functionality of the code
- Changing functionality may require changes throughout the application

Design Goals – Scalable: N+1

- Create multiple instances of a component (or object)

- Create different components that are part of the same family

Other Design Goals

- Leverage standard coding practices
- Minimize complexity
- Minimize unnecessary code
- Minimize coupling between components
- Design autonomous components
- Plan for deployment and portability

Keys to Quality in a Software Architecture

- Define software architecture and identify architecture components
- Take time to architect the application
- Learn the foundational design patterns
- Create new design patterns as required by your application (requires experience)

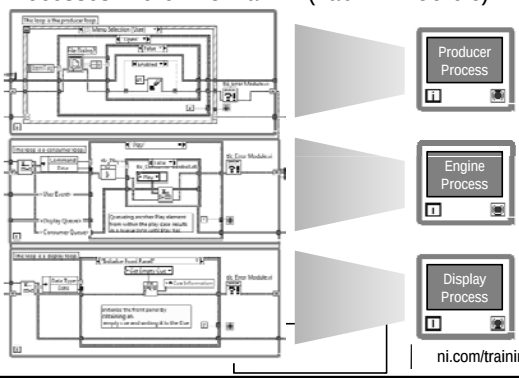
Software Architecture – Definition

Software Architecture–"structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them"– Software Architecture in Practice – 1st ed.

	In LabVIEW
Software Components	Processes SubVIs/Libraries
Properties	APIs for SubVIs and libraries
Relationship	Data communication techniques

NATIONAL INSTRUMENTS | ni.com/training

Processes in the TLC Main.vi (LabVIEW Core 3)



ni.com/training

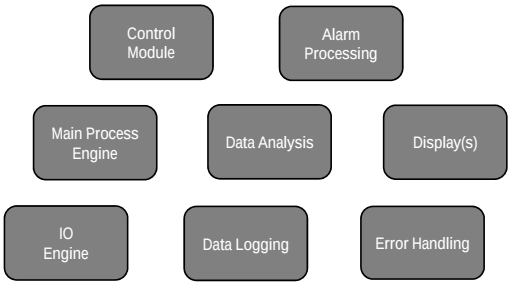
Identifying a Process

Characteristics
- Executes a sequence of steps or actions
- Continuously runs those steps or actions
- Has a specific and singular responsibility

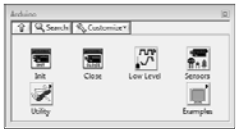
Implementation
- Typically based on Producer/Consumer or Queued Message Handler
- Can include one loop or two. Producer/Consumer may comprise one process

NATIONAL INSTRUMENTS | ni.com/training

Typical Processes in an Application

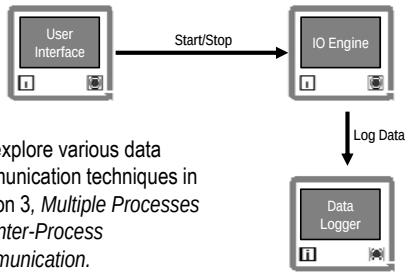


SubVIs and Libraries



The API is the interface to a subVI module or a library of VIs. You learn more about this in Lesson 2, *Designing an API.*

Data Communication Techniques



You explore various data communication techniques in Lesson 3, *Multiple Processes and Inter-Process Communication.*

Software Architecture Course Goals

- Improve the software design process
- Understand advanced design patterns that promote a quality architecture
- See different solutions implemented in the course project
- Be inspired to craft new reference architectures as needed

Software Architecture and Wind Farm Course Project

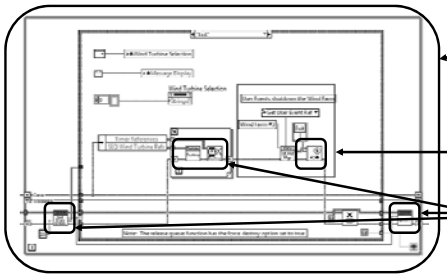
- The course material includes three solutions to the Wind Farm Course Project
- You will learn about the underlying design choices and the trade offs of the different implementation choices
- You will not build the course project
- Exercises are not unique to wind turbine operation
- For every exercise, you will be able to use the solutions or process in your application at your workplace

DISCUSSION

Wind Farm Course Project

- Requires:
  - An unlimited number of processes
  - Communication between those processes
- Reflects common design challenges
- Can be implemented many ways
- Simulated application

Software Architecture Components – Wind Farm Course Project



B. Documenting a Software Architecture

Consider building a house. The blueprints may include many different types of documents. These could include the framing architecture and electrical and plumbing diagrams.

Similarly, a software architecture will include many initial diagrams and other supporting documents that lay out essential implementation details.

Stakeholders of Software Architecture Documents

The documentation may change depending on the stakeholder. The project manager may need less detail or a higher level diagram than the engineering who will conduct black box testing.

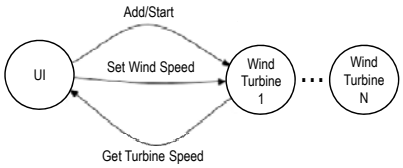
Stakeholders may include:

- Customer
- Project Manager
- Software Architect
- Hardware Designer
- Software Developer
- Black Box Tester
- Integration Tester
- Deployment Engineer
- Quality Assurance Personnel
- Sustaining Engineer

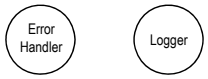
Types of Architecture Documents

- Diagrams
- Documents
- Prototypes
- Test Cases
- Other

Basic Model of the Wind Farm

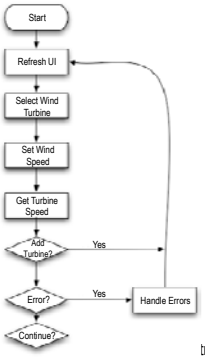


This is very simplistic with no diagram syntax rules. Can simply be the picture from the white board

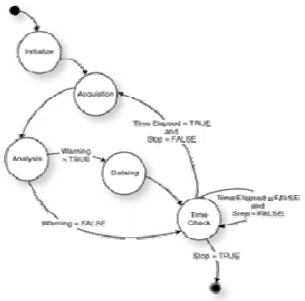


Potential Flow Chart for Parts of the Wind Farm

- Different models convey different information about a system
- No model is ideal for every use case

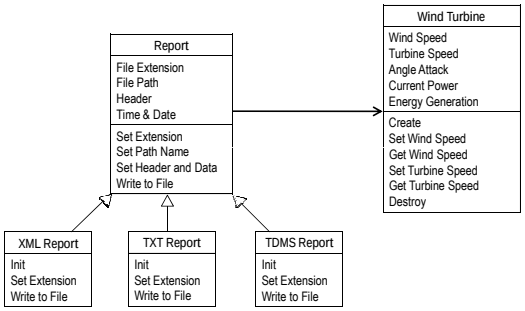


State Machine Diagram



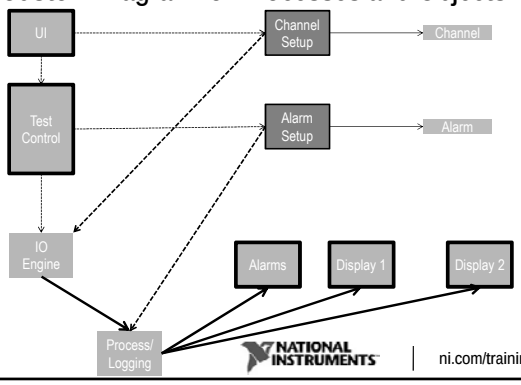
NATIONAL INSTRUMENTS | ni.com/training

UML Class Diagram for Wind Farm Components



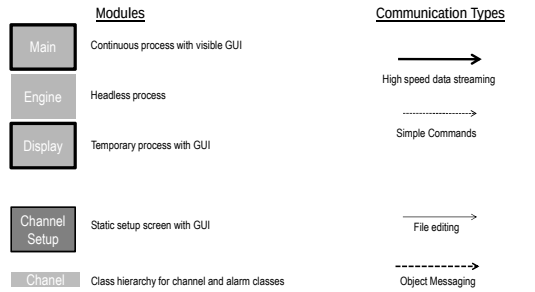
NATIONAL INSTRUMENTS | ni.com/training

Custom Diagram for Processes and Objects



NATIONAL INSTRUMENTS | ni.com/training

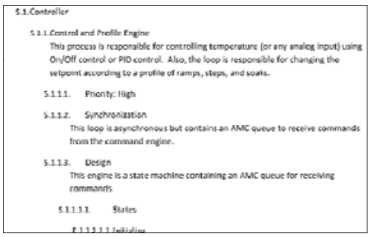
Custom Diagram – Legend



NATIONAL INSTRUMENTS | ni.com/training

Additional Clarifying Documentation

Diagrams convey much of the high-level information. Additional documents may further clarify necessary implementation details.


5.1.Controller

5.1.1.Control and Profile Engine

This process is responsible for controlling temperature (or any analog input) using On/Off control or PID control. Also, the loop is responsible for changing the setpoint according to a profile of ramps, steps, and soaks.

5.1.1.1. Priority: High

5.1.1.2. Synchronization

This loop is asynchronous but contains an AMC queue to receive commands from the command engine.

5.1.1.3. Design

This engine is a state machine containing an AMC queue for receiving commands.

5.1.1.3.1. States

NATIONAL INSTRUMENTS | ni.com/training

Prototype

Prototype–A early sample or model built to test a concept.

Advantages of Prototyping:
- Risk mitigation
- Evaluation of requirements
- Rapid development (minimal software engineering process)
- Communication to stakeholders
- Evaluation of design alternatives

Creating a prototype is not the same thing as developing.

NATIONAL INSTRUMENTS | ni.com/training

Test Cases

Test Case–For some components in your architecture, consider writing the test code for the component before developing the component itself.

- The component is loosely-coupled and testable
- All features for the component are considered in the test case
- Test case facilitates regression testing
- Very beneficial for Classes, X Controls, and FGVs

NATIONAL INSTRUMENTS | ni.com/training

Example Architecture Checklist Items

- I/O processes prototyped for performance?
- Class diagrams complete?
- User interface modularized to accommodate change?
- Have security issues been addressed?
- Has an error handling strategy been documented?
- Is every component technically feasible?
- Has scalability been considered?
- Has the architecture been peer reviewed?

NATIONAL INSTRUMENTS | ni.com/training

Exercise 1-1: Architecting an Application

- Practice designing an architecture
  - Define data structures
  - Define modules
  - Define communication mechanisms
- Compare your initial design with other Wind Farm solutions and other students' designs and ideas
- Identify gaps in existing programming techniques toolset
- Evaluate several potential solutions that demonstrate advanced design patterns
- If applicable, practice for the Certified LabVIEW Architect Exam

Hint: You could begin with nouns and verbs.

GOAL

Exercise 1-1: Architecting an Application

- Were any requirements unclear?
- What are your initial thoughts on your design?
- What was easy to determine?
- What are the challenges?
- At what point did you choose between two design options?
- Which programming techniques interest you most?

DISCUSSION

Implementing the Design

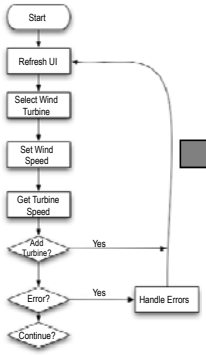
State Diagram

State machine

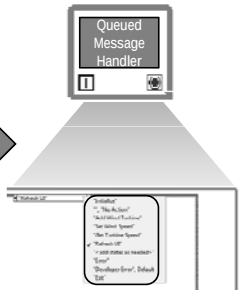
NATIONAL INSTRUMENTS | ni.com/training

Implementing the Design

Flowchart



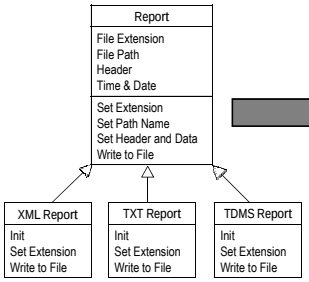
Process Loop



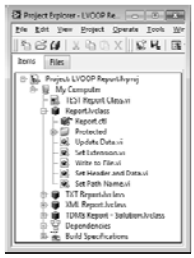
NATIONAL INSTRUMENTS | ni.com/training

Implementing the Design

Class Diagram



LabVIEW Classes



Implementing the Design – Code Generation Tools

- GOOP Development Suite by Symbio–Generate code such as classes, VIs, and icons directly from UML class diagrams
- G# StarUML Plug-in by AddQConsulting–Generate code from a StarUML class diagram
- Create your own code generation tools with VI Scripting

C. Design Patterns and Scalability

Design Patterns–"You know you don't want to reinvent the wheel (or worse, a flat tire), so you look to Design Patterns – the lessons learned by those who've faced the same problems."

– *Head First Design Patterns* by Freeman, Robson, Sierra & Bates

- Template or starting point
- Confidence in the design pattern
- Community of users define these as needed
- Common with OO

Design Patterns in LabVIEW

- Basic design patterns that ship with LabVIEW (instructional)
- Advanced design patterns and templates in this course
- Object-oriented design patterns
- New design patterns that emerge over time

So, why do we need advanced design patterns?

Exercise 1-2: Analyzing a Design Pattern (OPTIONAL)

To analyze the Producer/Consumer Design Pattern (Events).

GOAL

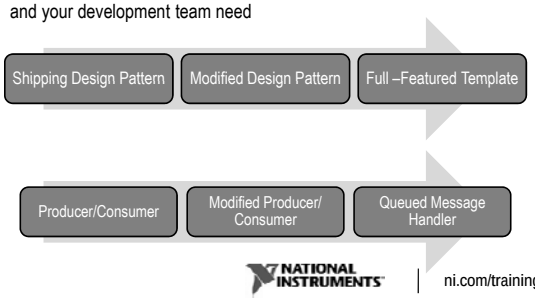
Exercise 1-2: Analyzing a Design Pattern (OPTIONAL)

- What use cases does this solve?
- What is missing?
- What features are necessary in a template that an architect hands off to a developer?

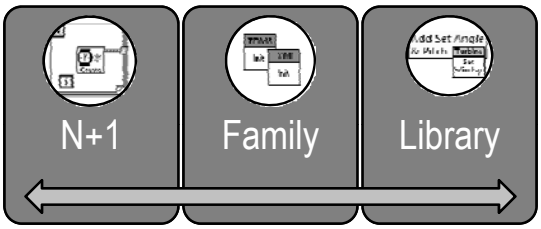
DISCUSSION

Modifying Basic Design Patterns

The design patterns that ship with LabVIEW form the foundation for you to build templates that include all common functionality that you and your development team need



Hallmarks of Scalability



Scalability of Design Patterns

- Each design pattern that you learn is scalable
- Each design pattern has limits to scalability (in particular, FGVs)
- Each design pattern requires a different skill level to modify
  - Easy to create new plug-ins
  - More difficult for the novice to add methods to a class

Exercise 1-3: Identifying Design Challenges

List three design challenges in an application that you are currently working on or that you have worked on in the past.

GOAL

Exercise 1-3: Identifying Design Challenges

- What are the challenges that you face in your application?
- Which have been mentioned previously?
- Which were not addressed?
- What should you mention for the benefit of the rest of the class?

DISCUSSION

Design Well!

- Take the time to thoughtfully design a scalable, readable, maintainable software architecture
  - Prevents last-minute changes
  - Mitigates the cost of any necessary changes in the future
- Last-minute changes are expensive

Software Engineering Process Phase	Cost Ratio For Making a Change
Requirements	1
Design	3-6x
Development	10x
Development – testing	15-40x
Validation	30-70x
Post-release	40-1000x

Lesson 2
Designing an API

TOPICS

A. APIs
B. API Design Techniques
C. Polymorphic VIs for an API
D. Project Libraries
E. Passing Data in an API

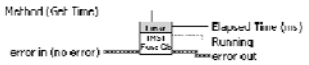
A. APIs (Application Programming Interfaces)

- Create a clean and consistent interface
- Internal implementation can change

APIs

- Interact with code through interfaces
  - A user interface is a tool you use to interact with an application
  - An API is a tool that you use to programmatically call code you or someone else wrote
- When you create an interface, you must carefully consider the design
  - Interfaces must be easy for the target audience to understand
  - Interfaces must allow the target audience to perform tasks efficiently
  - Interfaces must be scalable

APIs



- APIs in text-based languages
  - Function prototypes are the API for functions
  - Properties, methods, and/or events are the API for controls or objects
- APIs in LabVIEW
  - Connector panes are the API for subVIs
  - Palettes and libraries are the top-level API for a library
  - Properties, methods, and the data type are the API for XControls
  - Configuration dialogs and connector panes are the API for Express VIs

Designing an API

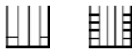
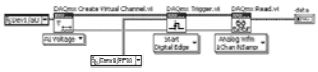
"Make simple things simple"

- 80% of users use only a small part of an API
  - Make that 80% as easy as possible
  - API still needs to serve the other 20%
- Balance ease of use and advanced capability

B. API Design Techniques

Consistency of SubVIs	• Connector Panes • Front Panels • Data types • Planning for Change
Organization of Libraries	• Layered API Design • Palette Menu Structure
Usability of APIs	• Documentation • Naming • Icons • Place VI Contents

Consistent Connector Panes

Consider 4×2×2×4 (or 5x3x3x5)	
Good Layout	
Needs Improvement	

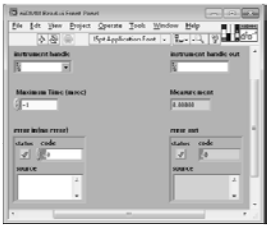
Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Consistent Front Panels

Organize front panels consistently

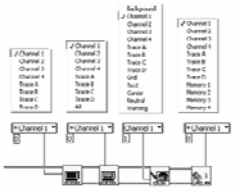
- References on top corners
- Error I/O on bottom corners
- Maintain consistent connector pane layout

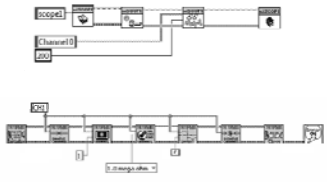


Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Consistent Data Types



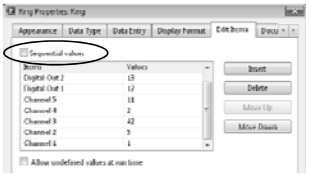


Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Data Types – Non-sequential Ring Controls

- Ring controls do not need to start with 0 and increment by 1
- Great data type for API design



Consistency | Organization | Usability

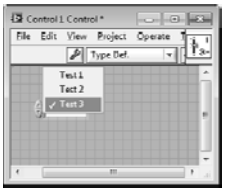
Data Types – Combo Boxes

- Have a string value, but the string value does not have to match what the user sees
- Isolate the block diagram from changes to the user interface
- Useful for localization

Consistency | Organization | Usability

Data Types – Enum & Cluster Type Definitions

- Enum type definitions
  - Useful for providing a selectable, fixed list of commands or options for an API
- Cluster type definitions
  - Useful for organizing related items into a single input or output terminal
- Automatic updates



Consistency | Organization | Usability

Changes to the Connector Pane

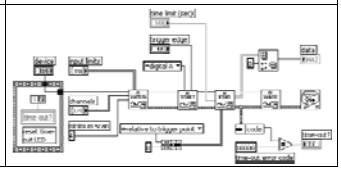
Your API may change as you maintain your code

- LabVIEW handles some changes automatically
  - Changes to numeric representation
  - New parameters added to empty terminals on connector pane
  - Updates to typedefs
- Other changes break code that calls the API
  - Changes to data type, such as a number to string
  - Changes to the connector pane pattern

Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

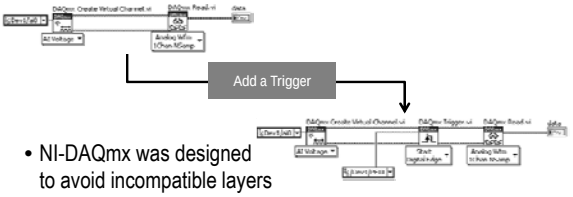
Designing Layered APIs – Avoid Incompatible Layers – Traditional DAQ API Case Study

Easy VIs	Intermediate VIs
	

Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Avoiding Incompatible Layers in APIs

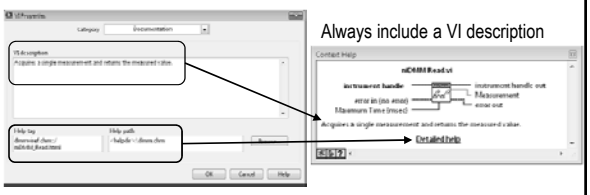


- NI-DAQmx was designed to avoid incompatible layers
- Well-defined default behavior if API functions are not used

Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Documentation



Always include a VI description

Link your API VIs to reference help files

Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Naming Conventions

Good	Better
Mode	Handshaking mode
Ch	Channel
Level	Trigger level, Trigger voltage level
Time	Time (sec)
Action	Action (Get Data)

Don't forget to include the units and default value.

Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

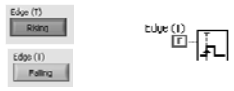
Increasing Usability with Appropriate Naming

- Keep it simple
- Avoid ambiguity
  - Unclear: Polarity (T)
  - Clear: Positive Polarity (T)
- Avoid reverse logic
  - Reverse logic: Don't Convert EOL (F)
  - Clear: "Convert EOL (T)"

Consistency | Organization | Usability

NATIONAL INSTRUMENTS | ni.com/training

Self Documenting Inputs

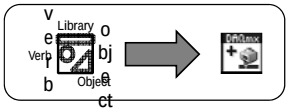
Using Boolean text on a button does not provide enough self-documentation on the functionality when used programmatically	
Ring and enum controls provide self-documentation for both interactive and programmatic use	

Consistency Organization Usability

NATIONAL INSTRUMENTS | ni.com/training

Icons

- Unifying Theme
  - Use a common element or theme to group your API.



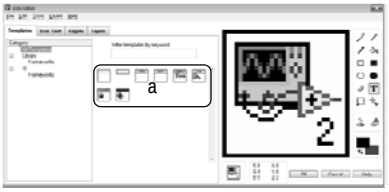
- Text
  - Avoid using culturally-specific images or words.
  - Use fonts consistently.

Consistency Organization Usability

NATIONAL INSTRUMENTS | ni.com/training

Icon Templates

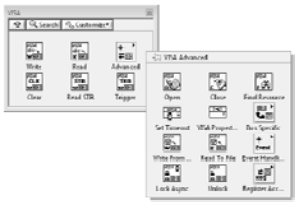
- Use the Icon Editor to design custom icon templates for your team
- Distribute icon templates to the `LabVIEW Data\Icon Templates` directory



Consistency Organization Usability

Palette Menu Structure

Place the most common icons at the top of the palette

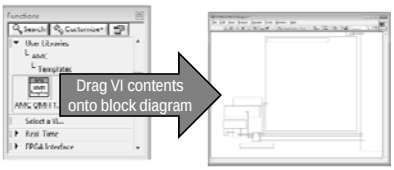


Consistency → Organization → Usability

Place VI Contents

- You can place the contents of the VI block diagram instead of a subVI from the Functions palette
- Consider placing VI contents for code templates in an API

Edit Palette Set dialog box

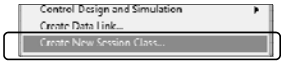


Consistency → Organization → Usability

Maintenance of the API

- The API must be a functioning and debugged black box for the user
- Designate an API owner who performs updates as needed
- The developer chooses to deploy code snippets or code generation tools as needed

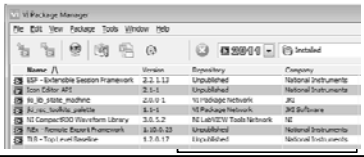
LabVIEW Tools Menu



Custom code generation tool

API Deployment Option – VI Package Manager

- Architects must handle dependencies when using multiple APIs and multiple versions
- VI Package Manager:
  - Combines all the code and resources for a specific add-on, code library, or API in a VI Package file
  - Facilitates sharing reusable code libraries among a team
  - Ensures a developer has the right toolkits and libraries installed



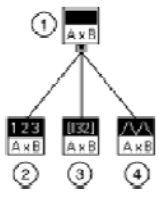
C. Polymorphic VIs for an API

Use polymorphic VIs to:

- Create an API that can accept different data types or has multiple functions
- Accept different data types for a single input or output terminal
- Collect VIs with the same connector pane
- Good for converting data types

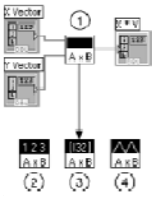
Polymorphic VIs – Example

Perform the same mathematical operation on a double-precision numeric, an array of numeric values, or a waveform



1	Compute
2	Compute Number
3	Compute Array
4	Compute Waveform

Polymorphic VIs – Example (continued)



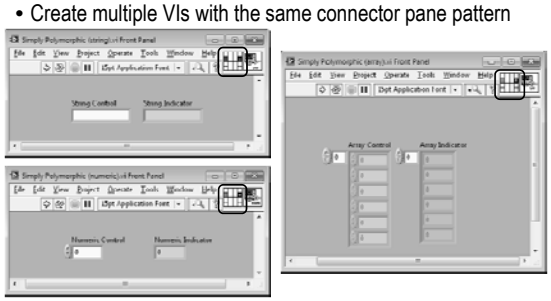
1	Compute
2	Compute Number
3	Compute Array
4	Compute Waveform

Polymorphic VIs – Considerations

- Each instance VI must have the same connector pane pattern
- The set of instance VIs must have compatible connector panes
  - Each input terminal on one connector pane must be an input or an empty terminal on all other connector panes
  - Each output terminal on the connector pane must be an output or an empty terminal on all other connector panes
- Polymorphic VIs cannot be used in other polymorphic VIs

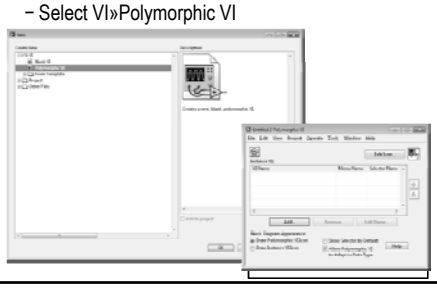
Building Polymorphic VIs – Step 1

- Create multiple VIs with the same connector pane pattern



Building Polymorphic VIs – Step 2

- Create a new polymorphic VI
  - Select File»New
  - Select VI»Polymorphic VI

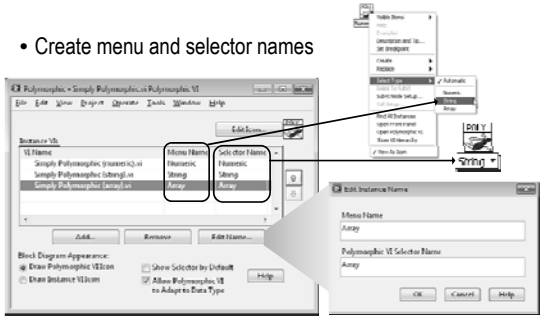


Building Polymorphic VIs – Step 3

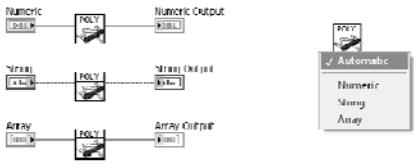
- Add instance VIs to the polymorphic VI

Building Polymorphic VIs – Step 4

- Create menu and selector names



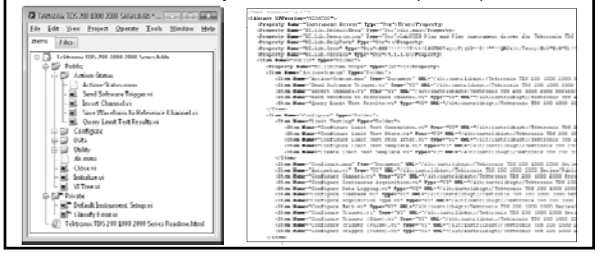
Using Polymorphic VIs



NATIONAL INSTRUMENTS | ni.com/training

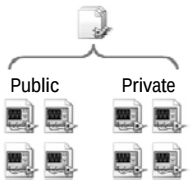
D. Project Libraries

- Tool to facilitate the creation of an API
- Set of related VIs and other LabVIEW files
- Information on library is stored in a text file (`.lvlib`)



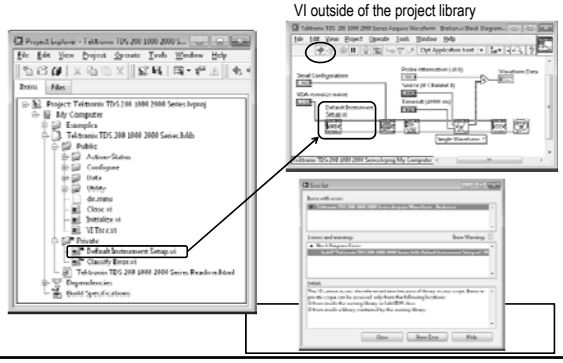
Project Libraries – Advantages

- Restrict access to specific VIs
  - Public VIs
  - Private VIs
- Reduce naming conflicts
  - Namespace applied to VI names



NATIONAL INSTRUMENTS | ni.com/training

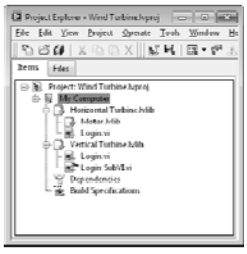
Project Libraries – Using an API from a Project Library – Private VI Example



Design Considerations For Project Libraries

- Facilitate the duplication of a group of VIs
- Consider locking a project library so that private items are hidden in the Project Explorer
- Remember to make naming and location changes from the project library in LabVIEW, not from Windows Explorer
- Calling VIs must be in memory when you make naming and location changes to a project library
- For large distributed projects, the architect should plan the processes for handling changes to a project library

Project Libraries

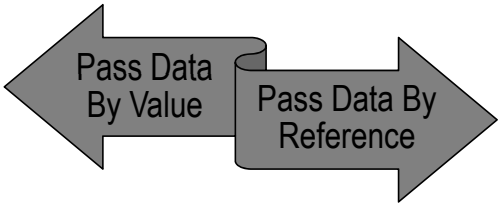


`<Exercises>\…\Demonstrations\Project Library\Wind Turbine.lvproj`

DEMONSTRATION

E. Passing Data in an API

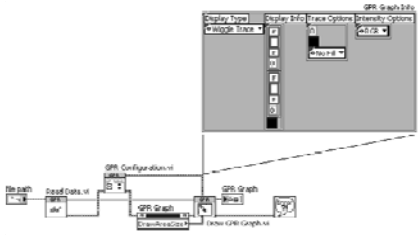
How do you pass state information among VIs in an API?



Pass Data By Value

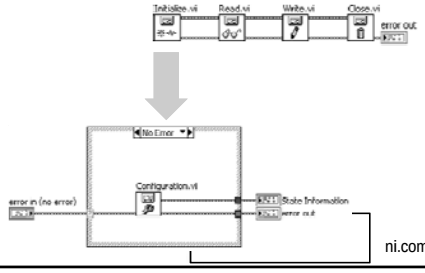
Consider using scalable data types to pass data

- Clusters
- Strings
- Variants
- Objects



Pass Data By Value (continued)

- Easiest method for passing state information in an API
- LabVIEW classes (objects) greatly improve data protection



Pass Data By Reference

- Data passed by reference such as a pointer or handle
- C/C++ uses a pointer or handle to pass data by reference
- Reference data is stored in memory accessible by multiple resources
- Mechanism that aids the development of APIs

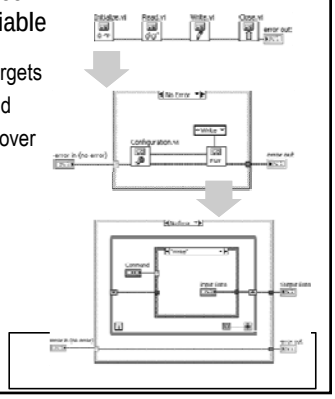
Pass Data By Reference (continued)

Examples of passing data by reference in a LabVIEW API

- Functional global variable (actually neither pass by value or pass by reference)
- Queue
- Data Value Reference

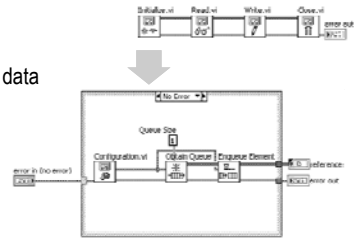
Pass Data By Reference – Functional Global Variable

- Available on LabVIEW targets
- Data can be encapsulated
- Programmer has control over data access
- Can create data copies
- Where is the reference?

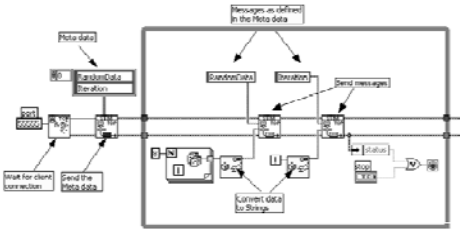


Pass Data By Reference – Queues

- Create a single element queue to pass data by reference in an API
- Fast method
- Does not copy data



Example API – Simple Messaging Reference Library (STM)



Exercise 2-1: Evaluating an API Design

To review design decisions behind the Simple Messaging Reference Library (STM) API.

GOAL

Exercise 2-1: Evaluating an API Design

DISCUSSION

Lesson 3
Multiple Processes and Inter-Process Communication

TOPICS

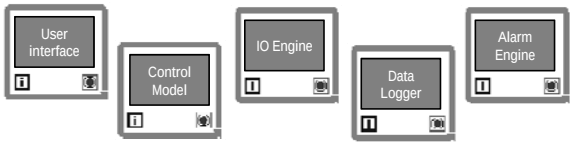
A. Introduction – Processes Foundational APIs
B. Foundational APIs
C. Foundational Design Patterns
D. Scaling Foundational Design Patterns
E. What is an Asynchronous Dynamic Process?
F. By Reference Inter-Process Data Storage
G. Inter-Target Communication
H. Analysis of LabVIEW Communication APIs
I. Communication Reference Architectures

NATIONAL INSTRUMENTS | ni.com/training

A. Introduction – Processes

Process – (in this discussion) an independent, asynchronously executing segment of code

A large application usually has many processes executing concurrently

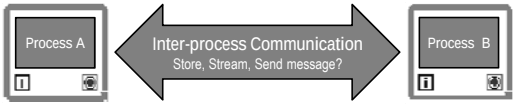


NATIONAL INSTRUMENTS | ni.com/training

Introduction – Inter-Process Communication

Communication – the act of transferring data from one process to another process

Includes moving data within applications, between applications, and between targets on a network



NATIONAL INSTRUMENTS | ni.com/training

Introduction – Inter-Process Communication

- Store Data
- Stream Data

Typically straightforward use cases with limited implementation options

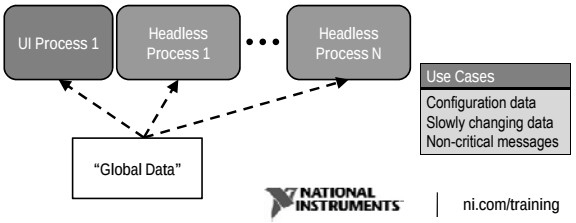
- Send Message

Many more variations, permutations, and design considerations

NATIONAL INSTRUMENTS | ni.com/training

Store Data

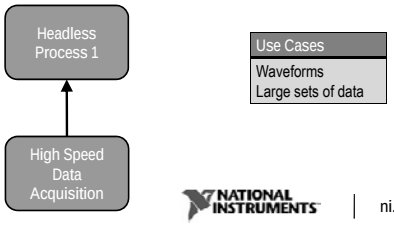
- Data is stored and made "globally" accessible
- Storage mechanism holds only the current value
- Other code modules access the data as needed



NATIONAL INSTRUMENTS | ni.com/training

Stream Data

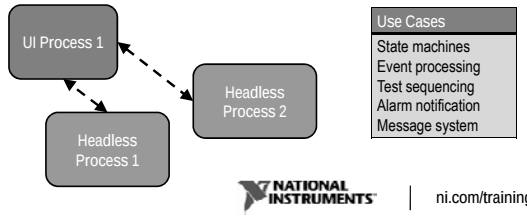
- Continuous stream of data
- Often a lossless buffered transmission
- Often a point-to-point transmission



NATIONAL INSTRUMENTS | ni.com/training

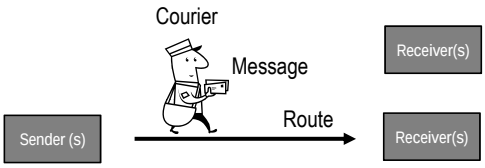
Send Message

- Define format of the message
- May or may not require a response or acknowledgement
- Could have one or more start and end points



NATIONAL INSTRUMENTS | ni.com/training

Special Considerations for Messages



NATIONAL INSTRUMENTS | ni.com/training

Design Considerations for the Message

- Can be simple state information
- Could include data that the state will process
- Can have unlimited messaging formats
- Could include routing information
- Could include the return address
- Confirmation of send, receive, etc.

NATIONAL INSTRUMENTS | ni.com/training

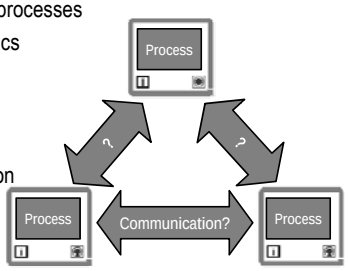
What Inter-Process Communication Considerations Might Apply to the Wind Farm Project?

DISCUSSION

Additional Considerations for Implementation

Important considerations when choosing a method to communicate between processes

- Message characteristics
- Scope
- Performance
- Scalability
- Ease of implementation



NATIONAL INSTRUMENTS | ni.com/training

Inter-Process Communication Consideration – Message Characteristics

- Message: List all data that must be transferred
- Route: List the sender(s) and receiver(s) of each piece of data
  - One-to-one, one-to-many, or many-to-one
- Courier: List how each piece of data should theoretically be transferred
  - Only need latest value
  - Need every value

ni.com/training

Inter-Process Communication Consideration – Scope

What is the scope of the communication?

- Different loops/VIs in the same application instance
- Different application instances on the same target
- Different targets on the network

Application1.exe
Application2.exe

VI in Project1.lvproj
VI in Project2.lvproj

Inter-Process Communication Consideration – Performance

- Throughput—how much data needs to be transferred within a given amount of time
- Latency—how much time between sending data and receiving data
- Processor or network bandwidth
- Memory

Inter-Process Communication Consideration – Scalability

Will you need to scale the communication?

- Communicate to or from *n* loops/applications/targets
- Pass *n* temperature measurements
- Add another data type
- Writers and readers
  - 1:1, 1:N, N:1, or N:N

Inter-Process Communication Consideration – Ease of Implementation and Use

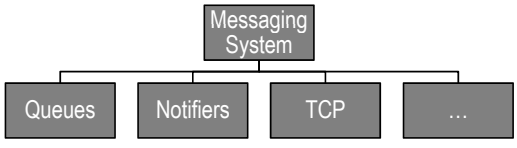
- Does the functionality of the communication method warrant the time and effort needed to implement the communication method?
- Will your team of developers be able to understand and use the communication method?

Other Design Considerations

- Will different processes periodically connect and disconnect?
- What kind of back-up process is required for an unexpected interruption of the communication?
- How are incorrectly formatted messages be handled?
- Is a message history or filter required?
- Does the message require acknowledgement or response?

B. Foundational APIs

Design patterns and scalable reference architectures are dependent on APIs that are native to LabVIEW



Various Inter-process Communication Methods

	Same target Same application instance	Same target, different application instances OR Different targets on network
Storing - Current Value	• Single-process shared variables • Local and global variables • FGV, SEQ, DVR • CVT • Notifiers (Get Notifier)	• Network-published shared variables (single-element) • CCC
Sending Message	• Queues (N:1) • User events (N:N) • Notifiers (1:N) • User Events	• TCP, UDP • Network Streams (1:1) • AMC (N:1) • STM (1:1)
Streaming	• Queues	• Network Streams • TCP

Example Native LabVIEW APIs

and more (RT, FPGA, etc)...

Inter-Process Communication APIs

- Native to LabVIEW
  - Ship with LabVIEW
- Common Design Patterns
  - Emerged over time in the LabVIEW community as excellent foundations to larger systems
- Reference Architectures (NI or the Community)
  - Tend to be larger frameworks that extend beyond the common design patterns
  - Do not follow a strict release process but have been generally accepted in the LabVIEW community.

APIs for Storing Data

- Native LabVIEW
  - Any File I/O
  - Global and local variables
  - Shared Variables
  - Data Value References (DVR) ←
- Common Design Patterns
  - FGVs (Foundational) ←
- Reference Architectures
  - CVT ←

We will cover these in detail

APIs for Streaming Data

- Native LabVIEW
  - Queues ←
  - Network Streams ←
  - TCP

We will cover these in detail

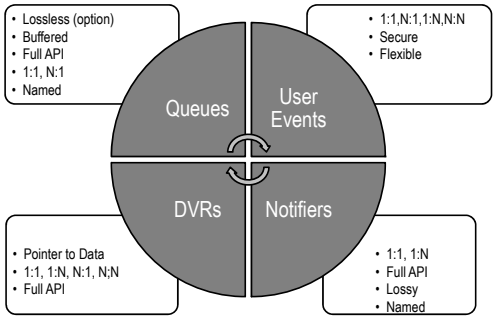
APIs for Sending Messages

- Native LabVIEW
  - Queues
  - User Events ←
  - Notifiers
  - Network Streams
  - TCP
  - UDP
- Reference Architectures
  - AMC ←
  - STM

We will cover these in detail

Note: AMC combines messaging with a foundational Queued Message Handler

Foundational Native LabVIEW APIs for Messaging



Foundational APIs for Messaging – Queues (Review)

- Lossless transmission of data
- Many-to-one relationship (1:1 or N:1)
- Good for data streaming and messaging
- Rich API for manipulating the queue

Queues – Benefits

- Secure
- Globally accessible by name within an application instance
- Flexible, element can be any data type
- Fast!
- Optimizes memory with limited data copies
- Full API for various queue operations
- Foundational API that must be mastered

Secure Messaging

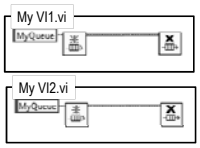
- Maximum queue size is set to 50.
- What happens if the producer tries to enqueue faster than the consumer dequeues?

Launch the `Queue Basics.vi` in the NI Example Finder.

