

LabVIEW™ Connectivity Exercises

Course Software Version 2010
May 2011 Edition
Part Number 325628A-01

Copyright

© 2004–2011 National Instruments Corporation. All rights reserved.

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

For components used in USI (Xerces C++, ICU, HDF5, b64, Stingray, and STLport), the following copyright stipulations apply. For a listing of the conditions and disclaimers, refer to either the `USICopyrights.chm` or the *Copyrights* topic in your software.

Xerces C++. This product includes software that was developed by the Apache Software Foundation (<http://www.apache.org/>). Copyright 1999 The Apache Software Foundation. All rights reserved.

ICU. Copyright 1995–2009 International Business Machines Corporation and others. All rights reserved.

HDF5. NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998, 1999, 2000, 2001, 2003 by the Board of Trustees of the University of Illinois. All rights reserved.

b64. Copyright © 2004–2006, Matthew Wilson and Synesis Software. All Rights Reserved.

Stingray. This software includes Stingray software developed by the Rogue Wave Software division of Quovadx, Inc.
Copyright 1995–2006, Quovadx, Inc. All Rights Reserved.

STLport. Copyright 1999–2003 Boris Fomitchev

Trademarks

LabVIEW, National Instruments, NI, ni.com, the National Instruments corporate logo, and the Eagle logo are trademarks of National Instruments Corporation. Refer to the *Trademark Information* at ni.com/trademarks for other National Instruments trademarks.

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

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

Patents

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

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office 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

Lesson 1

Calling Shared Libraries in LabVIEW Exercises

Exercise 1-1	Computer Name.....	1-1
--------------	--------------------	-----

Lesson 2

VI Server Exercises

Exercise 2-1	VI Server Options.....	2-1
Exercise 2-2	VI Statistics	2-3
Exercise 2-3	Remote Run VI.....	2-11
Exercise 2-4	Dynamically Calling VIs.....	2-15

Lesson 3

Using .Net and ActiveX Objects in LabVIEW Exercises

Exercise 3-1	Font Dialog.....	3-1
Exercise 3-2	Word Processor	3-8
Exercise 3-3	Auto Save	3-24
Exercise 3-4	Browse to URL and Display VI Statistics Report.....	3-32

Lesson 4

Connecting to Databases Exercises

Exercise 4-1	Viewing a Database.....	4-1
Exercise 4-2	Connect to a Theatre Database Using LabVIEW.....	4-3
Exercise 4-3	Select Data from a Table	4-6
Exercise 4-4	Insert New Record.....	4-13
Exercise 4-5	SQL Query.....	4-18

Lesson 5

TCP/IP and UDP Exercises

Exercise 5-1	Simple Data Client VI and Simple Data Server VI.....	5-1
Exercise 5-2	TCP Signal Data Transfer	5-6

Lesson 6

Web Services Exercises

Exercise 6-1	Create a LabVIEW Web Service to Add Two Numbers.....	6-1
Exercise 6-2	Accept POST Data from an HTML Form.....	6-9
Exercise 6-3	Generate Image with Web Method.....	6-16
Exercise 6-4	Create an HTTP Client in LabVIEW	6-21

Not for Distribution

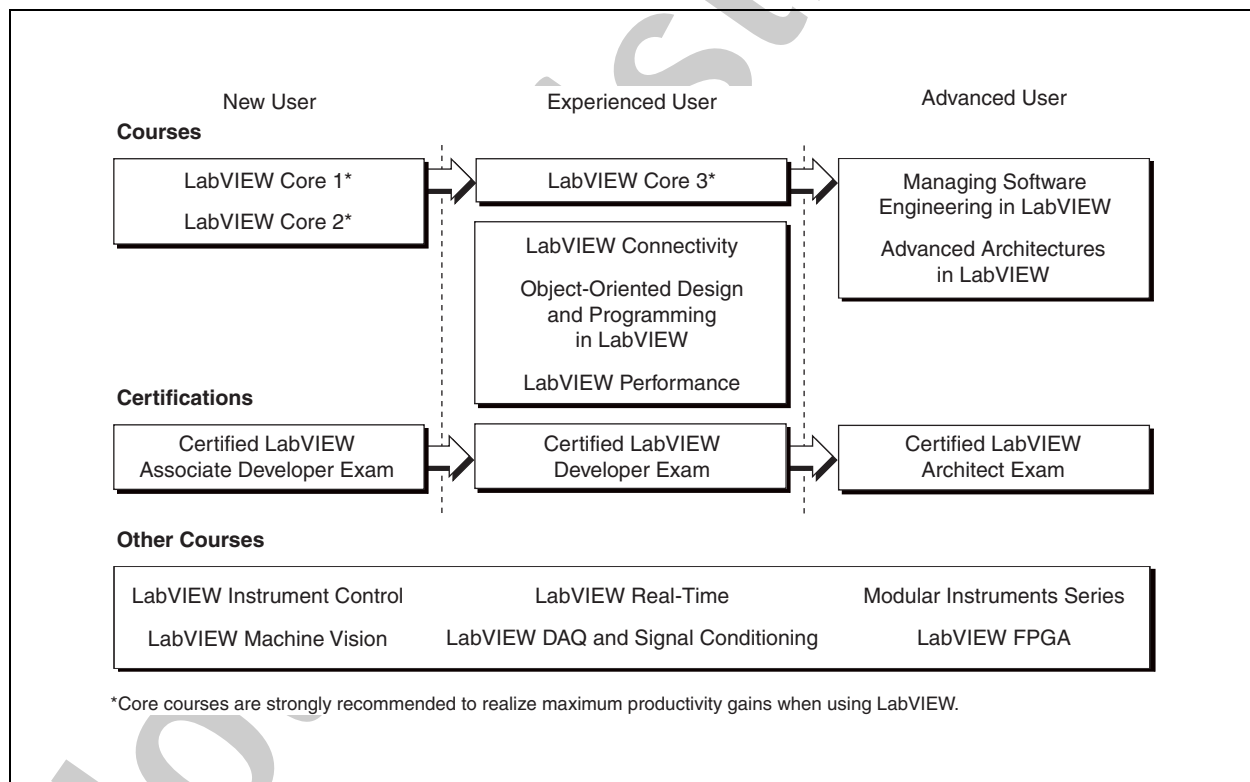
Student Guide

Thank you for purchasing the *LabVIEW Connectivity* course kit. This kit contains the materials used in the two-day, hands-on *LabVIEW Connectivity* 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 *LabVIEW Connectivity* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for exams to become an NI Certified LabVIEW Developer and NI Certified LabVIEW Architect. The following illustration shows the courses that are part of the LabVIEW training series. Refer to ni.com/training for more information about NI Certification.



B. Course Description

The *LabVIEW Connectivity* course teaches you how to use advanced connectivity in VIs. This manual assumes you are familiar with Windows, that you have experience writing algorithms in the form of flowcharts or block diagrams, and that you have taken the *LabVIEW Core 1* and *LabVIEW Core 2* courses or you are familiar with all the concepts contained therein. This course also assumes that you have one year or more of LabVIEW development experience.

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

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

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

- A set of exercises to reinforce topics
- Self-study and challenge exercise sections or additional exercises



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

C. What You Need to Get Started

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

- ☐ Windows XP or later installed on your computer; this course is optimized for Windows XP
- ☐ LabVIEW Professional Development System 2010 or later
- ☐ Microsoft Excel
- ☐ *LabVIEW Connectivity* course CD, containing the following folders:

Directory	Description
Exercises	Contains all the VIs and support files needed to complete the exercises in this course
Solutions	Contains completed versions of the VIs you build in the exercises for this course

D. Installing the Course Software

Complete the following steps to install the course software.