DEMONSTRATION

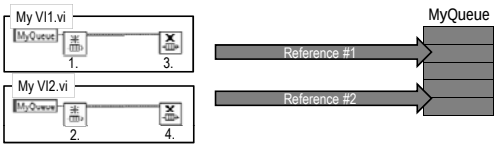
Named Queues – Global to an Application Instance

- If two separate VIs call the Obtain and specify the same name, then the second Obtain retrieves a second reference to the original queue
- This allows the two separate VIs to send and receive messages without the use of a global variable or an FGV
- This can be a powerful construct for scalable applications



Issues with named queues are covered later in this lesson.

Queues – Named Queues & References



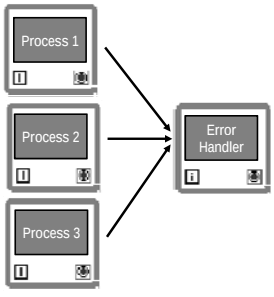
1. The 1st Obtain creates a reference to the queue.
2. 2nd reference to the same queue is created by the 2nd Obtain.
3. Original reference is released on the 1st Release. The queue is still in memory.
4. 2nd reference is released on the 2nd Release. The queue is destroyed and is no longer in memory.

Note Setting force destroy to TRUE releases all references and removes the queue from memory.

Named Queues & References – Example

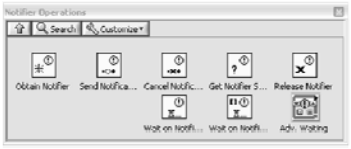
Assume that 3 processes obtain a reference to the same named queue in the Error Handler process

- When Process 1 completes, it should release its reference to the Error Handler queue. Other processes can continue to communicate with the Error Handler queue.
- If the Error Handler process shuts down, then it should destroy the Error Handler queue so that remaining processes no longer attempt to communicate.



Foundational APIs for Messaging – Notifiers (Review)

- Lossy
- Contain only current value
- Broadcast (or one-to-many, 1:N)
- Due to lossy nature, not used as frequently as queues



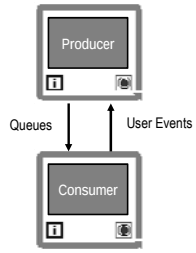
Foundational APIs for Messaging – DVRs

- Available in LabVIEW 2009 and later
- Allow the developer to wrap data in a LabVIEW reference which acts like a pointer
- Facilitate the creation of multiple instances of component
- Are very scalable and should be considered as a replacement for FGVs

DVRs are covered in detail later in this lesson.

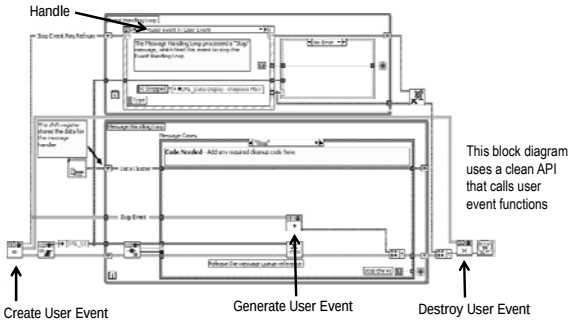
Foundational APIs for Messaging – User Events

Review



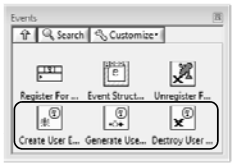
A common use case for user events is sending messages from the consumer loop back to the event structure in the producer loop

Using User Events to Stop a Process



NATIONAL INSTRUMENTS | ni.com/training

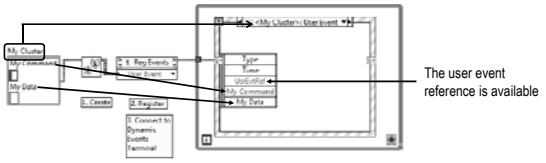
User Events – Generate, Unregister, and Destroy (Review)



- Limited API
- No way to preview events
- No way to flush buffered events that have not been handled
- Cannot be named like queues and notifiers

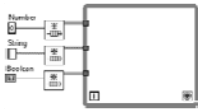
NATIONAL INSTRUMENTS | ni.com/training

User Events – Creating and Registering User Events (Review)

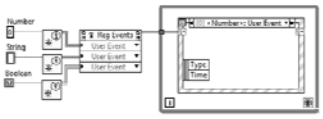


NATIONAL INSTRUMENTS | ni.com/training

Benefits of User Events – Scalable for Multiple Data Types



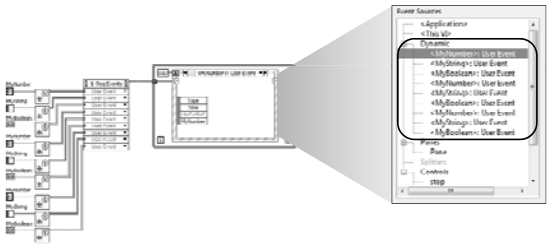
When using queues to pass different data types, each data type requires a unique queue and the associated reference.
The developer must code a system to dequeue from each unique queue. This does not scale well.



When using user events, all information is contained on one event registration refnum.
One Event structure handles all types of user events making this very scalable.

User Events Design Issue – Organizing User Events

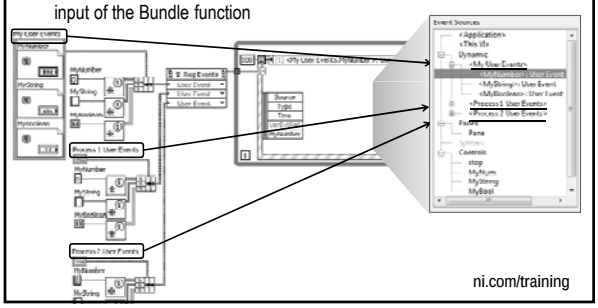
When using many different user events, the Edit Events dialog can become cluttered and difficult to manage



User Events Design Issue – Organizing User Events

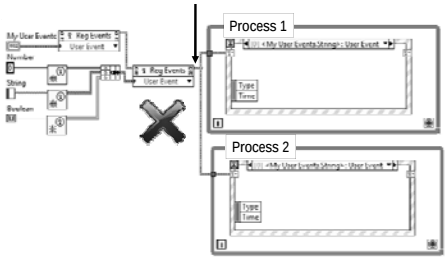
Organize the user event references in the Edit Events dialog box

- Group user event references by using the Bundle function
- Name the groups by creating and labeling a constant from the cluster input of the Bundle function



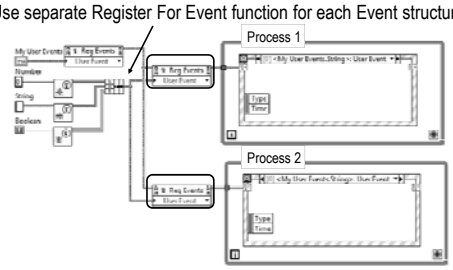
Handling Events in Multiple Loops – Wrong Method

When registering multiple Event structures to handle each event, do not fork the event registration refnum.



Handling Events in Multiple Loops – Correct Method

- Fork the user event refnum
- Use separate Register For Event function for each Event structure

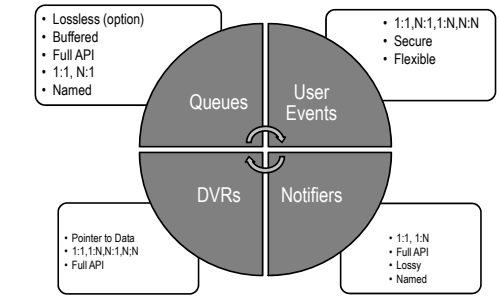


Understanding Event Registration

<Exercises>\...\Demonstrations\Event Registration

DEMONSTRATION

Summary – Foundational Native LabVIEW APIs for Messaging



DISCUSSION

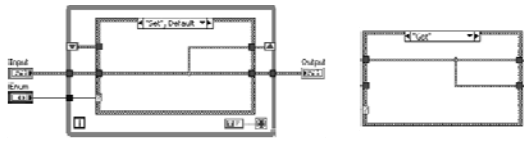
C. Foundational Design Patterns

- Design pattern for global data storage
  - Functional Global Variable (Review)
- Design patterns to implement processes
  - State machine (Review)
  - Producer/Consumer (Review)
  - JKI state machine
  - Queue-driven message handler

NATIONAL INSTRUMENTS | ni.com/training

Functional Global Variables (Review)

- A functional global variable usually has an action input parameter that specifies which task the VI performs
- The VI uses an uninitialized shift register in a While Loop to hold the result of the operation



NATIONAL INSTRUMENTS | ni.com/training

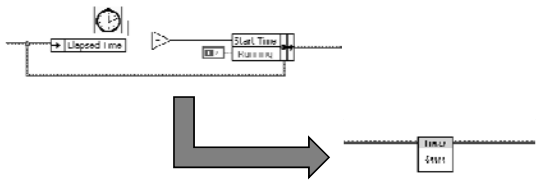
48

Functional Global Variables – History

(LV2 Style Global, Action Engine, VIGlobals, USRs, Components)

- Global data storage mechanism prior to the introduction of the global variable in LabVIEW 3
- Foundational programming technique that has been in extensive use in the LabVIEW community

SubVI Encapsulation



We know how to do this, but the data is still exposed
Additionally, the calling VI must be modified to use this VI

FGVs – Reusable Components

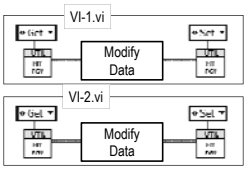
Look for reusable components

Now we have encapsulated data and have a reusable module.

The FGV encapsulates the data and becomes a reusable module.

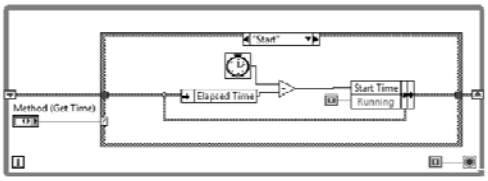
Do FGVs Eliminate Race Conditions?

What if the FGV only included Set and Get methods?



What happens if two VIs both call the Get method, then both modify the data, and then both call the Set method?

FGV – Action Engine Protects Critical Sections of Code

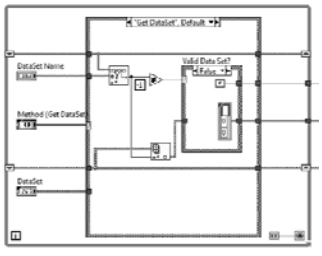


In an action engine, the method encapsulates the "get/modify/set" critical section of code, thus eliminating the race condition.

FGV – Resource Storage

Key-value look up table design pattern

- Array of names has a one-to-one correspondence to the array of data sets
- Can check for valid data
- Does not protect against race conditions



Functional Global Variables – Benefits

- Provide *global access to data* while minimizing potential race conditions introduced by global variables
- *Encapsulate data* so that debugging and maintenance is easier
- Facilitate the creation of *reusable modules* which simplifies writing and maintenance of code
- Can pass any kind of data and can enhance a variety of advanced design patterns (such as user events)

Recall the Discussion of User Events

Pros	Cons
• Can create custom events • Can pass data with the event • Can move data across the block diagram • User events are buffered • User events are ideal for shutdown and other high priority or global actions	• Can be confusing to a novice user • Cannot be named like queues • Must have a mechanism to pass the reference to multiple locations

Being able to pass the reference through a FGV adds flexibility!

FGV For User Events

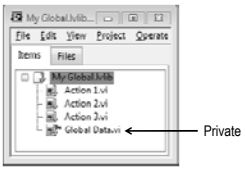
Use a scalable FGV that manages user event references

`<Exercises>\...\Demonstrations\FGV for User Events\User Events FGV.lvproj`

Students can refer to Exercise 3-7 (Optional) for detailed instructions to recreate the files in this demonstration

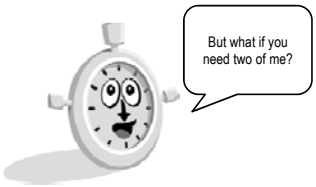
DEMONSTRATION

Another Option – Private Global Variable



- Global Data.vi is a global variable and a private member of a Project Library
- Access to the data is limited to other member VIs
- The action VIs perform get/modify/set in one function, thus eliminating race conditions

Functional Global Variables Limitation



`<Exercises>\...\Demonstrations\FGV vs DVR\Functional Global Variable\FGV Timer.lvproj`

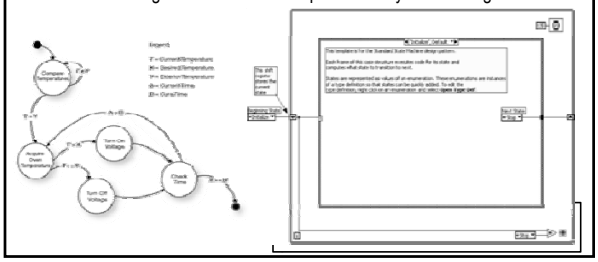
DEMONSTRATION

Design Patterns for a Process

- LabVIEW development teams should standardize on a core design pattern for the basic framework of a process
- Example starting points for developing your custom template:
  - State machine
  - Producer/Consumer
  - JKI state machine
  - Queue-driven message handler

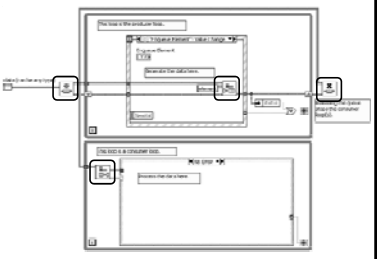
State Machine (Review)

- Usually has a startup and shutdown state, but also contains other states
- Actions may occur in any order and may be interrupted
- Good fit for algorithm that can be represented by a state diagram



Producer/Consumer (Events) – Review

- Efficiently responds to the user interface
- Divide commands into categories
  - Producer handles immediate response
  - Consumer handles extended processing
- Queues provide the communication mechanism

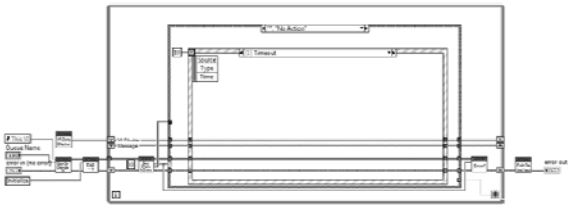


Queue-Driven Message Handler Design Pattern

Based on the Producer/Consumer design pattern

- Can include predefined list of queued messages
- Can dynamically change the flow of events based on results of a state
- Can respond to actions of an operator
- Can respond to interruptions from other sources

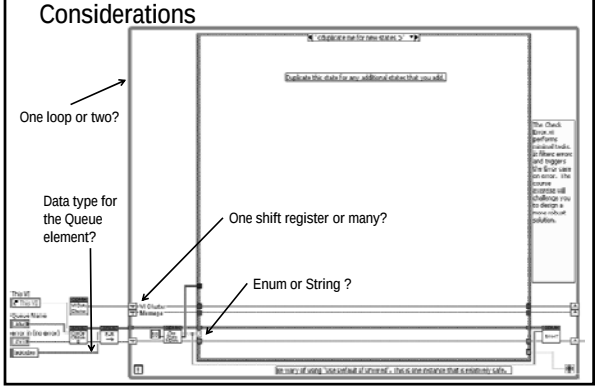
Example QDMH Implementation – QDMH Main.vi



- Queue VIs are embedded in SubVIs to facilitate extensibility and readability
- One loop is used, which simplifies passing data to the event structure and eliminates the need for user events

NATIONAL INSTRUMENTS | ni.com/training

Queue-Driven Message Handler Design Considerations



QDMH Design Consideration – Enum or String?

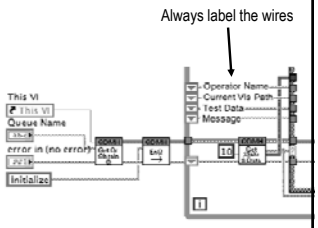
- Enum
  - Seamlessly connects to the case structure such that there are no typos
  - Facilitates adding cases
  - The architect can distribute the type def enum and QDMH to the developer
- String
  - The QDMH template can have a case in the structure that handles potential typos
  - Facilitate integration with other functions that use strings such as Run Time Menu Tags and TCP/IP Vis
  - Universal data type that is recognizable outside the LabVIEW environment
  - Allows the use of subVIs to encapsulate the enqueue and dequeue

NATIONAL INSTRUMENTS | ni.com/training

QDMH Design Consideration – One Cluster Shift Register or Many?

Using several shift registers

- Easy to view all of the data without unbundling or context help
- Group shift registers in one location (top or bottom)
- Can still organize data into clusters
- You can automatically wire the unwired cases



NATIONAL INSTRUMENTS | ni.com/training

QDMH Design Consideration – Message Formats (Data Type For the Queue Element)

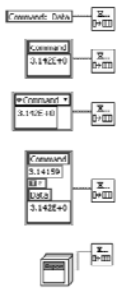
The message format may transfer:

- Command—determines which operation to execute
- Data—operation-specific data

Command formats	Data formats
• Enum • String	• Variant • String • Cluster • Object (LV Class)

NATIONAL INSTRUMENTS | ni.com/training

QDMH Design Consideration – Message Format



- What is the experience level of the developers?
- What kind of data is typically passed?
- Which is highly scalable?
- Have you created the extra tools/snippet VIs for the developer?

NATIONAL INSTRUMENTS | ni.com/training

Queue-Driven Message Handler Template Example

Explore the QDMH Main.vit template in the `<Exercises>\...\Shared` directory

DEMONSTRATION

Exercise 3-1: Queue-Driven Message Handler

To become familiar with a typical queue-driven message handler template.

GOAL

Exercise 3-1: Queue-Driven Message Handler

- Only uses Queues (no user events) which may facilitate learning curve
- Includes a function to exit when executing in run-time
- Uses a polymorphic VI around the enqueue
- Traps errors after the completion of the case structure
- Incorporates the run-time menu in the event structure

DISCUSSION

State Machine – JKI Software

A very well-thought-out alternative.

DEMONSTRATION

Exercise 3-2: JKI State Machine (Optional)

To explore a state machine in which the architect made very different decisions on the design.

GOAL

Exercise 3-2: JKI State Machine (Optional)

- Requires limited use of support VIs
- Defined syntax for the string (command & data) allows for the rapid scripting of a series of commands for prototyping
- Well-documented for ease of editing
- Data communication is not incorporated in this version

DISCUSSION

Another Queued Message Handler Template

- Very clean, readable, and maintainable support APIs
- Easy to modify without full understanding of support VIs
- Integrates user events for the shutdown case
- Facilitates smooth transition for those accustomed to using Producer/Consumer
- Very good starting point for further enhancement

```
<Exercises>\...\Adv Design Patterns and Tools\QMH
Template
```

DEMONSTRATION

Top Level Baseline (TLB) (Optional)

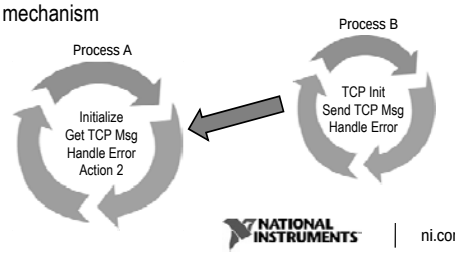
- Getting started tools and full-featured template for creating a top-level application
- Based on the Producer/Consumer (Events) design pattern
- Messaging is included
- Initialization and shutdown code is separated

```
<Exercises>\...\Adv Design Patterns and Tools\TLB
```

DEMONSTRATION

D. Scaling Foundational Design Patterns

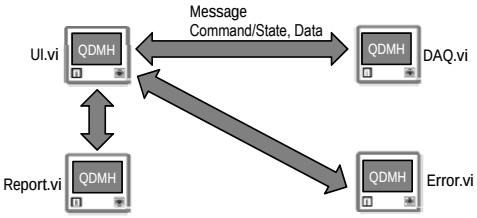
- Scale for multiple processes
- Actions may be invoked by a person, a remote system across TCP/IP, a database, another VI, or some other mechanism



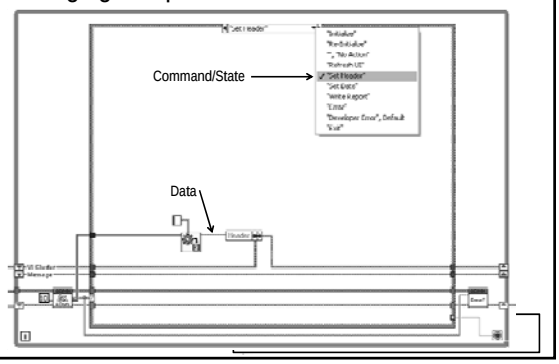
NATIONAL INSTRUMENTS | ni.com/training

Parallel Queue-Driven Message Handlers

- Scale to multiple parallel processes
- Each process may be implemented with its own QDMH
- QDMHs may communicate with each other

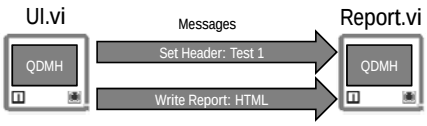


Queue-Driven Message Handler – Messaging Components



Queue-Driven Message Handler – Loose Coupling

- As long as the interface and messaging protocol of the QDMH remains the same, you can change the internal implementation of each QDMH without affecting the larger application



Queue-Driven Message Handler – Strong Cohesion

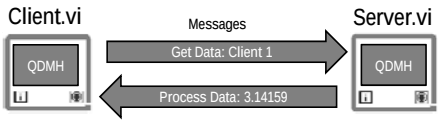
- Clearly-defined and static messaging interface
- Data is encapsulated

Queue Name - Report	
Commands/States	Data
Set Header Set Date Write Report	Test 1 03/06/11 HTML

Report.vi

Queue-Driven Message Handler – Client/Server

- Create Client/Server Architecture
- Multiple clients can connect to the server.
- Server can run independently



Queue-Driven Message Handler – Client/Server

Demonstrate how to communicate between parallel QDMHs

```
<Exercises>\...\Demonstrations\QDMH Client-
Server\Client-Server.lvproj
```

DEMONSTRATION

Issues with Named Queues – Name Collisions

When a process uses a named queue, other processes in the application instance can also access the named queue and interfere with the process.