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

Exercise files are located in the <Exercises>\LabVIEW Connectivity folder.



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

Repairing or Removing Course Material

You can repair or remove the course material using the **Add or Remove Programs** feature on the Windows **Control Panel**. Repair the course manual to overwrite existing course material with the original, unedited versions of the files. Remove the course material if you no longer need the files on your computer.

E. Course Goals

This course presents the following topics:

- Networking technologies
 - External procedure call model
 - Broadcast model
 - Client/server model
 - Publish/subscribe model
- Implementing the external procedure call model
 - Calling shared libraries from LabVIEW
 - Programmatically controlling VIs using the VI Server
 - Using the VI Server functions to programmatically load and operate VIs and LabVIEW itself
 - Using ActiveX objects in LabVIEW
 - Using LabVIEW as an ActiveX client
 - Using LabVIEW as an ActiveX server
 - ActiveX events




- Using .NET Objects in LabVIEW
 - Using LabVIEW as a .NET client
 - .NET events
- Implementing the broadcast model
 - Using the UDP VI and functions to create a UDP multicast session
- Implementing the client/server model
 - Using the TCP/IP VI and functions to communicate with other applications locally and over a network
- Using LabVIEW Web services and HTTP Client VIs to create and deploy Web services

This course does not present any of the following topics:

- Basic principles of LabVIEW covered in the *LabVIEW Core 1* and *LabVIEW Core 2* courses
- Every built-in VI, function, or object; refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course
- Developing a complete VI for any student in the class; refer to the NI Example Finder, available by selecting **Help»Find Examples**, for example VIs you can use and incorporate into VIs you create

F. Course Conventions

The following conventions are used in this course manual:

- | | |
|---|--|
| <> | Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, AO <3..0>. |
| [] | Square brackets enclose optional items—for example, [response]. |
| » | 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. |
|  | This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash. |

bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
<i>italic</i>	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.
monospace bold	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
<i>monospace italic</i>	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

Not for Distribution

Calling Shared Libraries in LabVIEW Exercises

Exercise 1-1 Computer Name

Goal

Call a DLL function from the Windows API.

Scenario

Programmatically determining the Windows computer name is necessary in a number of situations. For example, if you have several computers logging data to a central server, you should identify the computer name for each set of data in order to track the data. The computer name is also helpful for storing or retrieving data over a Windows File Share and is a useful piece of information to include when generating reports.

There is not a native function within LabVIEW to retrieve the computer name. However, the Windows operating system provides an API, in the form of DLLs, which allows you to interface with the operating system. One common use of this API is retrieving useful information about the operating system or computer, such as the computer name.

Call a Windows API DLL function to determine the name of the computer. You should also handle any errors returned by the function in an appropriate manner.



Note The Windows computer name is different from the network address of a computer. You can determine a computer's network address in LabVIEW by using the String to IP and IP to String functions located on the TCP/IP palette.

Design

Inputs and Outputs

Table 1-1. Computer Name Inputs and Outputs

Type	Name	Properties	Default Values
Numeric Control	Buffer Size	32-bit Unsigned Integer	256
String Indicator	Computer Name	—	Empty

Program Structure

When you call functions from a DLL, the DLL often defines a method for reporting errors. The Windows API returns a numeric value from each function call indicating whether the function was executed correctly. In the event of an incorrect execution, you can call additional Windows API functions to identify the error code and convert the code to an understandable string. To correctly identify the error, the error handling functions must be called immediately after the original function call, because the error information will be lost if any other Windows API calls are made from the same program. This error handling mechanism works differently than the normal error handling mechanism in LabVIEW, where each subVI reports its own errors and errors can be chained together and handled at the end of a program or section.

Get Computer Name

The Windows API provides a function called `GetComputerName` in `kernel32.dll`. You must allocate a string buffer large enough to store the string returned by the function, fortunately, the Call Library Function Node in LabVIEW Version 8.20 or later can automatically allocate the buffer based on a size parameter passed to the function. If the buffer is too small to store the computer name, or if another error occurs, this function returns a value of zero. Use the return value from this function as well as the error out cluster from the Call Library Function Node to control the transition logic of the state machine.

Handle Errors

This program has the potential to generate normal LabVIEW errors, such as a missing DLL file in a Call Library Function Node, as well as Windows API errors, such as an insufficient buffer size for the `GetComputerName` function to store the computer name. The handle errors case should handle both types of errors.

Use the General Error Handler instead of the Simple Error Handler to allow for custom messages generated by the Windows API error reporting calls.

In order to provide a meaningful error message for Windows API calls, you must perform two steps. First, you must call the Windows API function `GetLastError`. This function returns a numeric value that represents any error encountered during the last function call, in this case, `GetComputerName`. In order to translate the numeric code into a meaningful message, you must call the `FormatMessage` Windows API function. You can use this function many different ways. Control the behavior of the function by setting one or more flags in a parameter called `dwFlags`. In order to return a message for a system error, set the `FORMAT_MESSAGE_FROM_SYSTEM` flag. Also set the `FORMAT_MESSAGE_IGNORE_INSERTS` flag because you do not need to include any additional parameters in your message. Setting this parameter also allows you to ignore the **arguments** parameter of the function, thereby simplifying the function call. For more information on setting flags, refer to the *Background: Windows API Reference* section.



Tip All of the functions called in this exercise are in `kernel32.dll`. However, Windows API functions are found in many other DLLs. Refer to the *Windows API Reference* to determine in which DLL a given function is located.

Additional Information

The following sections describe some issues particular to Windows API calls that you must address in order to implement a solution.

Background: Windows API Reference

You can find the Windows API definitions for each of the functions used in this exercise in the <Exercises>\LabVIEW Connectivity\Computer Name directory. Open each of the PDF files in this directory and browse the reference material in them. As you proceed through the exercise, refer to the references periodically to identify the meaning of each parameter you pass to the functions.



Note The function information in this manual is from the *Windows API Reference* in the *MSDN library*. You can access the full *Windows API Reference* at:
<http://msdn.microsoft.com/en-us/library/Aa383749>.

Bitmasked Flags

Some functions in the API contain a flags parameter that allows you to enable one or more options by masking the bits in an integer. The `dwFlags` parameter of the `FormatMessage` function is an example of this technique.

In order to set a flag in LabVIEW, pass an integer constant with the appropriate size and value to the Call Library Function Node. Remember that you can change the way a numeric constant is represented to make it easier to enter the flag value. For example, the specification for the `FormatMessage` function gives the flags in hexadecimal format. You can set the format of your numeric constant to hexadecimal and then enter the value directly.

If you need to set multiple flags in a flags parameter, you can use a Bitwise Or function to combine multiple flags. In LabVIEW, the Or function is a polymorphic function which automatically becomes a Bitwise Or if you wire two integers to it.

Data Types

Windows API and other DLL function specifications often refer to many data types which have different names in LabVIEW. Table 1-2 lists LabVIEW types for some of the Windows data types used in this exercise. For a more complete list, the Call DLL example in the NI Example Finder provides a complete data type conversion list, as well as examples for each data type. The Windows Data Type reference is also a useful reference. It can be found at: <http://msdn.microsoft.com/en-us/library/Aa383751>.