Consider the following:

A tester has a named QDMH process, which is accessed through the queue name “Test”. Several years later, another engineer adds a named QDMH process. The engineer is unaware of the original QDMH process and also names the new queue “Test”.

What happens?

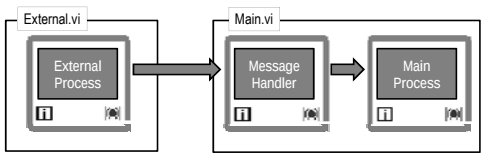
Issues with Named Queues – Disrupting a Process

- Using named queues for inter-process communication is very common.
- However, unexpected results can occur when a main QDMH receives a command on its queue from an external QDMH.

`<Exercises>\...\Demonstrations\Named Queue Issues`

DEMONSTRATION

Issues with Named Queues – Protecting a Process



Instead of allowing an external process to access the Main QDMH queue directly, consider adding code that manages the inbound messages and determines the safe and appropriate time to enqueue the message on the Main QDMH.

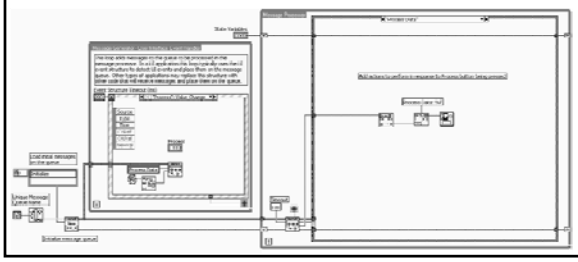
QDMH Implementation – Asynchronous Message Communication (AMC) Reference Library

- Can use the AMC API to implement QDMHs and communicate between QDMHs
  - Single QDMH
  - Multiple QDMHs on the same target
  - Multiple QDMHs on different targets on a network

AMC combines the QDMH with a messaging system

QDMH Implementation – Asynchronous Message Communication (AMC) Reference Library

- Download from `ni.com`
- Includes examples and templates



Asynchronous Message Communication (AMC) Reference Library

- Show how to use the AMC API to implement a single QDMH
- Show how to use the AMC API to communicate between two QDMHs on the same computer
- Show how to use the AMC API to communicate between two QDMHs on different targets

```
<Exercises>\...\Adv Design Patterns and Tools\
Installer for AMC
```

DEMONSTRATION

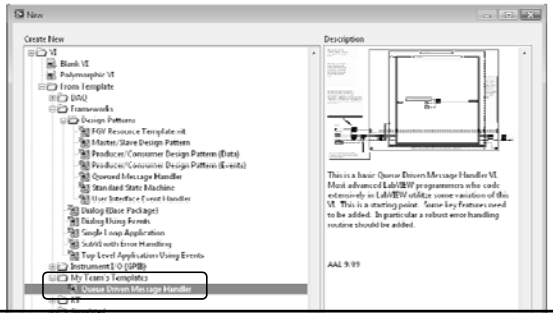
Scalable Components with User Events

<Exercises>\...\Demonstrations\Scaling Components with User Events

DEMONSTRATION

Develop Templates for Your Team

<labview>\templates\My Template.vit



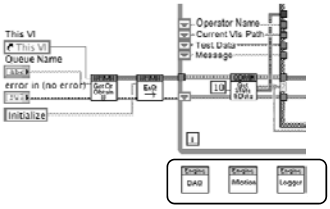
What do you plan to implement?

DISCUSSION

Further Considerations

- What about the scenario in which you don't want to load all of the code at run time?
- What about running a background process that handles archiving of data and runs independently of the main VI?
- What about dynamically determining at run time how many UUT user interfaces to launch?

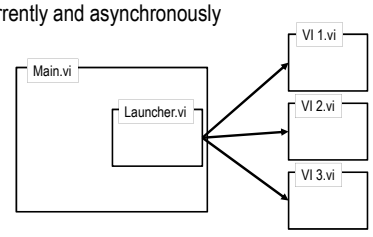
Further Considerations – SubVIs or Asynchronous Dynamic VIs?



VIs run concurrently with the Main.vi and communicate through named queues.

E. What is an Asynchronous Dynamic Process?

At run time, `Main.vi` launches other VIs that run concurrently and asynchronously



Wind Farm #1

Run Wind Farm #1 and observe the asynchronous dynamic wind turbine processes

`<Exercises>\...\Wind Farm Course Project`

DEMONSTRATION

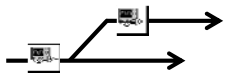
Asynchronous Dynamic Processes

Asynchronously call a dynamic VI representing the process

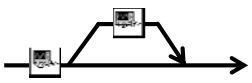
- Benefits
  - Calling VI does not need to wait for the asynchronous VI to finish
  - Launch time-consuming code asynchronously to improve performance
  - Launch processes that run independently
  - Dynamically determine how many instances of asynchronous VI to spawn
- Caveats
  - Harder to control than subVIs
  - Harder to communicate with asynchronous VIs than subVIs

Types of Asynchronous Dynamic VIs

- Asynchronous Independent VIs

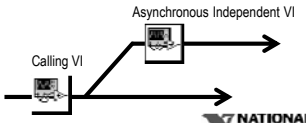


- Asynchronous Subroutines



Asynchronous Independent VIs

- Asynchronously launched from a calling VI
- Executes independent of the calling VI
- Manages its own lifetime
- Also referred to as daemons, spawned VIs, asynchronous call-and-forget VI call, fork-and-forget model



NATIONAL INSTRUMENTS | ni.com/training

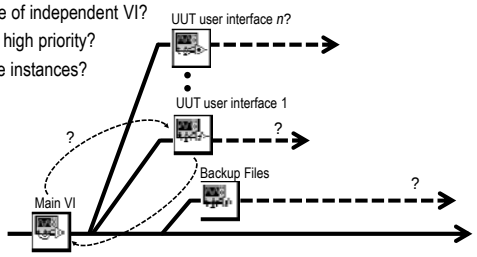
Asynchronous Independent VIs – Examples

- Launch VIs that run invisibly in the background
  - Auto-save utility
  - Periodic back-up service
  - Garbage collection of temporary files
  - Collect data
- Perform low-priority monitoring and/or maintenance services
- Process high-priority processes
- Spawn multiple independent instances of a UI for each Unit Under Test

NATIONAL INSTRUMENTS | ni.com/training

Asynchronous Independent VIs – Design Questions

- Visible UI or headless?
- Autonomous or regular communication with main VI?
- Lifetime of independent VI?
- Low or high priority?
- Multiple instances?



NATIONAL INSTRUMENTS | ni.com/training

Spawning Multiple Instances of an Asynchronous Independent VI

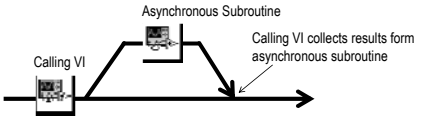
Explore Spawning Multiple Instances.lvproj in the Solutions folder

```
<Solutions>\...\Exercise 3-3\Spawning Multiple
Instances.lvproj
```

DEMONSTRATION

Asynchronous Subroutines

- Asynchronously start the execution of a subVI
- Collect the results of the subVI later
- Limited lifetime
- Also referred to as asynchronous call-and-collect VI call and fork-and-join model

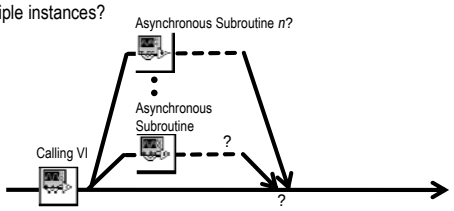


Asynchronous Subroutines – Examples

- Call a subVI asynchronously, continue executing other code, and receive the results of the asynchronous VI when needed later
- Call *n* instances of a subVI asynchronously and collect the results of the subVI in a parallel loop

Asynchronous Subroutines – Design Questions

- Visible UI or headless?
- What is the expected execution time?
- After the asynchronous subroutine finishes, when does caller needs to be notified and/or collect results?
- Multiple instances?

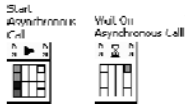


NATIONAL INSTRUMENTS | ni.com/training

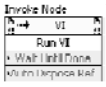
Asynchronous Dynamic VIs – LabVIEW Implementation

Two different LabVIEW implementations

Asynchronous Call nodes



Run VI method



NATIONAL INSTRUMENTS | ni.com/training

Asynchronous Dynamic VIs – LabVIEW Implementation

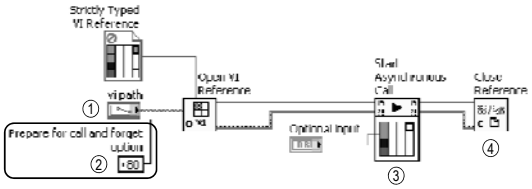
- Do you need to collect the value of the output terminals of the asynchronous VI?
- Do you need to know when the asynchronous VI finishes?
- Choose the model that applies to your use case

Call-and-forget—Start an asynchronous VI call without tracking when or what values the VI returns

Call-and-collect—Start an asynchronous VI call and collect the results later

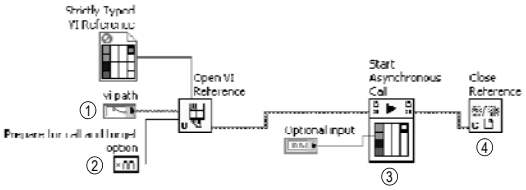
NATIONAL INSTRUMENTS | ni.com/training

Asynchronous Call Nodes – Call-and-Forget



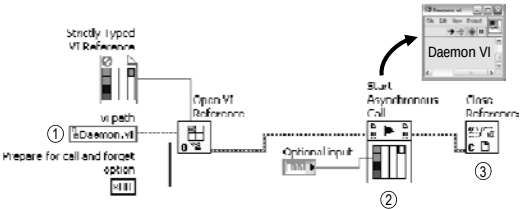
1. VI path of asynchronous VI
2. Include the 0x80 flag as part of the options input to open a call-and-forget VI reference
3. Starts an asynchronous call to the VI indicated by the reference input

Asynchronous Call Nodes – Call-and-Forget



4. Close the VI reference when the calling VI no longer needs the reference
   When you close a call-and-forget VI Reference, LabVIEW does not abort instances of the VI that are already running.
   The asynchronous VI manages its own lifetime.
   The asynchronous VI will leave memory when it has stopped running and has a closed front panel

Call-and-Forget Use Case – Background Daemon

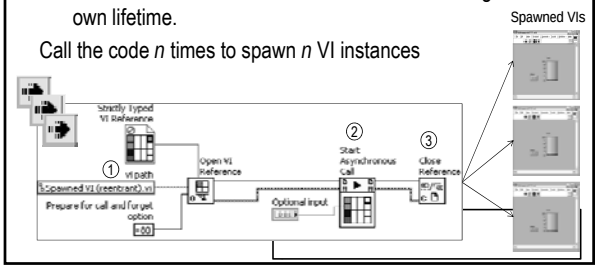


1. VI path of background daemon
2. Starts daemon asynchronously
3. Closes the VI reference. The daemon to manages its own lifetime

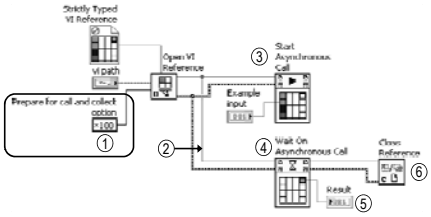
Call-and-Forget Use Case – Spawn Multiple VI Instances

1. VI path of a reentrant VI
2. Starts an asynchronous call to the reentrant VI
3. Closes the VI Reference. Each VI instance manages its own lifetime.

Call the code *n* times to spawn *n* VI instances

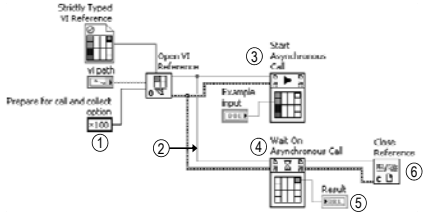


Asynchronous Call Nodes – Call-and-Collect



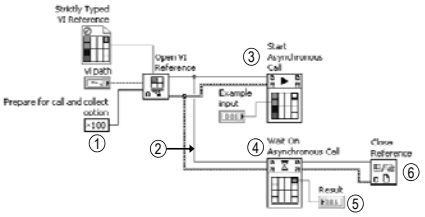
1. Include the 0x100 flag as part of the options input to open a call-and-collect VI reference
2. Wait On Asynchronous Call node is associated with a Start Asynchronous Call node that uses the same VI reference
3. Starts an asynchronous call to the VI indicated by the reference input

Asynchronous Call Nodes – Call-and-Collect



4. Wait On Asynchronous Call node waits until the VI associated with its reference input finishes executing.
5. Wait On Asynchronous Call node returns the outputs of the referenced VI, and dataflow continues along its output wires

Asynchronous Call Nodes – Call-and-Collect

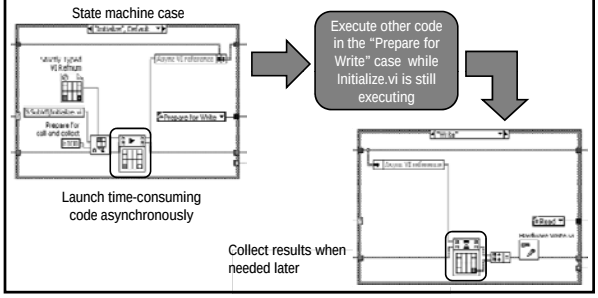


6. When you close a call-and-collect VI reference, LabVIEW aborts all running instances of the VI.

Note: When the VI that opened the call-and-collect VI reference finishes executing, LabVIEW automatically closes the reference and aborts all running instances of the VI.

Call-and-Collect Use Case – Asynchronous Subroutines

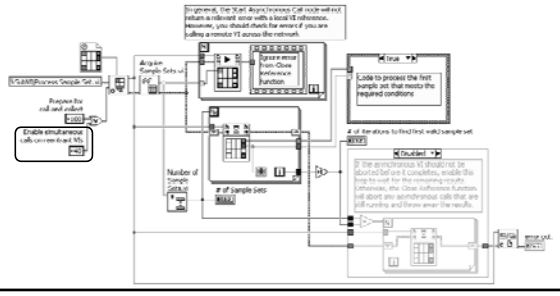
- Launch time-consuming code asynchronously to improve performance



Call-and-Collect Use Case – Multiple Asynchronous Subroutines

Multiple simultaneous asynchronous subroutines example

- Execute code on the first valid set of data



Multiple Asynchronous Subroutines

Explore an example of minimizing execution time by launching multiple asynchronous subroutines and executing code on the first valid result

`<Exercises>\...\Demonstrations\Async Subroutine – Multiple Instances\Async Subroutine – Multiple Instances.lvproj`

DEMONSTRATION

Exercise 3-3: Spawning Multiple Instances of an Asynchronous Independent VI

To create a subVI that you can use to spawn multiple instances of an asynchronous independent VI.

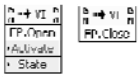
GOAL

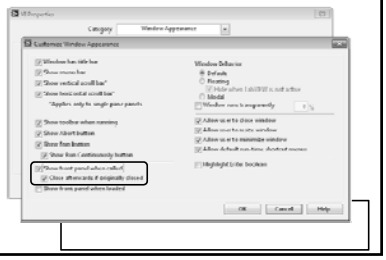
Exercise 3-3: Spawning Multiple Instances of an Asynchronous Independent VI

DISCUSSION

Front Panel of Asynchronous VI

- Front panel of asynchronous VI does not open by default
- Explicitly open and close the front panel of an asynchronous VI if necessary
  - Invoke node
  - VI Properties

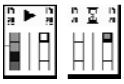




Asynchronous VI Reference Lifetime Management

- Three things can keep a VI in memory
  - Open caller
  - Open VI reference
  - Front panel is open
- Simply running does NOT keep a VI in memory!
- Opening a call-and-collect VI reference keeps ownership of the reference in the calling VI
- Opening a call-and-forget VI reference transfers ownership of the reference from the calling VI to the asynchronous VI

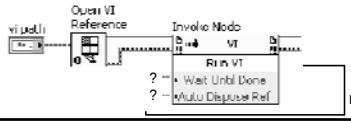
Asynchronous Call Nodes – Caveat



- Caveats
  - Asynchronous Call nodes are only available in LabVIEW 2011 or later
  - Asynchronous Call nodes require a strictly typed VI reference, so you must know the connector pane of the asynchronous VI at edit-time
- If you are using a prior version of LabVIEW or unable to not determine the connector pane of the asynchronous VI at edit-time, use the Run VI method instead

Run VI Method Alternative

- Does not require a strictly typed VI reference
- Wait Until Done input
  - False: Starts execution of VI asynchronously
- Auto Dispose Ref input
  - True: Transfers ownership of the VI reference to the asynchronous VI
  - False: Keeps ownership of the reference in the calling VI

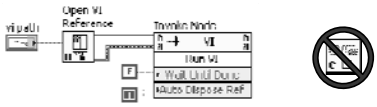


ni.com/training

Run VI Method Alternative – Example

Asynchronous independent VI example

- Starts execution of the VI asynchronously
- Transfers ownership of VI reference to asynchronous VI
- Do not use Close Reference function in the calling VI because asynchronous VI will manage its own lifetime



NATIONAL INSTRUMENTS | ni.com/training

Exercise 3-4: Fixing the Run VI Method

To fix a VI in which VI references are not being handled correctly when using the Run VI method.

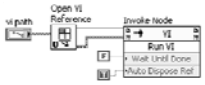
GOAL

Exercise 3-4: Fixing the Run VI Method

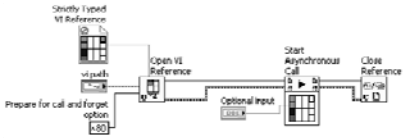
DISCUSSION

Run VI Method and Asynchronous Call Nodes Comparison – Launch Asynchronous Independent VI

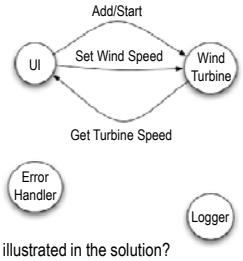
Run VI method



Asynchronous Call nodes



Wind Farm #1



- Which of the previous concepts are illustrated in the solution?
- How is communication handled between the main UI and the wind turbines?
- How is the Wind Farm shut down?
- What is clear and what is unclear in this implementation?
- What would be a more clever solution?

`<Exercises>\...\Wind Farm Course Project`

DISCUSSION

Wind Farm #1

	Wind Farm #1
User Interface	Dynamically launch a new UI for each wind turbine
Data Communication Between UI and Turbines	VI Server (Set/Get Ctrl Value)
Shutdown Turbines	Poll FGV

DISCUSSION

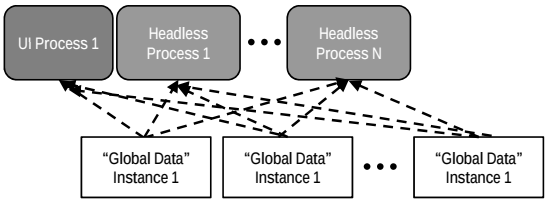
Sidebar: Using Get/Set Ctrl Value VI Server Methods for Data Communication

- Wind Farm #1 uses the Get/Set Ctrl Value VI Server methods for communicating to the dynamic processes.
- Several design issues arise:
  - Can be easy to implement initially
  - Developer can add capabilities to a main process without editing the dynamic process
  - Requires the front panel of the dynamic process to be in memory
  - Overuse of this method quickly increases the likelihood of race conditions
  - Main VI and dynamic process quickly become tightly coupled
- Wind Farm #2 and #3 implement better solutions

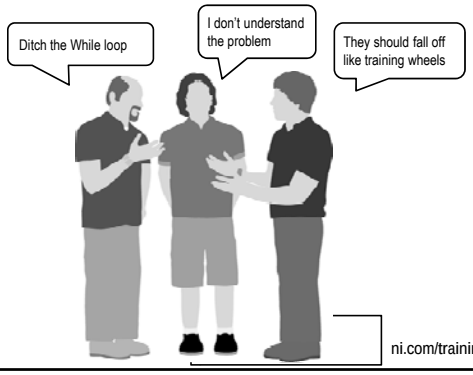
Note: Under the guidance of a skilled architect, some very-well organized development teams build reusable components based on a clearly-defined control labeling process and control references.

F. By Reference Inter-Process Data Storage

- You might need multiple instances of the global data object
- Additionally, the global data must be accessed by multiple processes



Return to Functional Global Variables



ni.com/training

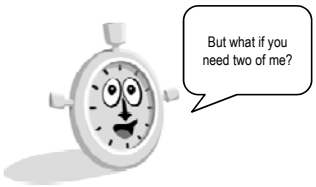
What if You Need Multiple Timers...

- Reentrant functional global?
- Array manipulation of the functional global data?
- Perhaps there is a better way…

?

NATIONAL INSTRUMENTS | ni.com/training

Multiple Timers Using Data Value References

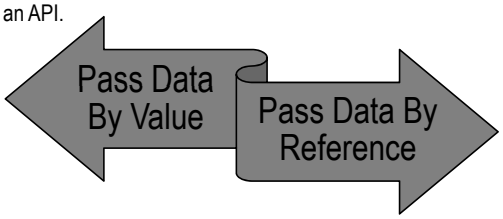


<Exercises>\...\Demonstrations\FGV vs DVR\Functional Global Variable\DVR Object.lvproj

DEMONSTRATION

Inter-process Communication – By Reference

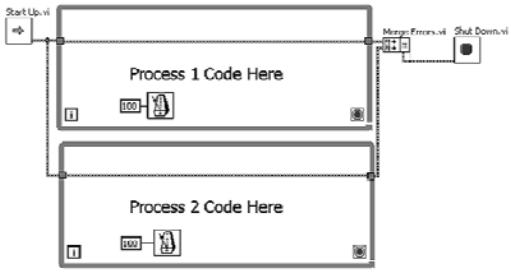
Recall the Lesson 2 discussion on passing state information in an API.



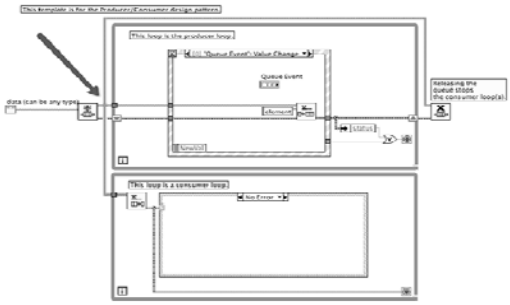
Why Use References? – Summary

- Access data from more than one location
- Control the creation and destruction of data
- Protect data (encapsulation)
- Manage multiple instances of an object

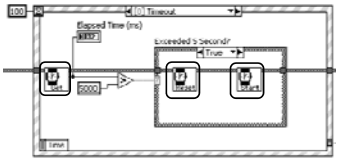
Why?– Access Data from More than One Location



Why?– Control Creation and Destruction of Data



Why? – Protect Data (Encapsulation)



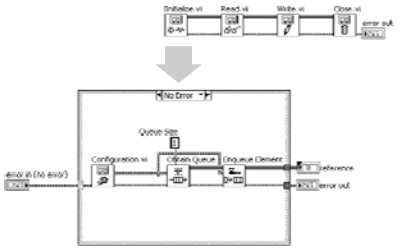
- Developer cannot unbundle a reference wire
- However, developer can still access data outside the Timer VIs (with Dequeue or In Place Element Structure)
- For better encapsulation, leverage the access scope of an `.lvlib`

Advantages of Common Reference Solutions

- Single Element Queue (SEQ)
  - Can name the reference
  - Get functions do not require blocking the data
- Data Value Reference (DVR)
  - Easy to implement

By Reference Implementation – Single Element Queues (SEQ) Method

Process can be streamlined through scripting



Exercise 3-5: Using Single Element Queues (SEQ)

To build a Wind Turbine Class based on a Single Element Queue (SEQ) architecture.

GOAL

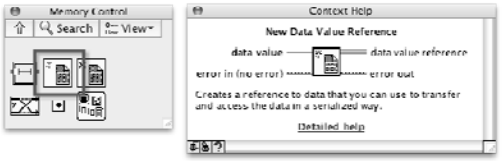
Exercise 3-5: Using Single Element Queues (SEQ)

- Could you follow the process, or did you find yourself lost at some point?
- Why did a specific order exist for the creating of the element data and then the creation of the queue reference?
- How would you change the templates?
- What other aspect of the process would you automate?

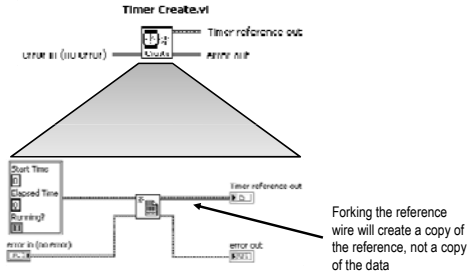
DISCUSSION

By Reference Implementation – Data Value Reference (DVR) Method

- Use DVR to place a reference wrapper around any data type (including LabVIEW classes)
- DVR acts as a pointer to the data

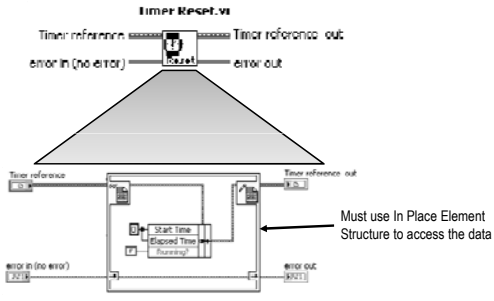


Data Value Reference – Timer



Forking the reference wire will create a copy of the reference, not a copy of the data

Data Value Reference – Timer



Must use In Place Element Structure to access the data

Exercise 3-6: Using Data Value References

To examine the design of a Timer class that was created with the Data Value Reference (DVR) feature in LabVIEW.

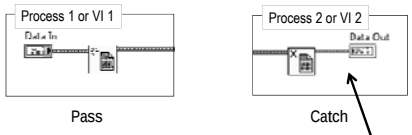
GOAL

Exercise 3-6: Data Value Reference

- How could you use the DVR today in your application?
- Where could the Timer DVR have helped you in an previous coding challenge?
- What would it take to automate the process of creating DVR classes?

DISCUSSION

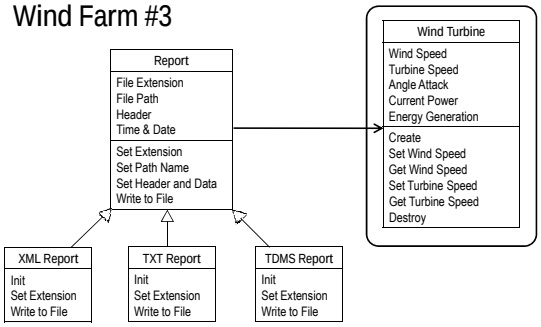
Data Value Reference – Simple Data Passing



Pass Catch

- Create a simple pass/catch design with DVRs
- Does not require the In Place Element Structure
- Delete Data Value returns the element referenced by the DVR

Wind Farm #3



<Exercises>\...\Wind Farm Course Project

DISCUSSION

Wind Farm #3

- Which of the previous concepts are illustrated in the solution?
- How is communication handled between the main UI and the wind turbines?
- How is the Wind Farm shut down?
- What is clear and what is unclear in this implementation?
- What would be a more clever solution?

DISCUSSION

Wind Farm #3

	Wind Farm #1	Wind Farm #3
User Interface	Dynamically launch a new window for each wind turbine	No wind turbine GUI - table on main UI displays wind turbine data
Data Communication Between UI and Turbines	VI Server (Set/Get Ctrl Value)	Single Element Queue
Shutdown Turbines	Poll FGV	User events

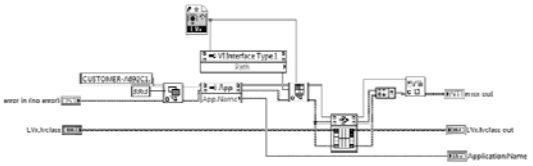
DISCUSSION

G. Inter-Target Communication

- Crosses the application instance and network boundary
- Can be used for local communication
- Good for design applications that must scale later
  - VI Server
  - Network Steams

Note: TCP is not covered in this course but falls under this category. You can use TCP to facilitate communication with non-LabVIEW systems across the network.

Inter-Target Communication – VI Server



- Call VIs across the network or application instances
- Simplifies communication across application instances or the network
- Good for LabVIEW-to-LabVIEW communication

Inter-Target Communication – Network Streams

Use case

- Stream buffered data between two processes in different applications or on different targets on a network
- High throughput

Caveat

- Both processes must be in a LabVIEW application

Network Streams

Network Streams Characteristics

- Lossless
  - Each network stream writes to and reads from FIFOs
- Unidirectional
  - Transfers data in only one direction for each network stream
- One-to-one communication channel
  - Each network stream consists of one writer and one reader

Network Streams

For more information, refer to the following topics of the *LabVIEW Help*:

- *Streaming Data and Sending Commands between Applications*
- *Network Streams Functions*

Standard Protocols

Use case

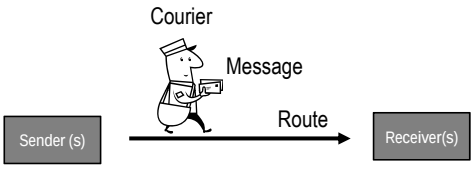
- Communicate with hardware and software that does not support LabVIEW
  - May already have a completed application that uses a standard protocol
- Standard protocol may be easiest solution
  - Implement a broadcast model using UDP protocol

Standard Protocols – Examples

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)
- Serial
- Web services (HTTP)

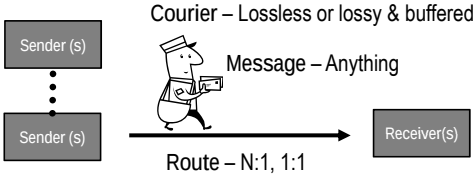
H. Analysis of LabVIEW Communication APIs

We use this frame of reference to review communication methods



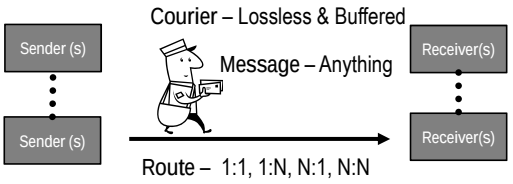
Scope:
Scalability:.
Performance:
Ease of implementation

Queues



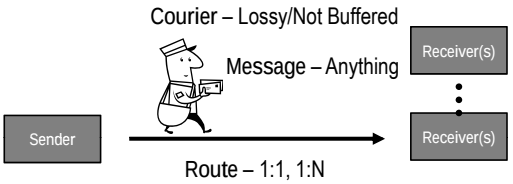
Scope: Same application instance
Scalability: Improves with API wrapper
Performance: Very fast
Ease of implementation: Introduced in early LabVIEW Core courses. Full-featured API.

User Events



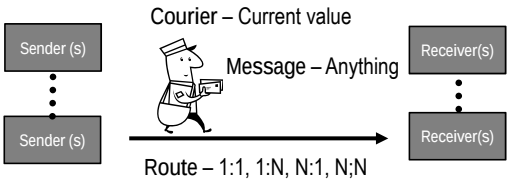
Scope: Same application instance
Scalability: Very scalable
Performance: Very fast
Ease of implementation: More difficult than queues. Lacks a full-featured API.

Notifiers



Scope: Same application instance
Scalability: Very
Performance: Very Fast
Ease of implementation: Reasonable. Not used as widely as queues or user events.

DVRs



Scope: Same application instance
Scalability: Very scalable with API
Performance: Fast
Ease of implementation: Not covered in Core courses, but not overly difficult.

Network Streams

Courier – Lossless/buffered

Message – Most data types (no classes)

Sender → Receiver(s)

Route – 1:1 only

Scope: Different application instances, different network targets
Scalability: Not as easy, but essential for streaming across instances and the network
Performance: Very Fast
Ease of implementation: Not too difficult. Trickier for connects that are not continuous.

NATIONAL INSTRUMENTS | ni.com/training

I. Communication Reference Architectures

	Same target Same application instance	Same target, different application instances OR Different targets on network
Storing - Current Value	• Single-process shared variables • Local and global variables • FGV, SEQ, DVR • CVT • Notifiers (Get Notifier)	• Network-published shared variables (single-element) • CCC
Sending Message	• Queues (N:1) • User events (N:N) • Notifiers (1:N)	• TCP, UDP • VI Server • Network Streams (1:1) • AMC • STM (1:1) And More...
Streaming	• Queues (N:1)	• Network Streams • TCP

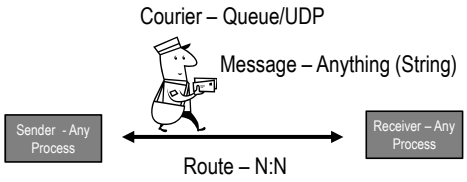
NATIONAL INSTRUMENTS | ni.com/training

Inter-Target Communication Reference Architecture – Return to AMC

- Can use the AMC API to implement QDMHs and communicate between QDMHs
  - Single QDMH
  - Multiple QDMHs on the same target
  - Multiple QDMHs on different targets on a network

ni.com/training

AMC

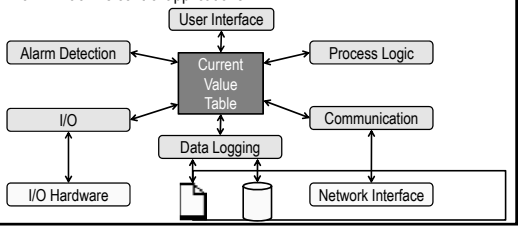


Scope: Same application instance, different application instances, different network targets
Scalability: Easy to add processes. Message format is defined and scalable. Bi-directional by design.
Performance: Good for messages, not streaming
Ease of implementation: Will require training, but does leverage foundational concepts.

Current Value Table

Use case

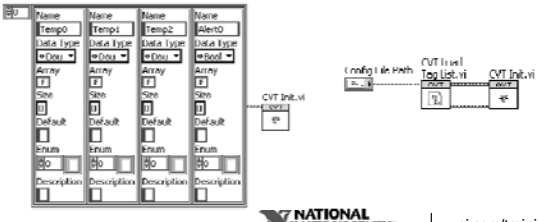
- Centralizes operations with data shared by many processes within the same application
- Allows many application components to share a common data repository
- Allows direct access to latest values of data
- Similar to a collection of single process shared variables
- Common in machine control applications



Current Value Table Example

Initialization Examples

- Programmatically initialize variable names, data type, and default value
- Initialize variables from file



Current Value Table Example

- Download *Current Value Table (CVT) Reference Library* example from `ni.com`
- Installs CVT VIs to User Library palette

Current Value Table Example

Dynamically write and read groups of variables using an example CVT implementation.

`<Exercises>\...\Adv Design Patterns and Tools\CVT`

DEMONSTRATION

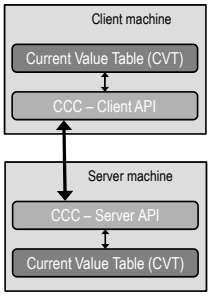
Inter-Target Communication Reference Architecture – CVT Client Communication (CCC)

Use case

- Share data between CVTs on multiple applications or targets
  - Host computer sends setpoints to Real-Time target
  - Real-Time target sends temperature readings to host computer
- Extend the advantages of CVTs to network communication
  - Dynamically create variables at run-time
  - Dynamically perform operations on large groups of variables

CVT Client Communication (CCC)

Complex to develop
- Client
  - Writes to and reads from server CVT
- Server
  - Listens for client commands
- CCC API
  - Allows client to bind its local CVT values to server CVT values
  - Transfers values between client and server CVT

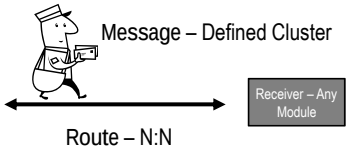


CVT Client Communication (CCC) Example

- Download *CVT Client Communication (CCC) Reference Library* example from `ni.com`
- Installs CCC VIs to User Library palette

Messages – CVT/CCC

Courier – FGV/TCP

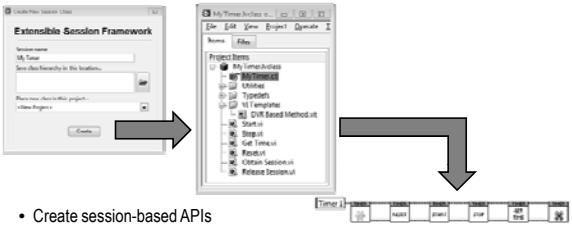


Scope: Same application instance, different application instances, different network targets.
Scalability: Can call from any module.
Performance: Reasonable. Not for streaming.
Ease of implementation: Uses FGVs and STM reference library. Not very difficult.

Other Messaging Reference Architectures (Optional)

- ESF (NI)–Extensible Session Framework
- REx (NI)–Remote Export Framework
- Message Routing Architecture (LabVIEW community)–Implementation of Observer Pattern

ESF – Extensible Session Framework (Optional)

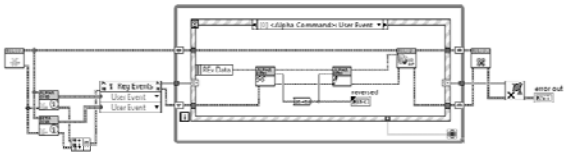


- Create session-based APIs
- Wrap a named DVR around a class
- Wizard simplifies creation and extension of methods

<Exercises>\…\Adv Design Patterns and Tools\Installers for REx and ESF

DEMONSTRATION

REx – Remote Export Framework (Optional)



- Messaging across application instance/target
- Includes the ability to have a message response
- True bi-directional communication
- Clean API abstracts the complexity

<Exercises>\…\Adv Design Patterns and Tools\Installers for REx and ESF

DEMONSTRATION

Message Routing Architecture (Optional)

- A sophisticated architecture designed by the LabVIEW community
- Implements an Observer design pattern
- Eliminates the processing overhead of a hub and spoke architecture
- Observers subscribe to receive new messages
- Based on LabVIEW Classes

<Exercises>\…\Adv Design Patterns and Tools\Message Routing Architecture

DEMONSTRATION

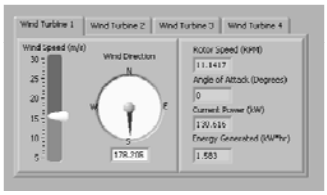
Lesson 4
Advanced User Interface Techniques

TOPICS

A. Subpanels – Examples
B. XControls Overview – Review of Controls
C. XControl Abilities
D. XControl Properties and Methods
E. XControl Development – Editing XControls in Use

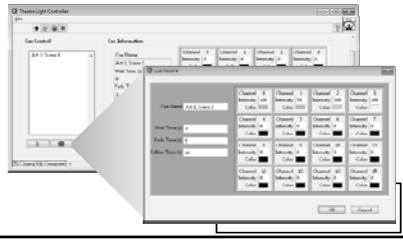
Using Tabs in User Interface Design

What are the benefits and limitations of tab controls when designing scalable and extensible user interfaces?

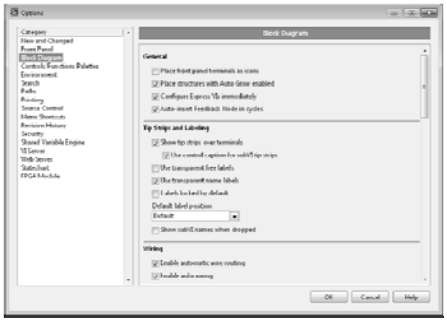


Using Pop-Up Displays in User Interface Design

- Do Pop-Up displays improve scalability?
- Does they enforce modularity better than tabs?
- When might they fall short for user interface design?

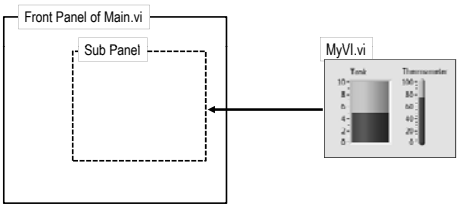


A. Subpanels – Example



What is a Sub Panel?