Table 1-2. Data Type Conversion

Windows Data Type	LabVIEW Data Type
LPTSTR	String (Pass as C String Pointer)
DWORD	U32
LPDWORD	U32 (Pass by pointer)
BOOL	I32
va_list*	Varies (Use Adapt to Type)
LPCVOID	Varies (Use Adapt to Type)

String Types

Many Windows API functions support two methods for representing strings. The first method is ASCII, in which each character in a string is represented by a single byte. Traditional ASCII has a 128 character set, which contains all of the upper and lowercase letters, as well as other common symbols and a set of control characters. LabVIEW typically uses ASCII to represent strings.

The second method uses “wide” strings, which is another term for Unicode string representation (UTF-16 encoding). Unicode requires two bytes for each character, and allows for a much larger character set than traditional ASCII. Unicode is not natively supported in LabVIEW, but it is possible to use Unicode through add on libraries or calls to external functions.

Windows API functions which deal with strings often have two versions present in the DLL, marked with an A for ASCII and a W for wide. For example, there are two `GetComputerName` functions in `kernel32.dll`, `GetComputerNameA` and `GetComputerNameW`. In most cases, you should use the “A” version of the function in LabVIEW.

Implementation

1. Create the front panel shown in Figure 1-1.

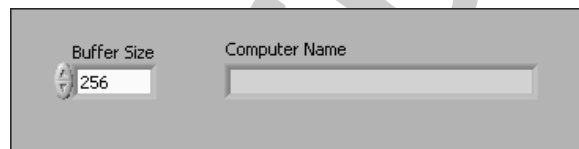


Figure 1-1. Computer Name Front Panel

- ☐ Create a blank VI and save the VI as `Computer Name.vi` in the `<Exercises>\LabVIEW Connectivity\Computer Name` directory.
 - ☐ Create the **Buffer Size** control as described in Table 1-1.
 - ☐ Create the **Computer Name** indicator as described in Table 1-1.
2. Call the `GetComputerName` function with the parameters shown in Table 1-3.

Table 1-3. `GetComputerName` Parameters

Parameter Name	Type	Format/Data type	Other
return type	Numeric	Signed 32-bit Integer	
lpBuffer	String	C String Pointer	Select lpnSize for Minimum size
lpnSize	Numeric	Unsigned 32-bit Integer	Select Pointer to Value for Pass



- ☐ Open the `GetComputerName` function reference from `<Exercises>\LabVIEW Connectivity\Computer Name\getcomputername.pdf` and identify the prototype and parameters for the function.
- ☐ Place a Call Library Function Node.
- ☐ Double-click the Call Library Function Node to open the **Call Library Function** dialog box.
- ☐ Click the **Function** tab and configure the settings to match Figure 1-2.
 - Click the **Browse** button and navigate to `\Windows\System32\kernel32.dll` or enter `kernel32.dll`.
 - Select **GetComputerNameA** from the **Function name** pull-down menu.
 - Select **Run in any thread** from the **Thread** section.



Note Because the Windows API functions are reentrant (multi-threaded), calling `GetComputerNameA` in UI thread functions correctly, except the error is not stored in the proper memory location for `GetLastError` to access it.

- Select **stdcall (WINAPI)** from the **Calling convention** section.

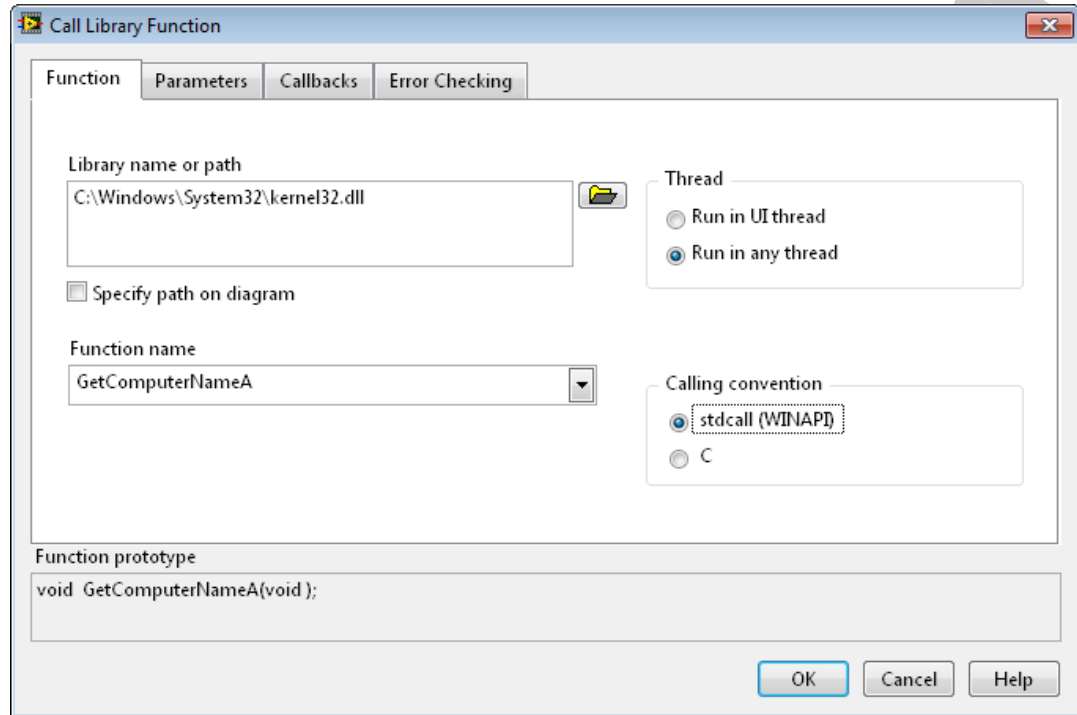


Figure 1-2. GetComputerNameA Call Library Function

- ❑ Click the **Parameters** tab.
 - Ensure the **return type** parameter is selected.
 - Select **Numeric** from the **Type** pull-down menu in the **Current parameter** section.
 - Verify the **Data type** pull-down menu is set to **Signed 32-bit Integer**.
- Click the **+** button to add a parameter after the return type parameter.
- Enter `lpBuffer` in the **Name** text box.
- Select **String** from the **Type** pull-down menu.
- Verify the **String format** pull-down menu is set to **C String Pointer**.



Note You must declare an additional parameter before setting the minimum size for the `lpBuffer` parameter. Leave **Minimum size** blank.

- Click the + button to add a parameter.
- Enter `lpnSize` in the **Name** text box.
- Select **Numeric** from the **Type** pull-down menu.
- Select **Unsigned 32-bit Integer** from the **Data type** pull-down menu.
- Select **Pointer to Value** from the **Pass** pull-down menu.
- Select the **lpBuffer** parameter from the parameter list.
- Select **lpnSize** from the **Minimum size** pull-down menu.
- Confirm that the function prototype matches the following text.

```
int32_t GetComputerNameA(CStr lpBuffer,
uint32_t *lpnSize);
```

- ☐ Click the **OK** button.

3. Call the `GetLastError` function with the parameters shown in Table 1-4.

Table 1-4. GetLastError Parameters

Parameter Name	Type	Format/Data type	Other
return type	Numeric	Unsigned 32-bit Integer	—

- ☐ Open the `GetLastError` function reference from `<Exercises>\LabVIEW Connectivity\Computer Name\getlasterror.pdf` and identify the prototype and parameters for the function.
- ☐ Place another Call Library Function Node after the call to `GetComputerName`.
- ☐ Double-click the Call Library Function Node to open the **Call Library Function** dialog box.
- ☐ Click the **Function** tab.
 - Click the **Browse** button and navigate to `\Windows\system32\kernel32.dll` or enter `kernel32.dll`.
 - Select **GetLastError** from the **Function Name** pull-down menu.