- A subpanel control is a container that can display the front panel of another VI
- VIs can be dynamically inserted and removed from the subpanel



Advantages of Using Sub Panels

- Flexible–UI can be anything
- Modular–VIs are independently coded and tested
- Simple–May not require communication back to main UI
- Clean UI–Decrease clutter on the front panel
- Extensible–Easily add additional UIs to run in the sub panel
- Scalable–Unlimited instances of a VI

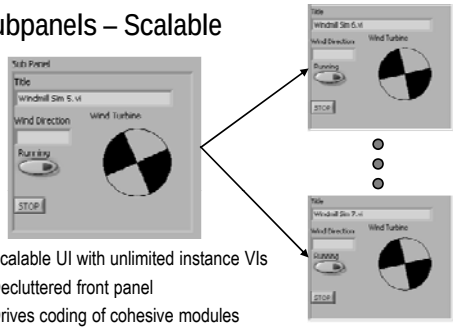
Wind Farm #2

- Wind turbine VI front panels displayed in a subpanel

`<Exercises>\...\Wind Farm Course Project`

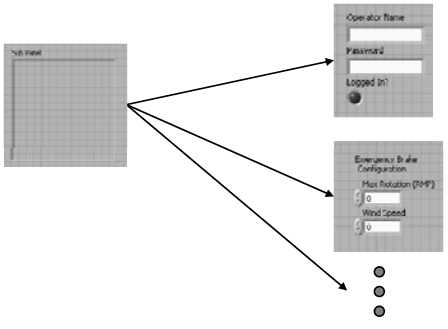
DISCUSSION

Subpanels – Scalable



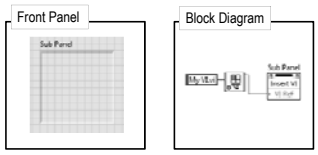
- Scalable UI with unlimited instance VIs
- Declluttered front panel
- Drives coding of cohesive modules

Subpanels – Extensible



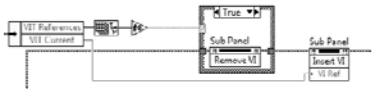
Subpanels – Getting Started

1. Create a subpanel control.
2. Obtain a reference to the VI to insert.
3. Insert the VI.



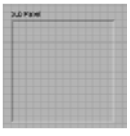
Note:
- If the front panel of My VI.vi is already open, the Invoke node outputs an error
- The Invoke node does not run My VI.vi

Subpanels – Require VI Reference Management

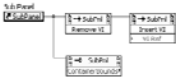


- Design a framework to handle the references for VIs that might be inserted into the subpanel
- Ensure that all asynchronous processes shut down at the appropriate time

Subpanel References



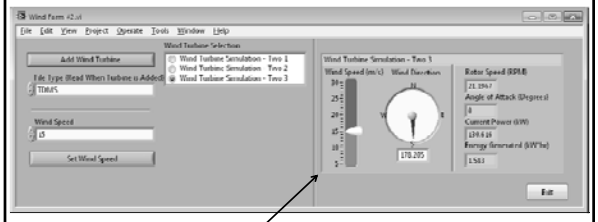
Explicitly linked



Implicitly linked

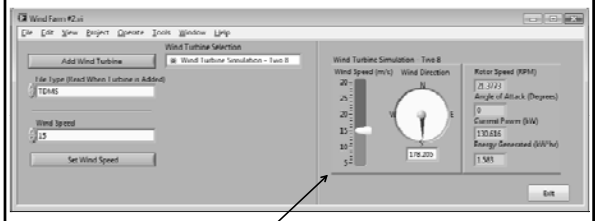
Considerations
- Create a control reference to access more properties and methods
- Consider sizing the subpanel to the front panel of the inserted VI

Integration of Subpanel – Frame is Visible



Subpanel frame defines the border of the VI to be inserted

Hide Subpanel Frame



Use the system subpanel to hide the frame
Use decorations to group the controls

Docking and Undocking Demonstration

Show a creative way of using subpanels

`<Exercises>\…\Demonstrations\Subpanels – Dock Undock\Main_CP_example.vi`

DEMONSTRATION

Subpanels – Additional Design Considerations

- Will the inserted VI run continuously in the background?
- If the VI does not run continuously, how does the VI stop execution?
- What is the lifetime of the inserted VI?
- Is the inserted VI autonomous or does it communicate with the main VI?
- Should the subpanel be integrated with tabs?
- Subpanels add scalability and complexity to the code. What is best for your application?

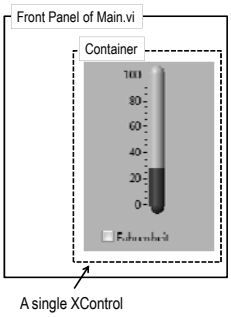
B. XControls Overview – Review of Controls

Types of controls

- Custom Control (review)
  - Provides a template for the appearance of visible controls
- Type Definition (review)
  - Enforces the data type of the control, commonly used on invisible controls or block diagram constants
- Strict Type Definition (review)
  - Enforces the data type and appearance of visible controls
- XControl
- Others (ActiveX, .NET)

What is an XControl?

- XControls allow the architect to create controls with custom appearance and edit-time and run-time functionality
- For the user, the XControl has the same functionality as other controls



XControl

Simple Xgraph

<Exercises>\…\Demonstrations\XControl\Simple XGraph\Test X Graph Simple XControl.vi

DEMONSTRATION

Designing an XControl

- What is an XControl?
- What aspects of an XControl need to be defined?

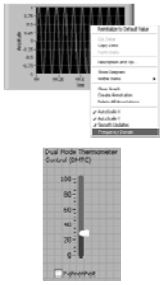
DISCUSSION

Advantages of XControls

- Encapsulate UI code into an abstract component
  - Reduce complexity and increase readability of main block diagram
  - Remove UI-based restraints from the architecture of the main VI
  - Abstract complex code away from standard developers
- Create reusable and distributable UI components
  - Easily distributable
  - Update control instances to new versions while maintaining integrity
- Can have custom dynamic run-time and edit-time behavior

Typical Uses of XControls

- Add Functionality to Existing Controls
  - New properties or methods
  - Inherent data analysis or processing
- Combine Existing Controls
  - Combine multiple controls into a single data type, abstracting out code and information related to the display of the controls



Typical Uses of XControls (continued)

- Abstract UI Components
  - Use event-based programming without using an event-based design pattern
  - Separate an entire UI from the rest of the implementation
- Create New Controls
  - Use the picture control
  - Custom graphs are a common example

When to Use XControls

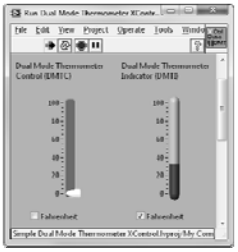
When you **should** use XControls:
- To create reusable components with dynamic behavior
- To encapsulate extended functionality of a control

When you **should not** use XControls:
- To accomplish purely cosmetic changes
- When working on a single-shot application that will not use the XControl is several places

Use Strict Type Defs **instead** when:
- You do not need dynamic run-time and edit-time behavior

Another XControl Example...

Design a thermometer control that can represent a single numeric input in either Celsius or Fahrenheit



- What makes this a good application for an XControl?

DISCUSSION

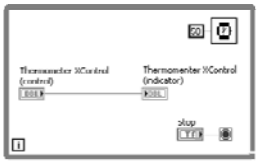
Dual Mode Thermometer XControl

Demonstrate the `Dual Mode Thermometer XControl` example in the NI Example Finder.

DEMONSTRATION

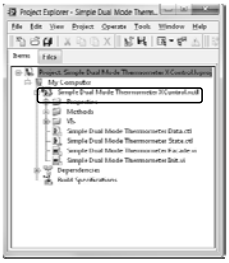
Using XControls

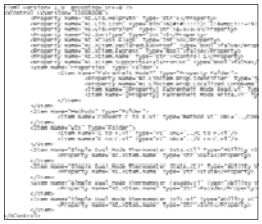
- Manage XControls via the Project Explorer
- XControls appear as regular terminals
- You start with shell code



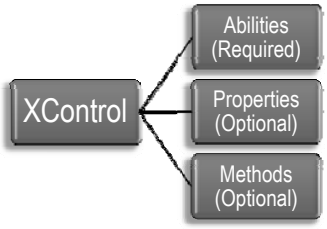
XControl Library File

XControl library file (`.xctl`) is an XML file that includes the names and locations of each component



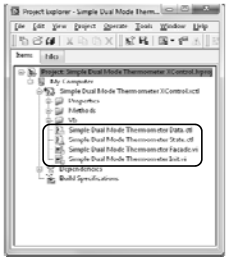


Structure of an XControl



C. XControl Abilities

- Required components for proper function
- Represented by VIs or Controls
- Four required abilities:
  - Data
  - State
  - Facade
  - Init
- Additional optional abilities exist



Exercise 4-1: Creating an XControl (Optional)

To create an XControl that functions as a scalable radio control button.

GOAL

XControl Data Ability

Developing the XControl | Using the XControl

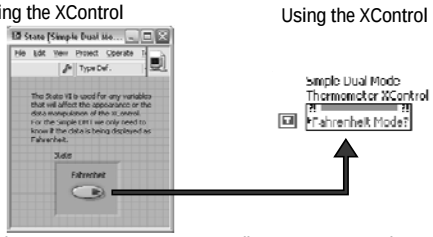
Data.ctl

- Specifies the data type of the XControl
- What should it look like on the block diagram?

XControl State Ability

Developing the XControl | Using the XControl

State.ctl

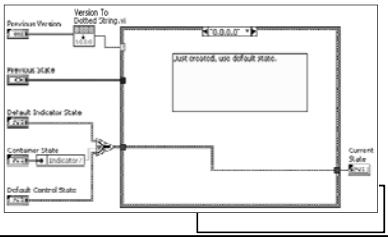


- Specifies information other than data type that affect the appearance of the XControl
- What information is necessary for describing the XControl?
- What methods and properties do you need for your XControl?

XControl Init Ability

- Called upon first placement or load into memory
- Initializes display state before being displayed
- Handles control versioning

Init.vi



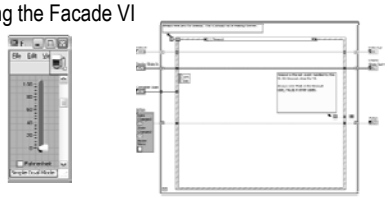
Exercise 4-2: Modifying X Listbox Abilities (Optional)

To modify the Data, State, and Init abilities of the X Listbox control you created in Exercise 4-1.

GOAL

XControl Facade Ability

- Determines the appearance and operating behavior of the control
- You spend most of your XControl development effort developing the Facade VI

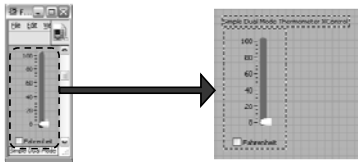


XControl Facade Ability – Front Panel

Front panel window defines the appearance

- Visible area becomes the XControl container border
- User can interact with and view only controls and indicators in the visible area

Facade.vi



NATIONAL INSTRUMENTS | ni.com/training

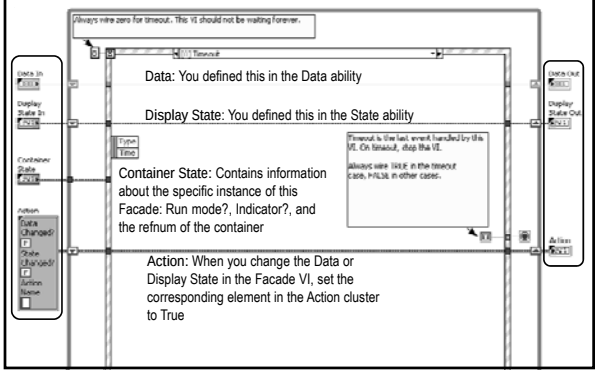
Facade Ability – Block Diagram

Block diagram defines how the XControl responds to events, properties, methods, and certain user interactions

- Consists of an event handler design pattern
- Does not run continuously
- Called to handle any event on the Facade front panel
- Called to process special events at defined times

ni.com/training

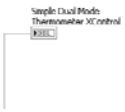
Facade: Inputs and Outputs



XControl Events Handled by the Facade VI

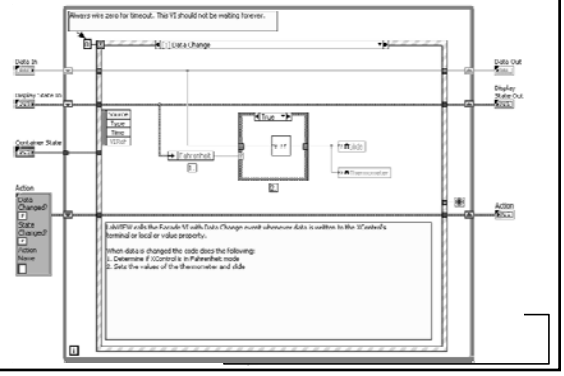
- Data Change event
- Display State Change event
- Direction Change event
- Execution State Change event
- Facade front panel events

Facade Ability – Data Change Event

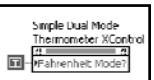


- Event Generation
  - Generated when the value of the XControl changes as a result of writing to its terminal, local variable, or Value property
  - Not generated when other events set the Data Changed? element of Action
- Purpose
  - Performs any formatting or modification of the data
  - Updates the state and control values as necessary

Facade Ability – Data Change Event

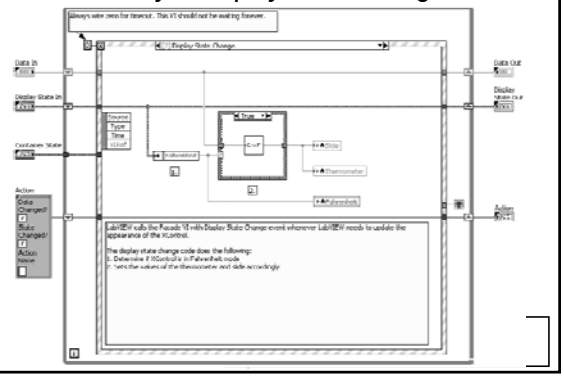


Facade Ability – Display State Change Event

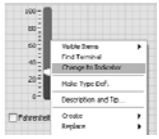


- Event generation
  - Generated when the display state of the XControl changes, usually after executing a property or method or from the Init ability
  - Not generated when other events set the State Changed? element of Action
- Purpose
  - Uses the values in the state to update the appearance of the front panel
  - Requires handling the Display State Change event for the XControl to function properly

Facade Ability – Display State Change Event

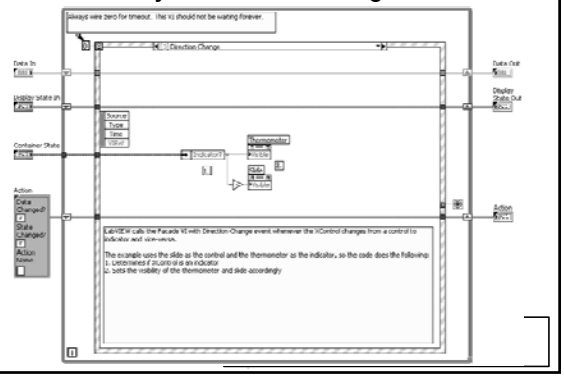


Facade Ability – Direction Change Event



- Event generation
  - Generated when an instance of the XControl changes from a control to an indicator or vice versa
- Purpose
  - Updates the appearance and properties of the control
    - Modifies the properties of controls
      <OR>
    - Puts both a control and an indicator on the front panel and makes the correct object visible

Facade Ability – Direction Change Event

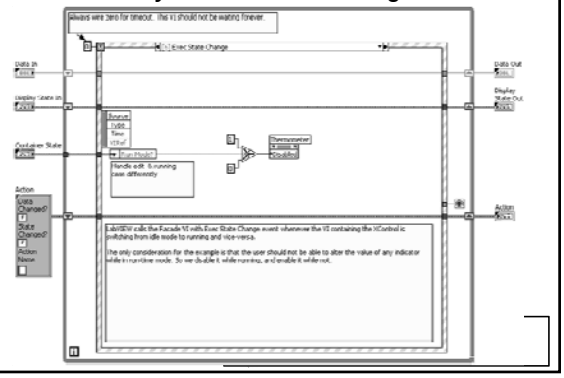


Facade Ability – Exec State Change Event

- Event generation
  - Generated when a VI containing the XControl changes from edit mode to run mode or vice versa
- Purpose
  - Makes modifications to the appearance or properties of the control if necessary
  - Use the Run Mode? element of Container State to determine the execution state (also works in other events)

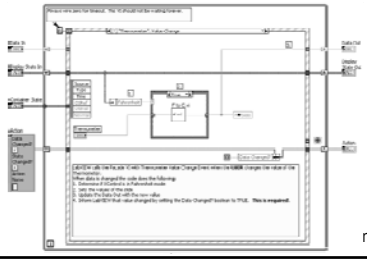
NATIONAL INSTRUMENTS | ni.com/training

Facade Ability – Exec State Change Event



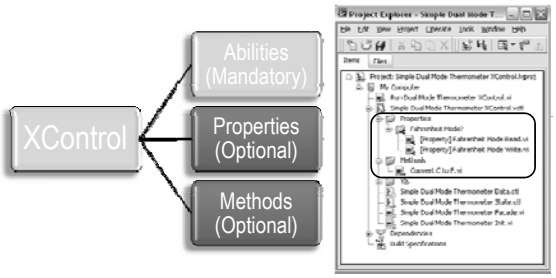
Facade Ability – Front Panel Events

- Handles events for the controls on the front panel of the Facade
- Can use events to update the Data and/or State ability



ni.com/training

D. XControl Properties and Methods

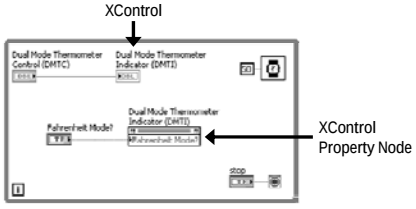


NATIONAL INSTRUMENTS | ni.com/training

XControl Properties

- Allow the user to configure the XControl programmatically via Property Node

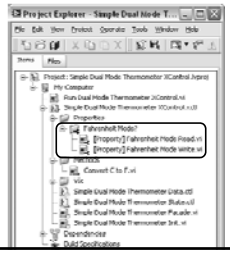
Main VI containing the XControl



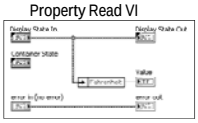
NATIONAL INSTRUMENTS | ni.com/training

XControl Properties (continued)

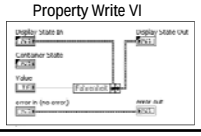
- To create, right-click the XControl library and select New»Property
- Operate by accessing or changing the state
- If executing a property changes the display state, a Display State Change event is generated on the Facade VI



Property Read VI



Property Write VI



ni.com/training

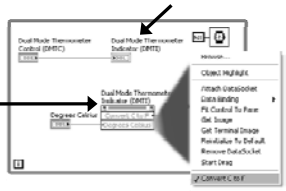
XControl Methods

- Allow the user to engage functionality of the XControl programmatically via Invoke Node

Main VI containing the XControl

XControl

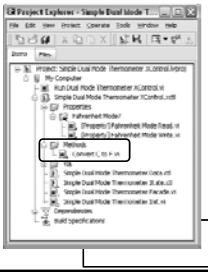
XControl Invoke Node



NATIONAL INSTRUMENTS | ni.com/training

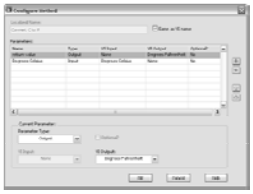
XControl Methods

- Create by right-clicking the XControl library and selecting New»Method
- Configure by right-clicking the Method and selecting Configure Method

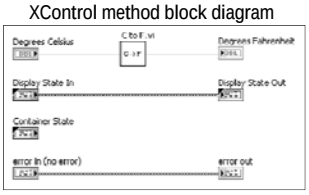


ni.com/training

XControl Methods (continued)



XControl method block diagram



- Configuration defines which connector pane terminals are available from the Invoke Node
- Operate by accessing or changing the state
- If invoking the method changes the display state, a Display State Change event is generated on the Facade VI

Exercise 4-3: Creating X Listbox Properties and Methods (Optional)

To create XControl properties and methods.

GOAL

How an XControl Works

- An XControl is similar to a subpanel that hosts the Facade VI.
- The Facade VI is called only when there are events that require handling.
- Data, Display State, and Container State are stored by the parent VI and passed each time the Facade VI is called.
- The Display State information is saved with the parent VI on disk.
- The XControl may run in its own context and does not share data space with the VI using it.

E. XControl Development – Editing XControls in Use

- When an open VI uses an instance of an XControl, the XControl is locked from editing
- You can edit the control by closing the VI using it, or by right-clicking the XControl and selecting **Unlock Library for Editing**
- While the XControl is being edited, XControl instances are broken
- Right-click the library and select **Apply Changes to Instances** to finish editing

XControl Development – Debugging XControls

- To debug the Facade VI:
  - Go to the front panel of the VI using the XControl
  - At edit-time: Right-click the XControl and select **Advanced»Show Block Diagram**
  - At run-time: Right-click the XControl and select **Show Diagram**
- To debug abilities, properties, and methods:
  - Set a breakpoint in the ability, property, or method
- Use normal debugging tools and techniques such as single-stepping, probes, and breakpoints

XControl Development – Distributing XControls

- Like any other custom control, VIs using an XControl must have access to the XControl file library.
- All files used by the XControl library must be present and accessible.
- Create an installer to distribute XControls to the `<labview>\user.lib` directory to avoid confusion.
- Create a palette file to hide all the XControl files except the `.xctl`.
- Create an icon for the control under the control properties page.

XControl Programming Caveats

- The appearance of the XControl should depend only on the container state, display state, and data
  - Do not use variables, shift registers, or synchronization mechanisms in the Facade VI or other XControl VIs
  - Use the State ability to store data which needs to persist between calls to the Facade VI
- Avoid blocking calls
  - Can interfere when you are editing VIs that use the control
  - Can hang the development system in extreme cases
  - Set the cursor to busy for any intense processing

XControl Design Issues

- Consider building the test VI for the XControl first
- The architect is responsible for fixing bugs and issues that other developers and users of the XControl may find
- XControls can significantly improve scalability by modularizing UI components into reusable libraries
- XControls may initially appear to be more complex than simple UI coding

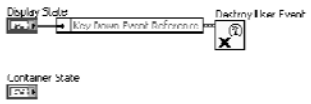
Exercise 4-4: Creating the X Listbox Facade VI (Optional)

To create the Facade VI.

GOAL

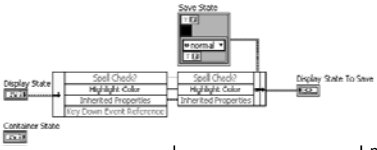
Uninit Ability (Optional)

- Optional VI that cleans up and closes XControl resources
- Right-click XControl and select New»Ability to add
- Release any resources allocated in Init and stored in the State ability



Convert State for Save Ability (Optional)

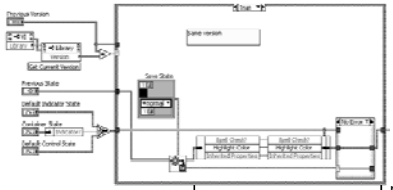
- Modifies the state before saving to disk
- State can contain non-persistent data
- Remove any non-persistent data by creating a Save State cluster and converting the Display State to the Save State
- Removing non-persistent data saves space



Convert State for Save Ability (Optional)

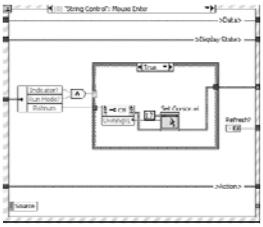
Requires Init changes

- Modify the Init ability to convert the Save State back to a Display State
- Stored state data is lost if not properly converted



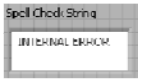
XControl Programming Techniques – Accessing the Parent VI

- Use the Refnum input of the Container State to access the container holding the XControl
- Use the Owning VI property of the container to access the VI using the XControl
- Use carefully. Suggested uses:
  - Getting information about the VI
  - Setting the cursor when over the XControl



XControl Programming Techniques – Error Handling

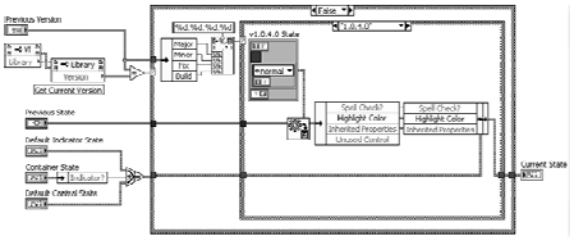
- XControls do not have a built-in mechanism for error handling
- Error handling techniques
  - Corrective error handling
  - Alter XControl Appearance
  - Error Dialogs (use with caution)
  - Custom Events



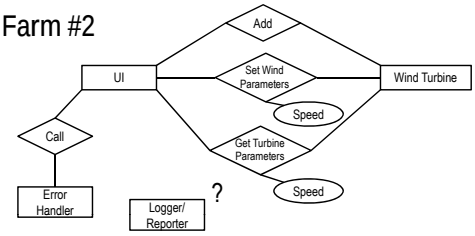
XControl Programming Techniques – XControl Version Control

- After an XControl is used in a VI, state information is stored with the VI
- Changing the state of the XControl can cause the state to become incompatible, reverting all values to the default
- Manage the versioning of XControls to avoid losing state information

XControl Programming Techniques – XControl Version Control (continued)



Wind Farm #2



- Which of the previous concepts are illustrated in the solution?
- How is communication handled between the main UI and the wind turbines?
- How is the Wind Farm shut down?
- What is clear and what is unclear in this implementation?
- What would be a more clever solution?

`<Exercises>\...\Wind Farm Course Project`

DISCUSSION

Wind Farm #2

	Wind Farm #1	Wind Farm #2	Wind Farm #3
User Interface	Dynamically launch a new UI for each wind turbine	Use XControl to select which wind turbine to view. Subpanel displays selected wind turbine.	No wind turbine GUI. Table on main UI displays wind turbine data.
Data Communication Between UI and Turbines	VI Server (Set/Get Ctrl Value)	Queues	SEQ
Shutdown Turbines	Poll FGV	Queues	User events

DISCUSSION

Lesson 5
Introduction to Object-Oriented Programming in LabVIEW

TOPICS

A. What is Object Oriented Programming?
B. Recommended Resources
C. Class Diagrams and Design (Optional Section)

A. What is Object Oriented Programming?

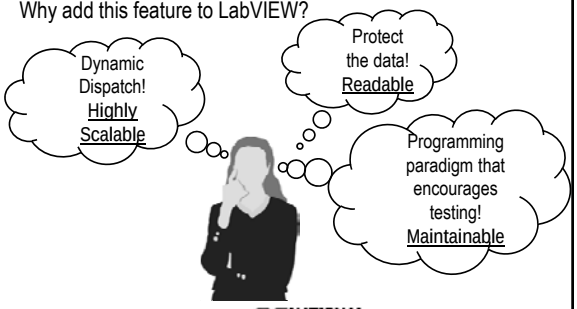
OO is a programming construct in which data is grouped with the functions that operation on that data

OO emphasizes:

- Encapsulation– Only the methods grouped with the data can access it so the data is protected
- Inheritance–Methods are dynamically called at run time without the need to create a special framework

LabVIEW Object-Oriented Programming

Why add this feature to LabVIEW?



LVOOP

Explore the dynamic dispatch VIs in an application using LVOOP

`<Solutions>\...\Exercise 5-1\LVOOP Report.lvproj`
Test LVOOP Plug Ins – Challenge.vi
"Save to File" case

DEMONSTRATION

OOP Terminology

Class — A collection of data and the methods that interact with that data

- Classes are actors or things acted upon
- Classes are NOUNS
- Classes are data types
- Classes are similar to clusters with special functionality

OOP Terminology

Object—A specific instance of a class

- Refers to individual pieces of data
- There may be many objects (instances) of a given class

Examples of Classes and Objects

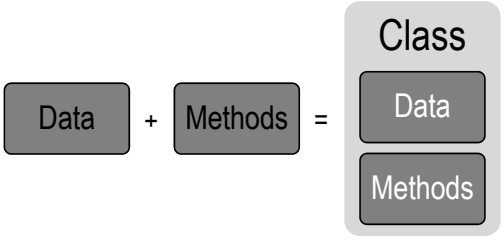
Class:	Car	Report
Object:	• John's 1965 Ford Mustang • Jane's 2004 Honda Accord	• Wind Turbine One Report • Wind Turbine Two Report

OOP Terminology

Method—An action that can be performed using the data of a class.

- Methods are performed by objects
- Methods are VERBS
- Methods are VIs
  - Use or modify the data in an object

Class as a Data Type



Class Data and Methods

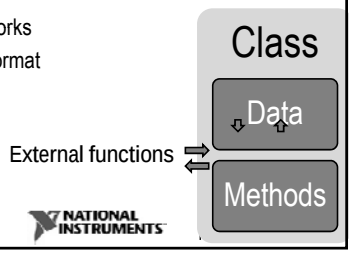
	Car	Report
Class		
Data	Make Model Year Mileage	Header Date & Time Data Array
Methods	Calculate Sell Value Rotate Tires Change Oil	Set Header and Data Update Data Set Extension Write to File

NATIONAL INSTRUMENTS | ni.com/training

OOP Terminology

Encapsulation—Consolidation of data and methods into a class, with restricted access to data

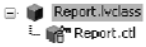
- Hides how an object works
- Easy-to-change data format



NATIONAL INSTRUMENTS

LVOOP Basics – Creating a LabVIEW Class

Step 1

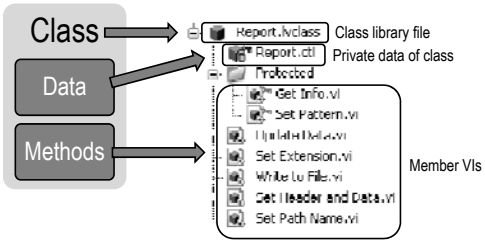


Step 2

Step 3

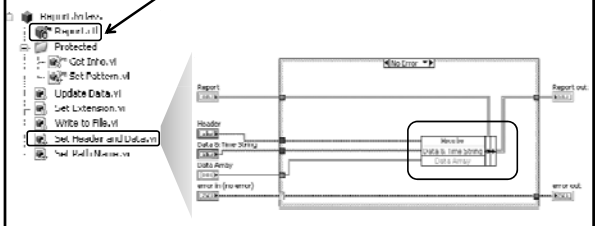
NATIONAL INSTRUMENTS | ni.com/training

LVOOP Basics – LabVIEW Class



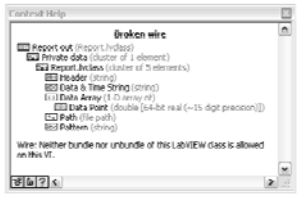
LVOOP Basics – Encapsulation and Libraries

Report.ctl. is Class Private Data – it is "private" and can only be accessed by a Method that is a Member of the class.

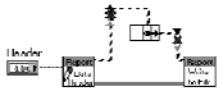


LVOOP Basics – Encapsulation

Limited Access to the Data

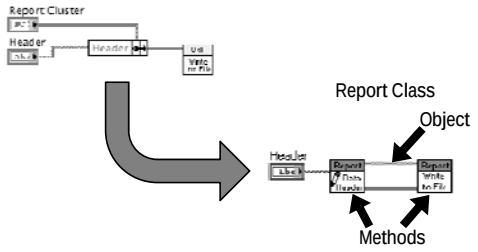


Block diagram of VI that is not a member of the Report class



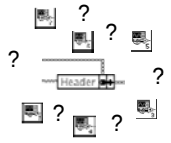
Migrating to OOP

For those who choose object designs, wire and node should morph naturally into class and method



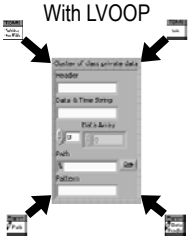
Encapsulation – Simplifies Debugging

Without LVOOP



How many VIs access the cluster? How long could it take to find the source of the bug?

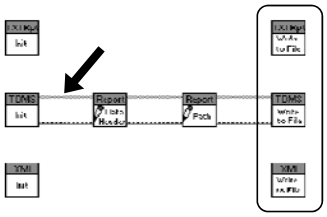
With LVOOP



Only the method VIs touch the data. The developer's search for the source of the error narrows significantly.

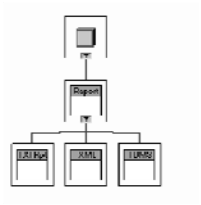
Inheritance – Dynamic Dispatch

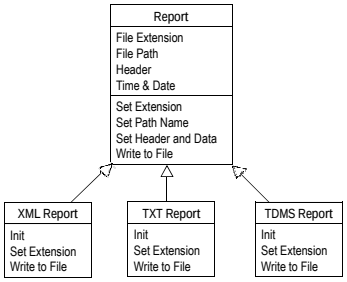
In essence, dynamic dispatch is polymorphism at run time…



The object type determines which VI is dispatched at run time.

Inheritance – Class Hierarchy





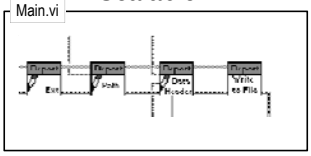
Dynamic Dispatch - Scalability

- Main code can call VIs from the parent class
- Then, new child classes can be added so that no change to the main code is required
- The only requirement is that the new child class is fully tested.

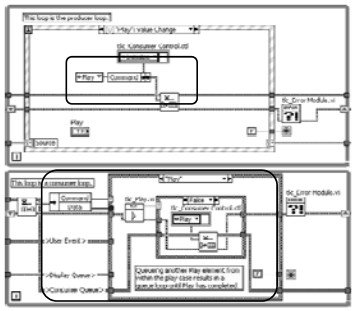
1. XML and TXT implemented.

2. TDMS added with no changes to Main.vi.

Scalable!



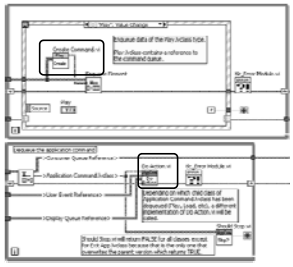
Migrating to OOP – Replacing Enum/Variant Clusters



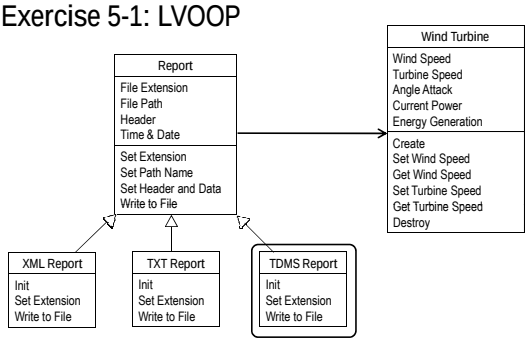
Migrating to OOP – Replacing Enum/Variant Clusters

Replace TLC enum+variant clusters with:

- Application Command.lvclass and its child classes
- Class methods
  - Create Command.vi
  - Do Action.vi
  - Should Stop.vi



Exercise 5-1: LVOOP



Add a sibling class to an existing class

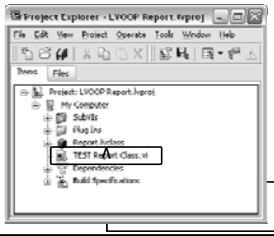
GOAL

Exercise 5-1: LVOOP

DISCUSSION

LVOOP Benefits – Testing

- Yes, the initial development of a class may take a little extra time
- Yes, you should create the VIs to test the class
- Yes, the effort pays off once you extend the class to add new features. A successful retest of a class with new features, children, or siblings increases the likelihood of successful integration into the main application



B. Recommended Resources

- *Applying Common OO Design Patterns to LabVIEW* on NI Community – information and examples for several OO design patterns

Design Pattern	Usage
Channeling Pattern	Guarantee pre-processing/post-processing around some dynamic central functionality
Factory Pattern	Provide a way to initialize the value on a parent wire with data from many different child classes based on some input value, such as an enum or string input
Command Pattern	Uses child classes to represent messages (command and data) that need to be sent to a consumer loop, which executes the functionality associated with the message
Hierarchy Composition Pattern	Develop a single object as a tree of smaller instances of that same object type
And more…	

Recommended Resources

Actor Framework

- Software library containing classes that supports the writing of applications in which multiple VIs run independently while communicating with each other
- `<Exercises>\Advanced Architectures in LabVIEW\Adv Design Patterns and Tools\Actor Framework`

Recommended Resources

- *Object-Oriented Design and Programming in LabVIEW* course – covers the fundamental concepts of object-oriented design and programming and then discusses how to implement those concepts in LabVIEW
- *LabVIEW Object-Oriented Programming* topic of *LabVIEW Help* – detailed conceptual and procedural information for LVOOP
- *LabVIEW Object Oriented Programming FAQ* on ni.com – answers many questions and provides useful related links, such as whitepapers, presentations, videos, and exercises
- *LabVIEW Object Oriented Programming: The Decisions Behind the Design* on ni.com/zone – covers the design issues behind the LVOOP implementation of object-oriented programming
- Refer to `www.expressionflow.com` for several good articles, including one on plug-ins
- Explore the many threads on `www.lavag.org`

Wind Farm #3

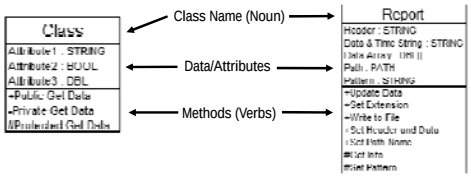
- Uses LVOOP to implement different types of reports in the wind turbine VIs

`<Exercises>\...\Wind Farm Course Project`

DISCUSSION

C. Class Diagrams and Design (Optional Section)

Unified Modeling Language (UML) provides a structured syntax for documenting classes:



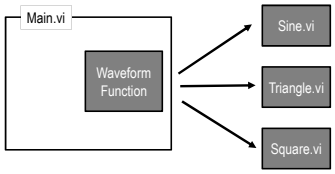
Lesson 6
Plug-In Architectures

TOPICS

A. Using Plug-In Architecture with VI Server
B. Using Plug-In Architecture with LVOOP

What are Plug-Ins?

- Plug-in architectures allow the extension of features without changes to the main code
- New modules are detected and linked into the application dynamically while the code is running.



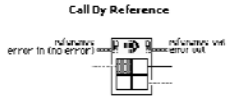
Advantages of Plug-Ins

- Allow addition of features without re-running a build
- Encourage you consider ways an application might change
- Facilitate good programming habits by coding to an interface
- Examples
  - Add additional test VIs to folder containing plug ins
  - Add a VI that uses a new instrument without needing to change main VI code

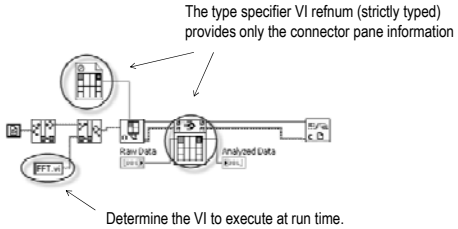
A. Using Plug-In Architecture with VI Server

Call By Reference Node

- Calls the VI specified by reference
- The specified VI is no longer implicitly part of the block diagram. Instead, the application can dynamically determine which VI to call at run time, not edit time



Call By Reference Node – Strictly Typed VI



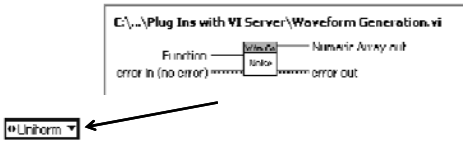
Plug-Ins with VI Server

Explore an application that implements a plug-in architecture using VI Server.

`<Solutions>\…\Exercise 6-1\Plug Ins with VI Server.lvproj`

DEMONSTRATION

Design Considerations



- Will the Plug-In architecture truly be extensible if an enum is used for the function input?

DISCUSSION

Plug-In VI Design Issues

- Remember that changing the connector pane will break the interface
- Utilize scalable data types such as strings and variants
- Consider adding a spare input for future features
- Consider a different mechanism for finding the Plug-Ins, such as a naming convention.
- If the connector pane will not be constant, but the plug-in VI will not run asynchronously, use the Run VI method.
- Remember to have a defined place to locate the Plug-Ins

Exercise 6-1: Using Plug-Ins with VI Server

To evaluate the key components of a plug-in architecture implemented with VI Server and to add a plug-in to the existing architecture.

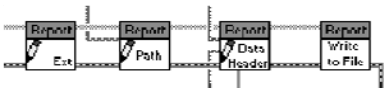
GOAL

Exercise 6-1: Using Plug-Ins with VI Server

DISCUSSION

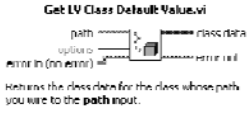
B. Using Plug-In Architecture with LVOOP

- An entire group of VIs becomes dynamic. One change on an Init of the class changes the functionality of all subsequent VIs
- Dynamic Dispatch is powerful!



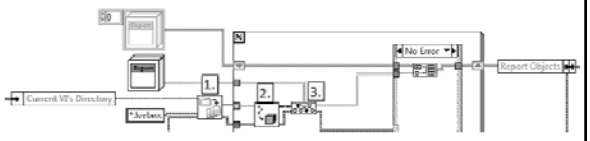
Key VI For Implementing LVOOP Plug-Ins

This VI is the power behind LVOOP Plug-Ins



Implementing LVOOP Plug-Ins

1. Get all paths to all .lvclass files in a specific directory.
2. Get the default value for the class.
3. Check to see if it can be “made more specific” to the parent.



Exercise 6-2: Using LVOOP Plug-Ins

To write the calling code to allow new LVOOP sibling classes to be plugged in at run time.

GOAL

Exercise 6-2: Using LVOOP Plug-Ins

DISCUSSION

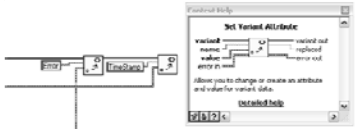
Lesson 7
Tips, Tricks, & Other Techniques

TOPICS

A. Variant Attributes
B. Callback VIs
C. VI Scripting
D. What is a Drop In VI?
E. Other

A. Variant Attributes

- Very flexible mechanism for storing data
- Accepts any data type
- Lookup mechanism is a hash table
- Can replace the core of an FGV for implementations that store a large number of elements



Variant Attribute

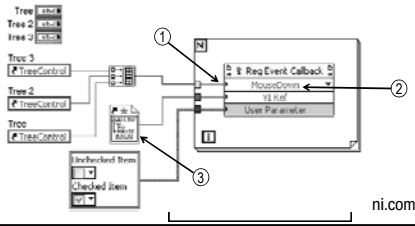
<Exercises>\…\Demonstrations\Variant Attribute\Name Value Lookup.vi

DEMONSTRATION

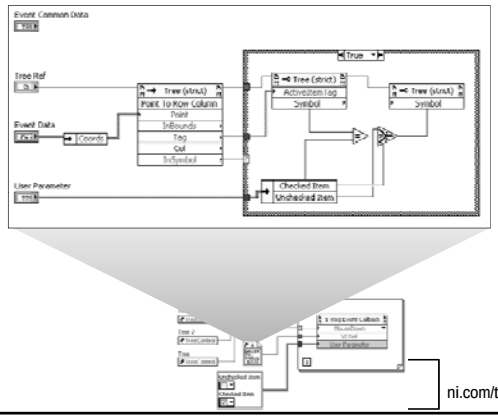
B. Callback VIs

A callback VI runs when a specified event occurs

1. Wire the reference of the event source.
2. Select the event.
3. Create and edit the callback VI from the VI Ref input.