- Select **Run in any thread** from the **Thread** section.
 - Select **stdcall (WINAPI)** from the **Calling convention** section.
 - ☐ Click the **Parameters** tab.
 - Ensure the return type parameter is selected.
 - Select **Numeric** from the **Type** pull-down menu.
 - Select **Unsigned 32-bit Integer** from the **Type** pull-down menu.
 - Confirm that the function prototype matches the following text.


```
uint32_t GetLastError(void);
```
 - ☐ Click **OK**.
4. Create the block diagram as shown in Figure 1-3 using the following items:

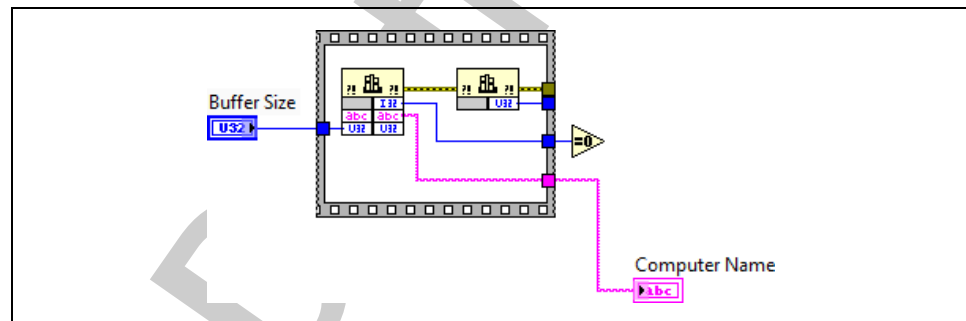


Figure 1-3. Get Computer Name

- ☐ Flat Sequence structure—Place around the two Call Library Function Nodes.



Note The Flat Sequence structure ensures that the VI checks for an error right after calling the `GetComputerNameA` function from the DLL and does so in the same thread.

Because of the way the LabVIEW execution system works, it is possible for something to run between the two DLL nodes. This could cause the `GetLastError` call to return an incorrect result. The Flat Sequence structure with nothing but the two DLL nodes in it reduces the likelihood that this would occur.

- ☐ Equal To 0?
- ☐ **Computer Name** indicator
- ☐ **Buffer Size** control

5. Call the `FormatMessage` function with the parameters shown in Table 1-5.

Table 1-5. `FormatMessage` Parameters

Parameter Name	Type	Format/Data type	Other
return type	Numeric	Unsigned 32-bit Integer	—
dwFlags	Numeric	Unsigned 32-bit Integer	Select Value for Pass
lpSource	Adapt to Type	Pointers to Handles	—
dwMessageId	Numeric	Unsigned 32-bit Integer	Select Value for Pass
dwLanguageId	Numeric	Unsigned 32-bit Integer	Select Value for Pass
lpBuffer	String	C String Pointer	Select nSize for Minimum size
nSize	Numeric	Unsigned 32-bit Integer	Select Value for Pass
Arguments	Adapt to Type	Pointers to Handles	—

- ☐ Open the `FormatMessage` function reference from `<Exercises>\LabVIEW Connectivity\Computer Name\formatmessage.pdf` and identify the prototype and parameters for the function.
- ☐ Place a Call Library Function Node after the Flat Sequence structure.
- ☐ Double-click the Call Library Function Node to open the **Call Library Function** dialog box.
- ☐ Select the **Function** tab.
 - Click the **Browse** button and navigate to `\Windows\System32\kernel32.dll` or enter `kernel32.dll`.
 - Select **FormatMessageA** from the **Function Name** pull-down menu.
 - Select **Run in any thread** from the **Thread** section.
 - Select **stdcall (WINAPI)** from the **Calling convention** section.
- ☐ Click the **Parameters** tab.
 - Ensure the **return type** parameter is selected.

- Select **Numeric** from the **Type** pull-down menu.
- Select **Unsigned 32-bit Integer** from the **Type** pull-down menu.
- Click the + button to add a parameter.
- Enter `dwFlags` in the **Name** text box.
- Select **Numeric** from the **Type** pull-down menu.
- Select **Unsigned 32-bit Integer** from the **Data type** pull-down menu.
- Verify the **Pass** pull-down menu is set to **Value**.
- Click the + button to add a parameter.
- Enter `lpSource` in the **Name** text box.
- Select **Adapt to Type** from the **Type** pull-down menu.
- Select **Pointers to Handles** from the **Data Format** pull-down menu.
- Click the + button to add a parameter.
- Enter `dwMessageId` in the **Name** text box.
- Select **Numeric** from the **Type** pull-down menu.
- Select **Unsigned 32-bit Integer** from the **Data type** pull-down menu.
- Verify the **Pass** pull-down menu is set to **Value**.
- Click the + button to add a parameter.
- Enter `dwLanguageId` in the **Name** text box.
- Select **Numeric** from the **Type** pull-down menu.
- Select **Unsigned 32-bit Integer** from the **Data type** pull-down menu.
- Verify the **Pass** pull-down menu is set to **Value**.
- Click the + button to add a parameter.
- Enter `lpBuffer` in the **Name** text box.

- Select **String** from the **Type** pull-down menu.
- Verify the **String format** pull-down menu is set to **C String Pointer**.



Note You must declare an additional parameter before setting the Minimum Size pull-down menu. Leave Minimum Size set to **<None>** for now.

- Click the **+** button to add a parameter
- Enter `nSize` in the **Name** text box.
- Select **Numeric** from the **Type** pull-down menu.
- Select **Unsigned 32-bit Integer** from the **Data type** pull-down menu.
- Verify the **Pass** pull-down menu is set to **Value**.
- Click the **+** button to add a parameter
- Enter `Arguments` in the **Name** text box.
- Select **Adapt to Type** from the **Type** pull-down menu.
- Select **Pointers to Handles** from the **Data Format** pull-down menu.
- Select the **lpBuffer** parameter from the parameter list.
- Select `nSize` from the **Minimum size** pull-down menu.
- Confirm that the function prototype matches the following text.
- ```
uint32_t FormatMessageA(uint32_t dwFlags, void
*lpSource, uint32_t dwMessageId, uint32_t
dwLanguageId, CStr lpBuffer, uint32_t nSize,
void *Arguments);
```



**Note** The terminals corresponding to the arguments with type **Void** are blank until wired because void parameters accept any type of data.

- ☐ Click the **OK** button.

6. Create the error handling code shown in Figure 1-4 using the following items.

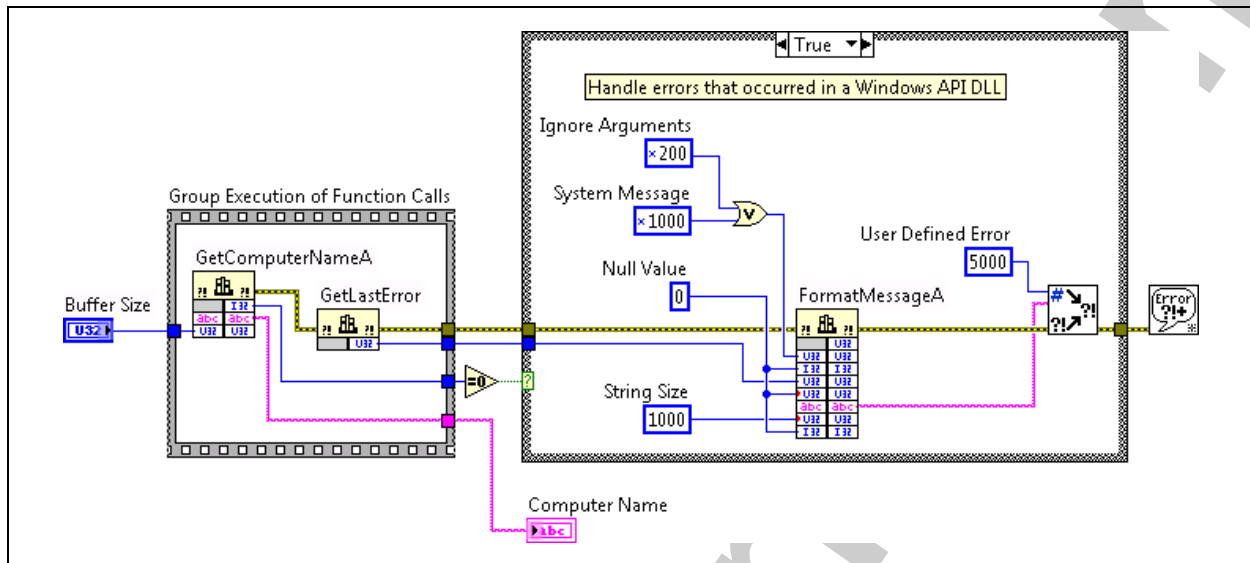


Figure 1-4. Error Handling Code

- ☐ Numeric constant
  - Set the representation to U32.
  - Right-click the numeric constant and select **Visible Items» Radix**.
  - Click the radix and select **Hex**.
  - Set the value of the constant to 200.
  - Label the constant Ignore Arguments.
- ☐ Numeric constant
  - Create a copy of the **Ignore Arguments** constant.
  - Change the label of the new constant to System Message.
  - Set the value of the **System Message** constant to 1000.
- ☐ Or
- ☐ Numeric constant
  - Set the representation to I32.
  - Label the constant Null Value.

- ☐ Numeric constant
  - Set the representation to U32.
  - Label the constant `String Size`.
  - Set the value of the **String Size** constant to 1000.
- ☐ Error Cluster From Error Code VI
  - Right-click the **error code** input terminal of the Error Cluster From Error Code VI and select **Create»Constant**.
  - Set the value of the constant to 5000.
  - Label the constant `User Defined Error`.
- ☐ Place a Case structure around the error handling code.
- ☐ General Error Handler VI.



## Testing

1. Test the VI with an appropriate buffer size.
  - ☐ Run the VI with the default **Buffer Size** (256).
  - ☐ The name of your computer should display in the **Computer Name** indicator.
2. Verify that the correct computer name displays.
  - ☐ Locate **My Computer** on the desktop of your computer or in Windows Explorer.
  - ☐ Right-click **My Computer** and select **Properties** from the shortcut menu.
  - ☐ Select the **Computer Name** tab and verify that the **Full Computer Name** matches the value the VI returns.



**Note** The case of the names does not need to match.

3. Test the error handling in the VI.
  - ☐ Set the **Buffer Size** control to 1 and run the VI.
  - ☐ Verify that the VI displays **The file name is too long.** as an error message.





**Tip** The error message that displays for this VI is the description for the Windows error message `ERROR_BUFFER_OVERFLOW` (System Error 111). Refer to [http://msdn.microsoft.com/library/en-us/debug/base/system\\_error\\_codes.asp](http://msdn.microsoft.com/library/en-us/debug/base/system_error_codes.asp) for more information about system error codes and their descriptions.

### End of Exercise 1-1

## Notes

---

Not for Distribution

## VI Server Exercises

### Exercise 2-1 VI Server Options

#### Goal

Observe and set the VI server configuration options.

#### Description

VI Server presents a potential security risk because other programs and/or computers can use it to call VIs, which can in turn be used to do almost anything with a computer. To protect your computer, VI server contains security options which allow you to select who can use VI server, which VIs users can access, how the VIs can be used, and what communication mechanisms can be used.

This exercise demonstrates the VI Server configuration options and sets them to a configuration you can use to run the remaining exercises in the course. The security configuration set in this exercise is very light. Therefore, if you are completing these exercises on a development machine or any other important machine, increase the security level by only allowing certain machines to access VI Server or return the settings to the default values when you finish the course.

#### Implementation

1. Configure VI Server.
  - ☐ Select **Tools»Options** and select **VI Server** from the **Category** list.
  - ☐ In the Protocols section, select **TCP/IP**. Note the **Port** number.
  - ☐ Verify **ActiveX** is selected.
  - ☐ Verify that all options under **Accessible Server Resources** are selected.
2. Configure Machine Access.
  - ☐ In the Machine Access section, enter \* in the **Machine name/address** field.



**Tip** Entering \* in the **Machine name/address** field opens VI Server Access to all computers. For security reasons, you should not do this on a production computer. Instead, add the machine name or IP address of each computer that needs access to VI server on this computer.

3. Configure Exported VIs.

- ☐ In the **Exported VIs** section, verify \* is entered in the **Exported VIs** list.
- ☐ Click the **OK** button to exit the **Options** dialog box.

**End of Exercise 2-1**

## Exercise 2-2 VI Statistics

### Goal

Open and use VI server references to Application and VI objects to gather information about all open VIs.

### Scenario

The Application VI Server object allows you to retrieve a list of all VIs in memory. Using this list, you can determine information about the VIs in memory. This information is useful for writing tools which track information about the VIs on your system. In this exercise you track VI usage statistics to identify the number of VIs running and the number of VIs in memory at any given time. Because VI Server has the ability to access application instances on remote machines, you can use this program to track the VI usage on any computer which allows you VI Server access.

### Design

#### Inputs and Outputs

**Table 2-1.** VI Statistics Inputs and Outputs

| Type              | Name         | Properties                    | Default Value                     |
|-------------------|--------------|-------------------------------|-----------------------------------|
| String Control    | Machine Name | String                        | Empty (defaults to local machine) |
| Numeric Indicator | VIs Open     | Signed 32-bit Integer         | 0                                 |
| Numeric Indicator | VIs Running  | Signed 32-bit Integer         | 0                                 |
| Table             | VI Report    | Table, Column headers visible | Empty (Column headers only)       |

#### Program Flow

1. Acquire a reference to the LabVIEW Application object by using the Open Application Reference function.
2. Use this reference to access the ExportedVIs property, which gives you a list of each VI in memory.
3. Use a For Loop and the Open VI Reference function to get a reference to each VI in the list.
4. Using the VI reference, access the desired properties, in this case, Name, VIType and Exec.State.

5. Close each VI reference.
6. Gather and display the data.
7. Close the application reference.

## Property Descriptions

Use the following properties in this program:

**ExportedVIs** (Application Object)—This property returns an array of strings, which represent the name of all VIs in memory, if run on the local machine, or a list of all exported VIs in memory, if run on a remote machine. You must use this property instead of the AllVIs property to run the program on a remote machine. Notice that the strings contain only the names of the VIs, and not their paths. However, because the VIs are already guaranteed to be in memory, you only need the VI name to open a VI reference.