Callback VI



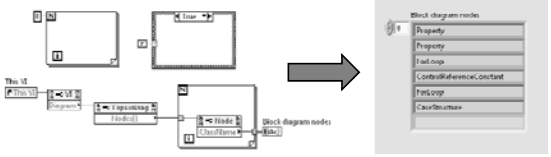
Callback VI

DEMONSTRATION

C. VI Scripting

Use VI Scripting VI and functions with the associated properties and methods to create, edit, and run VIs programmatically

VI Scripting



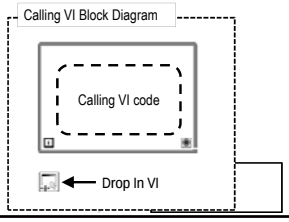
VI Scripting – More Information

- Refer to the *Programmatically Scripting VIs in LabVIEW* topic of the *LabVIEW Help* and NI Example Finder
- JKI Software is facilitating scripting with the JKI Right-Click Framework for LabVIEW:
  `http://decibel.ni.com/content/groups/labview-apis`

D. What is a Drop In VI?

Drop In VI can operate on the calling VI in which the Drop In VI is placed

- Does not need to know any details about the calling VI
- Flexible enough to function with little to no input



ni.com/training

Advantages of Drop In VIs

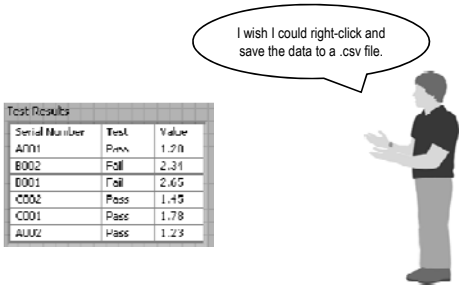
- Achieve complex operations with little or no wiring
- Modularize and abstract the complexity
- Foster reuse
- Enable plug & play code (or drop & run)

The scope of the code does not depend on predefined limitations.

It adapts to the code it's in."

NATIONAL INSTRUMENTS | ni.com/training

Drop In VIs – Wouldn't it be nice if...



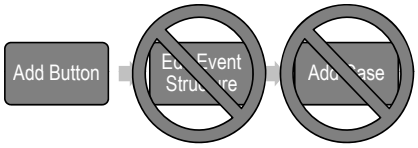
NATIONAL INSTRUMENTS | ni.com/training

Wouldn't it be nice if...

NATIONAL INSTRUMENTS | ni.com/training

How do we normally code?

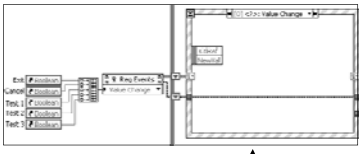
For every new feature we…



What if we could eliminate steps?

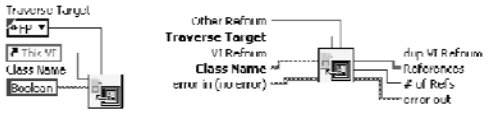
NATIONAL INSTRUMENTS | ni.com/training

What if we used control references, thus eliminating the step of editing the event structure?



NATIONAL INSTRUMENTS | ni.com/training

A Better Solution...



Use the Traverse For GObjects VI from the VI Scripting palette

Drop In VI Example

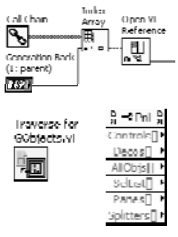
Right-click Drop In VI

`<Exercises>\…\Demonstrations\Drop In - Right Click\Drop In – Right Click.lvproj`

DEMONSTRATION

Fundamentals of the Drop In VI

- Get a reference to the high-level object of interest
  - Front Panel Ref
  - Application Ref
  - Path
- Scan results to get items
  - Utilize unused/hidden fields to flag items
- Perform extraction or operation



No Passing of Data Needed

Drop In VI – Two Types

- Single-shot Drop In VI
  - Performs functionality on calling VI and finishes execution
  - Examples
    - Resize the front panel of calling VI to the largest decoration
- Continuous Drop In VI
  - Performs functionality on calling VI continuously
  - Examples
    - Right-click to show description of any Boolean control
    - Allow user to move the VI by clicking and dragging the front panel

Design Challenge – How do you terminate a Continuous Drop In?

Termination of a continuous Drop In VI can be implemented in many ways. The method will be driven by the style adopted by the organization.

- User Events
- Named Queues
- The "Exit" control
- Other intertask communication methods

More Drop In VI Examples

- Single-shot Drop In VI
- Continuous Drop In VI

`<Exercises>\…\Demonstrations\Drop In – Single-shot and Continuous\Drop In – Single-shot and Continuous.lvproj`

DEMONSTRATION

Exercise 7-1: Using Drop Ins

To create a drop-in tool that logs all Boolean value change events to a file.

GOAL

Exercise 7-1: Using Drop Ins

- Keep a running list of features you would like to incorporate in each application.

DISCUSSION

E. Other

Lesson 8
Error Handling

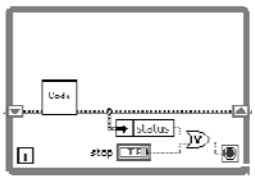
TOPICS

A. Basic Error Handling Strategies
B. Comprehensive Error Handling Strategies
C. LabVIEW Implementation Example – SEH

A. Basic Error Handling Strategies

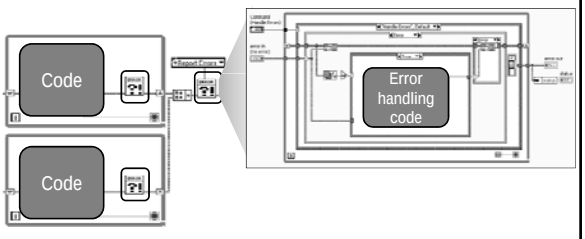
Simple error handling

- Code called in a central location to respond to errors
- Does not respond to specific error codes, but can respond to types of errors differently
- Responds to errors by shutting down the loop
- Ignores warnings
- Sufficient for simple examples



Basic Error Handling Strategies – General Error Handler Module

Create an general Error Handler module to place in each loop



General Error Handler

- What are the strong points of this architecture?
- What could be changed?
- What information could be added to a general error handling system?
- How should errors related to code issues be distinguished from errors due to product failure?

DEMONSTRATION

B. Comprehensive Error Handling Strategies

- Critical applications require more complete error handling than ordinary LabVIEW code
- Comprehensive error handling usually involves the following:
  - Performance tradeoffs
  - Increased development time

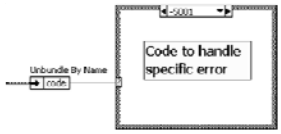
Types of Error Handling

- Specific Error Handling
- Error Classification and Description
- Error Communication
- Central Error Handling
- Error Reporting
- Error Logging
- Throwing Errors

Specific Error Handling

Specific Error Handling–code called in specific locations to respond to specific error codes with an action.

Possible actions
- Ignore
- Retry
- Correct



NATIONAL INSTRUMENTS | ni.com/training

NATIONAL INSTRUMENTS | ni.com/training

Specific Error Handling
Analyzing a Section of Code

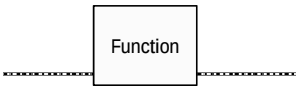
- What could go wrong?
- Can I do anything about it (including logging/reporting)?
- Does responding to it require central error handling code (specific versus central error handling)?
- What is the effect on subsequent code?
- What should be the effect on subsequent code?

NATIONAL INSTRUMENTS | ni.com/training

Specific Error Handling – Guidelines

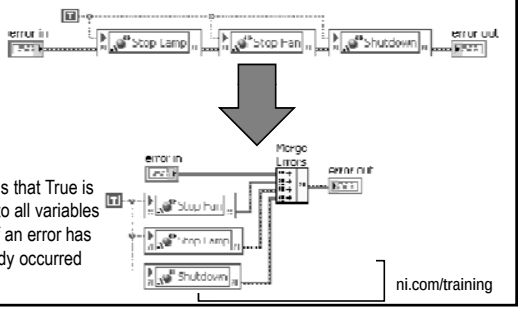
Be aware of how errors affect subsequent code!

- Not all code behaves the same in the event of an error
  - Most functions do not execute when error wire contains error
  - Some functions do execute when error wire contains error
- Better to specifically decide which code to execute using case structures



Specific Error Handling – Avoid allowing errors to affect subsequent code

Example – Shutting down your application



Ensures that True is written to all variables even if an error has already occurred

Error Classification and Description

Organize errors into classifications

Examples

- Warning
- Critical Error
- User Error
- Communications Error

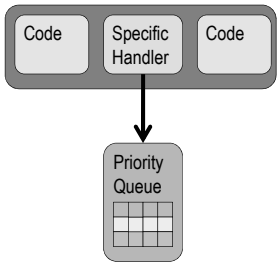
Record additional information about errors

Examples

- Timestamp
- Number of Occurrences
- Call chain
- Data values

Error Communication

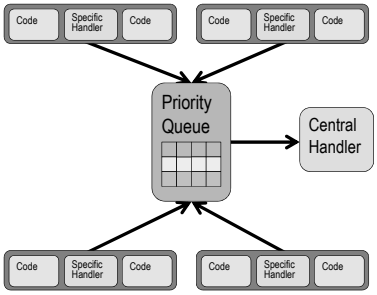
- Transfers an errors from their origin to a central location
- Desirable features:
  - Priority
  - Filtering (count occurrences)



Central Error Handling

- Responds to classes of errors rather than specific codes
- Takes asynchronous or system wide actions
- Example Actions
  - Logs errors to file
  - Display prompts or sends messages to operator
  - Initiate system shutdown/reboot
  - Apply Safe-state to outputs

Central Error Handling



Error Reporting

- Display a message on a user interface
- Desirable features:
  - Understandable messages
  - Localization
  - User response
  - Adaption to error classification
- Can use Simple Error Handler and General Error Handler VIs

Error Logging

- Create a record of errors for future reference and/or debugging
- Desirable features:
  - Human readable
  - High-performance
  - Limited space or lifespan

Throwing Errors

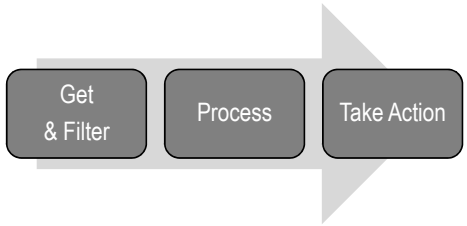
- Generate an error based on something that happened during execution
- Often uses custom error codes and messages

Which Errors To Handle?

- Only handle errors you can do something about
  - Ignore errors in error handling and shutdown
- Focus on mission-critical sections of code
- Document any assumptions or unhandled errors
- Use code reviews to achieve consensus on where to focus effort
- Keep future changes in mind

Developing an Error Handling System

After you finalize your error handling system requirements, you start designing the architecture, which may be composed of several modules



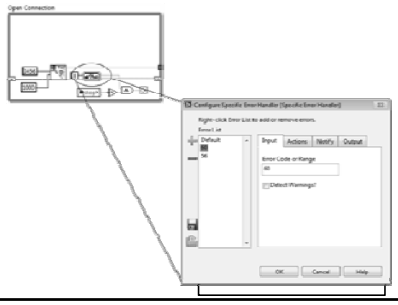
C. LabVIEW Implementation Example – SEH

Structured Error Handler (SEH) Reference Library

- Provides tools for comprehensively handling errors
  - Configurable Express VI to handle specific errors
  - Communication mechanism for transmitting errors
  - Template for a central error handler
  - Various supporting VIs and utilities
- Can download from `ni.com`

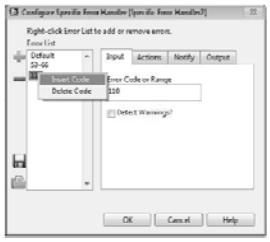
Structured Error Handler (SEH) Implementation

- Specific Error Handler Express VI designed to handle common responses to an error



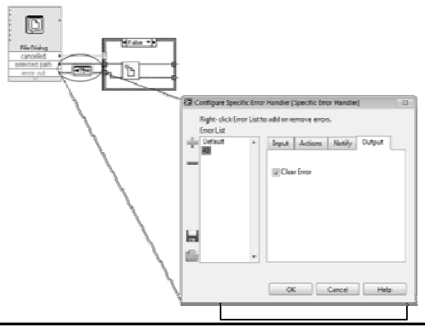
Structured Error Handler (SEH) Implementation

- Specific error handling – Define error code(s) to handle



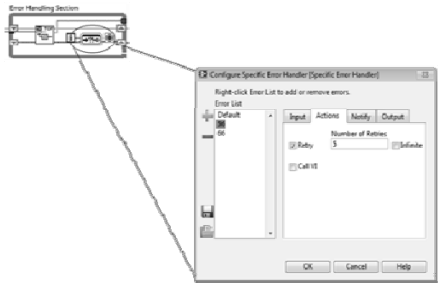
Structured Error Handler (SEH) Implementation

- Specific error handling – Ignore



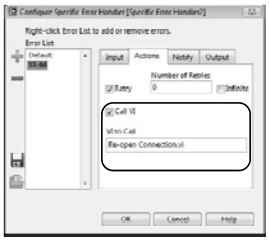
Structured Error Handler (SEH) Implementation

- Specific error handling – Retry



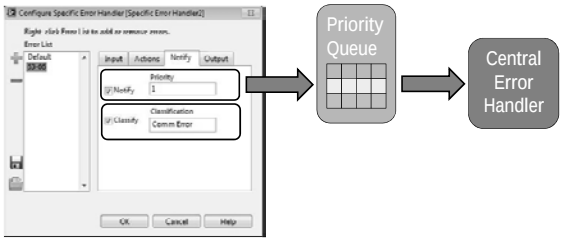
Structured Error Handler (SEH) Implementation

- Specific error handling – Call VI to correct or handle the error



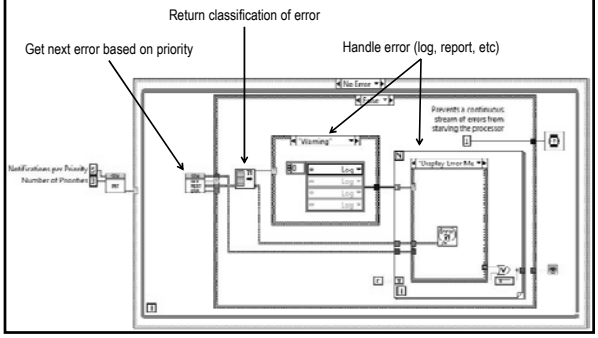
Structured Error Handler (SEH) Implementation

- Classify the error
- Transmit the error to a Central Error Handler



Structured Error Handler (SEH) Implementation

- Central error handler



LabVIEW Implementation Example – Error Constant

- Can use Error constant to throw and generate specific warnings and errors
- Included in LabVIEW 2012
- Downloadable for LabVIEW 2011

Exercise 8-1: Designing an Error Handling System

To design and prototype an error handling system that you can use in your daily LabVIEW development.

GOAL

Exercise 8-1: Designing an Error Handling System

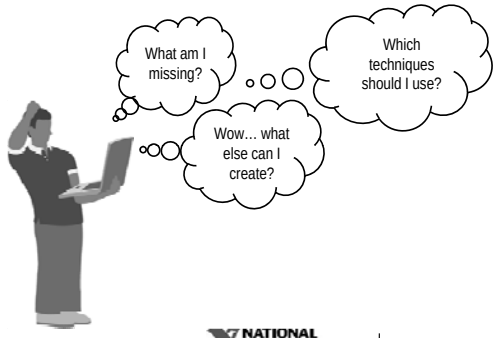
DISCUSSION

Lesson 9
Additional Resources

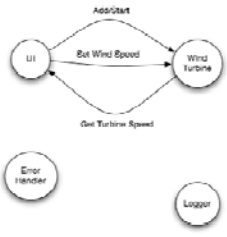
TOPICS

A. Now What?

A. Now What?



Wind Farm



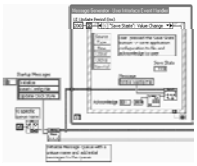
Return to your design from Lesson 1.

- What do you still like about your design?
- What would you do differently now?
- Which advanced design pattern is still confusing?
- Which design pattern will you use when you return to the workplace?
- Which design pattern will you not use?
- Which design pattern should have been included in this course?

DISCUSSION

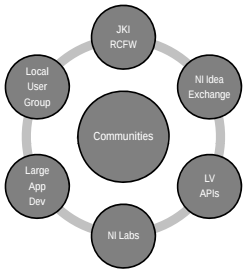
Where Can You Go for More Code and Ideas?

Reference Designs Group on NI Community
- Asynchronous Message Communication (AMC)
- Structured Error Handler (SEH)
- Multi-Process Engine (MPE)
- Extensible Session Framework (ESF)
- Generic Configuration Editor (xCE)
- And more…



DEMONSTRATION

Where Can You Go for More Code and Ideas?



NI Developer Zone Community

DEMONSTRATION

Where Can You Go for More Code and Ideas?

`www.lavag.org`
- More Than A Message Board
- Community of avid LabVIEW programmers
- Place to learn about new technologies and advanced design patterns
- Place to shape the future of LabVIEW

In Conclusion...

Do not be a mason, stacking blocks of code.

Do not be a plumber, laying pipes for data to flow.

Do not be an electrician, placing switches in place.

Rather, be an architect, using tools that allow you to design with creativity, flow, and elegance.

Appendix A
Additional XControl Programming Techniques

TOPICS

A. Refresh Section
B. Shortcut Menus
C. Resizing the XControl
D. Defining Custom Events

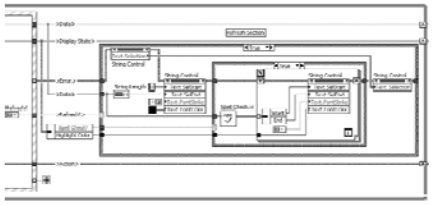
A. Refresh Section

- XControls often have complex code to update the appearance of the control
  - Redraw pictures
  - Process data before display
  - Others
- Code to update the appearance of the XControl may need to be called from multiple events
  - Display State Change
  - Data Change
  - Resize event
  - Front panel events
  - Others

Refresh Section (continued)

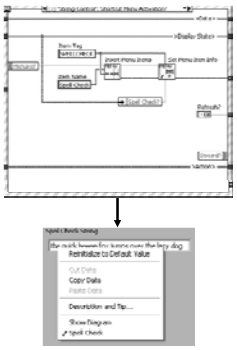
Create a refresh case after the main Event structure

- Avoids repeating code
- Prevents the use of excessive local variables and/or control references
- Allows each event to trigger a refresh with a Boolean value



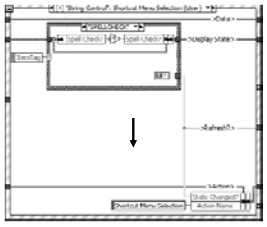
B. Shortcut Menus

- Define right-click commands for XControls like any other control
- Different controls on the Facade can each have their own right-click menus
- Handle the Shortcut Menu Activation? event and alter the menu
- Use to provide quick access to properties or display a configuration dialog box



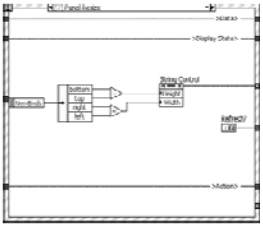
Shortcut Menus (continued)

- Handle the Shortcut Menu Selection event to process shortcut commands
- Shortcut commands typically alter the state
- Remember that setting the state changed action does not trigger the Display State Change event



C. Resizing the XControl

- The user can resize the container of an XControl instance
- Two techniques for handling XControl resizing:
  - Set the Scale Object with Pane option for one or more controls
  - Handle the Panel Resize event

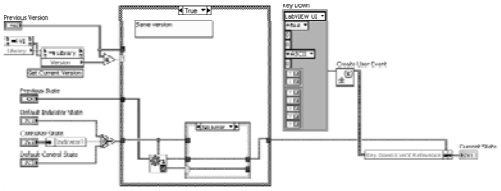


D. Defining Custom Events

- Each XControl exposes standard events to its parent VI
  - You cannot directly alter or override the events
  - The events apply to the XControl as a whole
- XControls often need the ability to notify the parent VI when an event occurs in the XControl
  - Propagate events from the Facade VI controls
  - Send commands to the parent VI to initiate actions based upon XControl interactions
- One solution is to define user events from the XControl and make them available to the parent VI

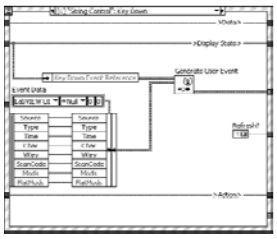
Defining Custom Events – Init Ability

1. Define the event data for your custom or propagated events
2. Create the events in the Init ability
3. Store the event references in the Display State
4. Destroy events in an Uninit ability



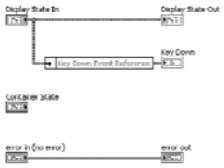
Defining Custom Events – Generating Events

Generate events from any XControl VI with access to the display state



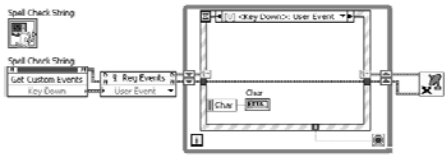
Defining Custom Events – Accessing Events

Provide properties or methods to access event references from the state



Defining Custom Events – Using Custom Events

- Access event references from the parent VI
- Register events as user events
- Handle events normally



B

Additional Information and Resources

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

National Instruments Technical Support Options

Log in to your National Instruments `ni.com` User Profile to get personalized access to your services. Visit the following sections of `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 eLearning training modules at `ni.com/elearning`. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

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

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

Other National Instruments Training Courses

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

National Instruments Certification

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