**Name** (VI Object)—This property accesses the name of the VI. You could use the names from the ExportedVIs property in place of this property. However, using the property provides a consistent technique for accessing the VI data and also simplifies wiring.

**VIType** (VI Object)—This property returns an enumeration containing the type of the VI. This property is useful because not all VIs are standard, executable VIs. Examples of other types of VIs include global variables, type definitions, and custom controls. In this program, this property provides information for the VI report. Certain VI properties are valid for only some VI types, and therefore, it may be necessary to check the value of this property before accessing it. For example, if the program uses any properties from the Execution group other than Exec.State, you would need to check this property before accessing the properties to ensure that the current VI reference is not a control or a global variable.

**Exec.State** (VI Object)—This property returns an enumeration containing the execution state of the VI. In this program, you increment the number of running VIs if this property is equal to Run top level or Running.

## Implementation

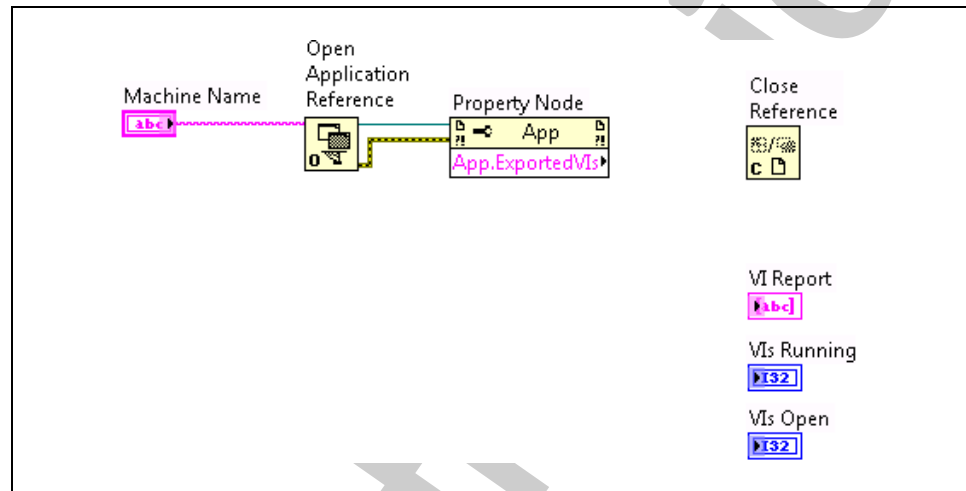
1. Create a blank VI and save the VI as `VI_Statistics.vi` in the `<Exercises>\LabVIEW Connectivity\VI_Statistics` directory.
2. Create the front panel as shown in Figure 2-1.



**Figure 2-1.** VI Statistics Front Panel

- ☐ Create the following items as described in Table 2-1.
  - **Machine Name** control
  - **VIs Open** indicator
  - **VIs Running** indicator
- ☐ Place a Table control on the front panel.
  - Label the table `VI Report`.
  - Right-click the table and select **Change to Indicator** from the shortcut menu.
  - Right-click the table and select **Visible Items»Column Headers** from the shortcut menu.

- Enter VI Name, VI Type, and VI State as the first three column headers.
3. Acquire a reference to the LabVIEW Application object and access the ExportedVIs property. Create the block diagram as shown in Figure 2-2 using the following items.



**Figure 2-2.** Application Properties

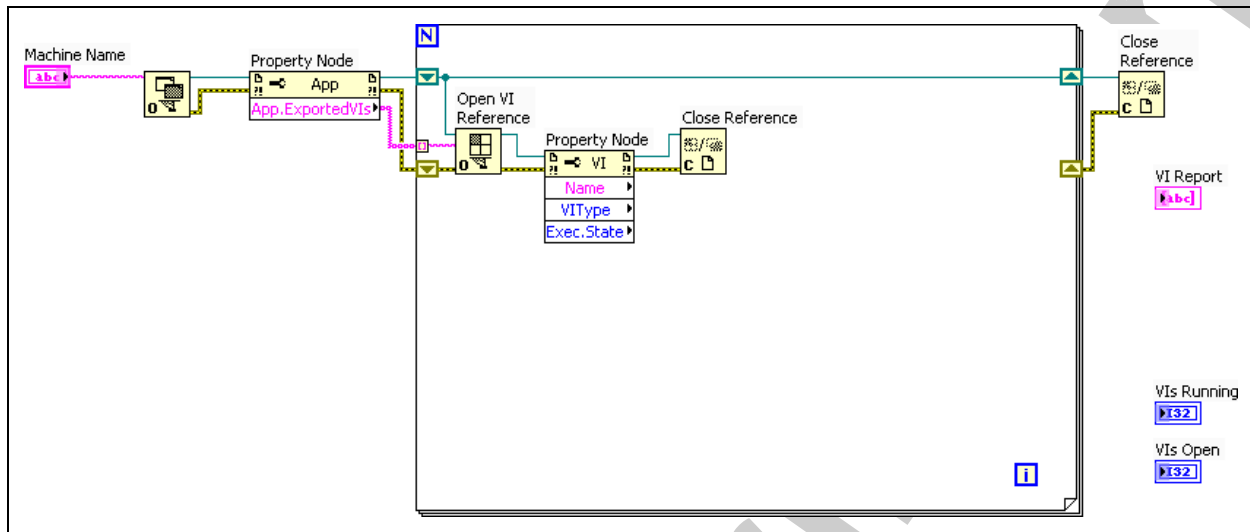
- ☐ Open Application Reference
- ☐ ExportedVIs Property Node—Right-click the **application reference** output of the **Open Application Reference** function and select **Create»Property for Application Class»Application»Exported VIs In Memory** from the shortcut menu.
- ☐ Close Reference



**Tip** Leave space between the Property Node and the Close Reference function so you can insert more code between them in later steps.



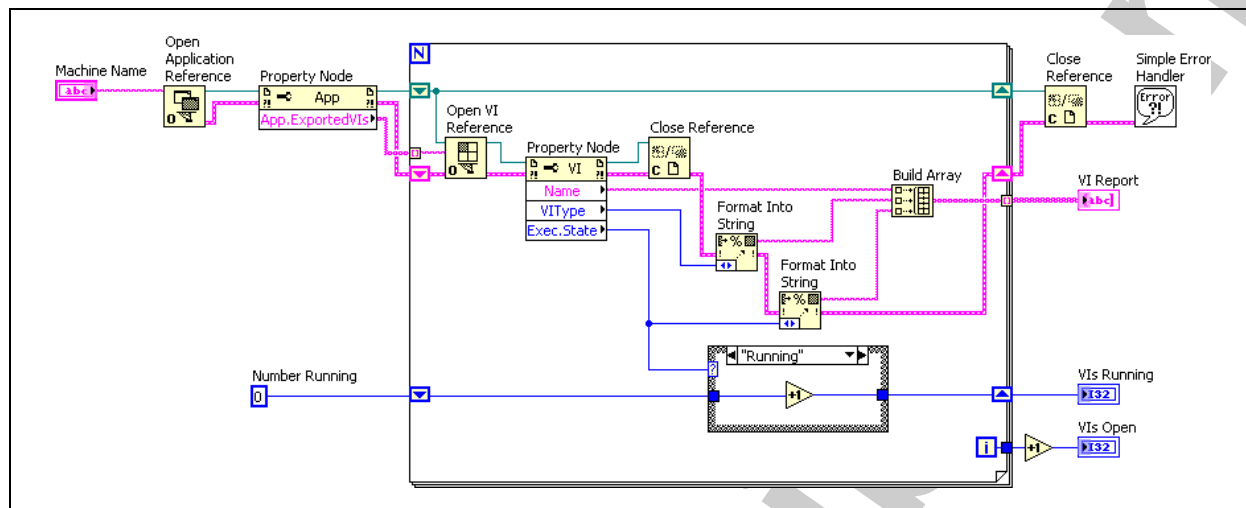
4. Acquire a VI reference to each exported VI. Use the following items to modify the block diagram as shown in Figure 2-3.



### Figure 2-3. VI Properties

- ❑ For Loop
- ❑ Open VI Reference
- ❑ VI Name Property Node
  - Right-click the **vi reference** output of the Open VI Reference function and select **Create»Property for VI Class»VI Name** from the shortcut menu to create the VI Name Property Node.
  - Expand the VI Name Property Node so that three items are available.
  - Click the second item in the Property Node and select **VI Type** from the list.
  - Click the third item in the Property Node and select **Execution»State** from the list.
- ❑ Close Reference
- ❑ Shift Registers—Replace the application reference and error tunnels on the For Loop with Shift Registers.

- Figure 2-4 shows the items and wiring you add in steps 5 and 6. In this step, add the following items to gather and display VI statistics.



**Figure 2-4. VI Statistics Block Diagram**

- ❑ Numeric constant
  - Set the representation to I32.
  - Label the constant `Number Running`.
- ❑ Increment function—Wire the value from the **Number Running** constant through the Increment function to the **VIs Running** indicator. Use shift registers to wire through the For Loop.
- ❑ Case structure
  - Place the Case structure around the Increment function.
  - Wire the output of the `Exec.State` property to the case selector terminal.
  - Right-click the Case structure and select **Add Case After** from the shortcut menu.
  - Verify that the Run top level case of the Case structure is visible. Right-click the Case structure and select **Swap Diagram With Case»Idle** from the shortcut menu.
  - Wire the numeric data through the Idle and Bad cases of the Case structure.
  - Switch to the Run Top Level case, which should have the increment function in it, then right-click the Case structure and

select **Duplicate Case** from the shortcut menu. This creates a Running case which contains an increment function.

The Case structure increments the number in the **VIs Running** indicator if the VI is in the Run Top Level or the Running state.

- ☐ Increment function
    - Place the function to the right of the For Loop.
    - Wire the iteration terminal of the For Loop to the second Increment function through the border of the For Loop. Disable indexing on the tunnel.
  - ☐ Two Format Into String functions—The Format Into String function determines the string representation of an enumerated value.
  - ☐ Build Array
6. In this step, add the following item to handle errors.
    - ☐ Simple Error Handler VI
  7. Save the VI.

## Testing

1. Run the VI.
  - ☐ Close all other open VIs.
  - ☐ Run the VI Statistics VI.
2. Run the VI with multiple VIs in memory.
  - ☐ Open the solution to the Word Processor project in the <Solutions>\LabVIEW Connectivity\Exercise 3-2 directory.



**Note** If you have the LabVIEW Core 3 files installed, you also can use <Solutions>\LabVIEW Core 3\Course Project\Exercise 7-8\TLC Main.vi.

- ☐ Run the VI Statistics VI.
- ☐ Observe the results in the **VI Report**.

## 3. Test the VI on a remote system.

- ☐ Determine the network address of a target computer near you and enter it in the **Machine Name** control.



**Tip** In most cases, you can use the Computer Name, which you found in Exercise 1-1, as a network address. If this name does not work, find the IP address of the computer by using the String to IP and IP to String functions.

- ☐ Verify that the VI Server settings on the target computer are configured as described in Exercise 2-1.
- ☐ Open one or more VIs on the target computer.
- ☐ Run the VI. All exported VIs in memory on the target computer should be displayed.

## Challenge

Add statistical information for the VI priority, VI execution system, and/or state of the front panel to your table. Remember that not all properties are valid for all types of VIs. Use the context help to identify which types of VIs a property applies to and the VIType property to determine which VIs have the property, otherwise you receive an error.

## End of Exercise 2-2

## Exercise 2-3 Remote Run VI

### Goal

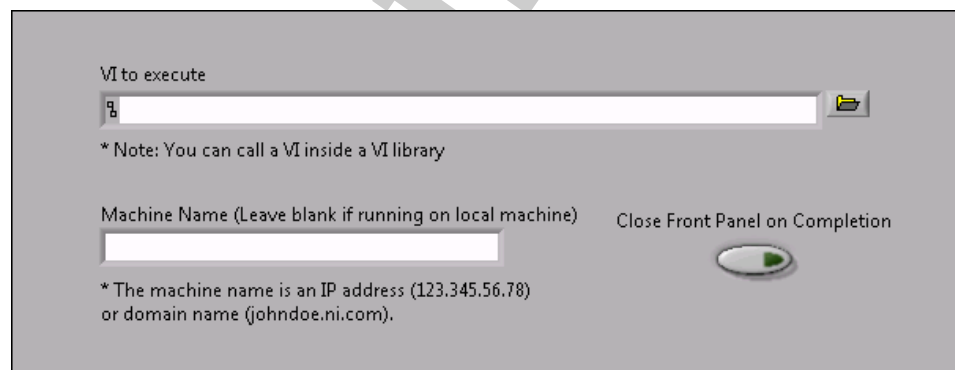
Build a VI that programmatically opens and runs another VI on a remote computer.

### Description

You have seen how the Application refnum runs transparently over a network. In this exercise, use VI Server to run a VI remotely. The techniques in this exercise show how to use VI Server to open and run a VI on a remote machine. VI Server operates the same whether it is on a network or a local machine.

### Implementation

1. Open the Remote Run VI located in the <Exercises>\LabVIEW Connectivity\Remote Run directory. The front panel is built for you.



**Figure 2-5.** Remote Run VI Front Panel

2. In the **VI to execute** control, browse to <Exercises>\LabVIEW Connectivity\Remote Run\Statistics.vi. Right-click the control and select **Data Operations»Make Current Value Default** from the shortcut menu.

3. Build the block diagram shown in Figure 2-6 using the following items.

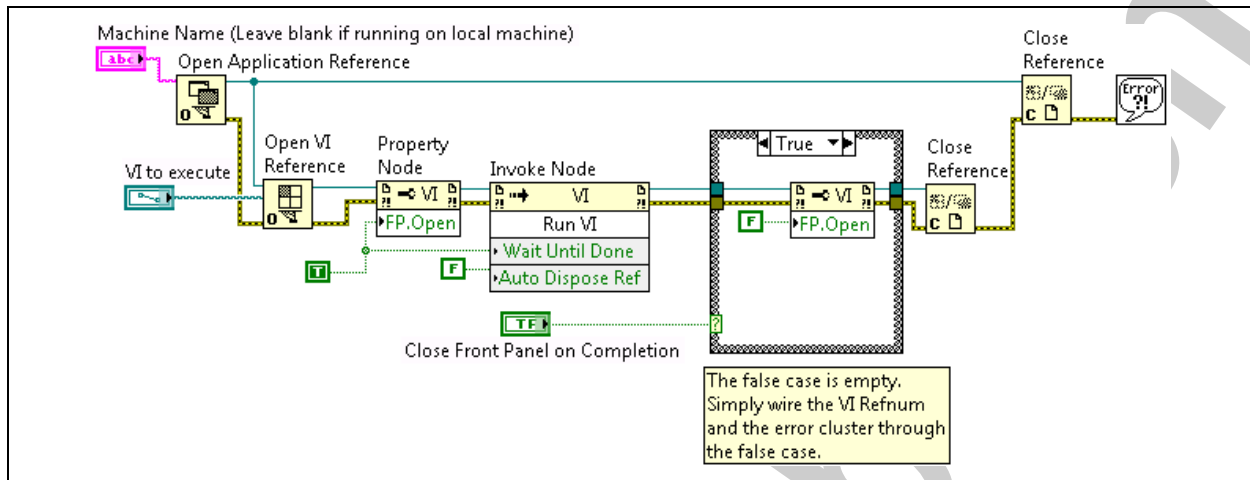


Figure 2-6. Remote Run VI Block Diagram



☐ Open Application Reference



☐ Open VI Reference



☐ Wire the **VI to execute** path control, which determines the VI to execute, to the **vi path** input of the Open VI Reference function.



☐ Two Close References

☐ Property Node

– Wire the **vi reference** output of the Open VI Reference function to the **reference** input of the Property Node.

– Click the **Property** terminal and select **Front Panel Window» Open**.

– Right-click the Property Node and select **Change All to Write** from the shortcut menu.

– Wire a TRUE Boolean constant to the **Front Panel Window Open** property terminal.

– Create a copy of this Property Node.



#### ☐ Invoke Node

- Wire the VI reference from the Property Node to the Invoke Node.
- Click the method terminal and select **Run VI** from the list.
- Wire the TRUE Boolean constant to the Wait Until Done property terminal and a FALSE Boolean constant to the Auto Dispose Ref property terminal.



#### ☐ Simple Error Handler VI

- ☐ Case structure—Use the Case structure to select whether the front panel of the called VI remains open when the VI completes execution.
  - Place the Case structure around the second Property Node.
  - Wire a FALSE Boolean constant to the Front Panel Window Open property of the Property Node. Verify these items are in the True case. This case closes the front panel of the called VI if it is selected.
  - Wire the VI refnum and the error cluster through the False case.
  - Wire the case selector to the **Close Front Panel on Completion** control.

#### 4. Save the VI.

## Testing

Run the VI on the local computer.

This VI opens a reference to the Frequency Response VI located in the <Exercises>\LabVIEW Connectivity\Remote Run directory. The front panel of the VI is opened by accessing the Front Panel Open property of the VI. Then, the Run VI method runs the VI. Because the Wait Until Done property is TRUE, this VI waits for the Frequency Response VI to complete execution. After exiting the Frequency Response VI, the front panel remains open or closes depending on the position of the front panel switch. Finally, the Close Reference function closes the VI reference, freeing the resources.

If time permits, complete the following Optional and Challenge steps, otherwise close the VI.

## Optional

If your computer is connected through TCP/IP to another computer that has LabVIEW and each computer has a unique IP address, you can run the Remote Run VI on one computer and have it call the Frequency Response VI on the other computer.

1. Find a partner and exchange IP addresses. Decide which computer is the server. Complete the following steps on the server computer to set up the VI Server.
  - ☐ Select **Tools»Options** and select **VI Server** from the **Category** list to display the **VI Server** page. Verify that **TCP/IP** is selected and that a port number is entered.
  - ☐ In the **Machine Access** section, enter the IP address of the client computer. Select **Allow Access** and click **Add**.
  - ☐ In the **Exported VIs** section, confirm that a wildcard (\*) is allowed access. This allows the client computer, or any computer allowed access in the **VI Server: Machine Access** section, to access any VIs on your computer. Click the **OK** button.
2. On the client computer, verify the path to the Frequency Response VI on the server computer. Enter the IP address of the server computer in the **Machine Name** control.
3. Run the Remote Run VI on the client computer. Does the VI behave as expected? Repeat steps 1 and 2, but reverse situations with your partner.

## Challenge

Break into groups of three. Write a VI on the first computer that calls the Remote Run VI on the second computer, which then calls the Frequency Response VI on the third computer.

### End of Exercise 2-3



## Exercise 2-4 Dynamically Calling VIs

### Goal

Observe two different methods for calling VIs dynamically and learn the difference between strictly and weakly typed refnums.

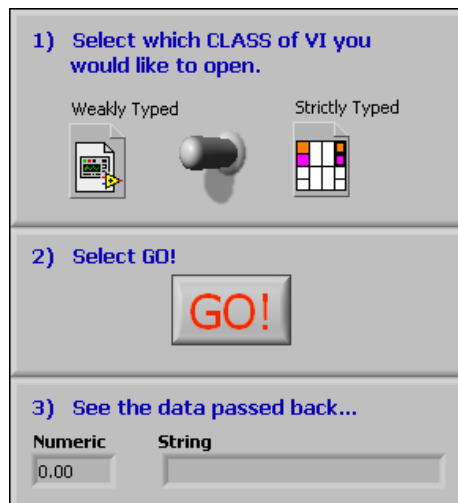
### Description

This exercise demonstrates two ways to dynamically call a VI. The first method is to use a Weakly Typed VI Refnum. This technique is advantageous because it can be used to call any VI, regardless of the VI's connector pane. However, passing data to the VI using a Weakly Typed VI Refnum is difficult.

The second method uses a strictly typed VI refnum. The strictly typed refnum specifies the connector pane for the called VI, and allows you to use a Call By Reference Node, which simplifies the passing of data to the dynamically called VI. However, a Strictly Typed VI Refnum only allows you to call VIs with a matching connector pane. Therefore it is not as flexible as the weakly typed VI refnum.

### Instructions

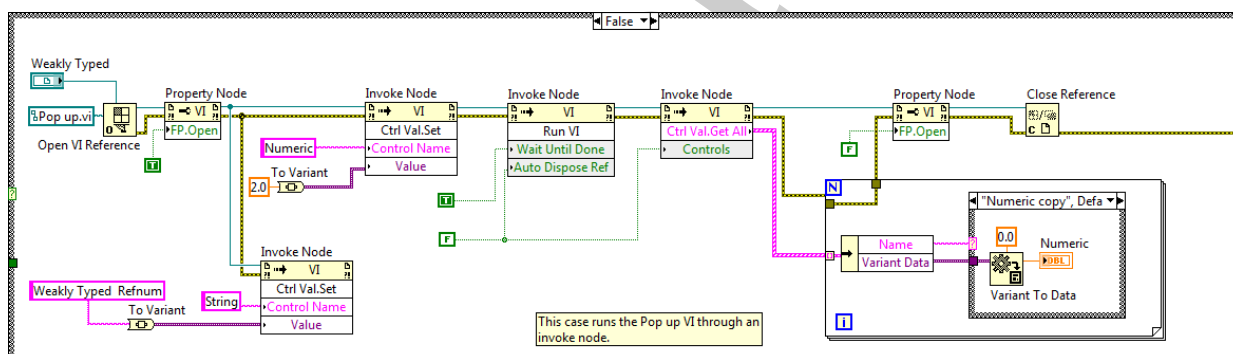
1. Open the Dynamically Calling VIs VI located in the <Exercises>\LabVIEW Connectivity\Dynamically Calling VIs directory.
2. Complete the VI front panel as shown in Figure 2-7 using the following items.
  - ☐ Place a VI Refnum to the left of the Boolean switch.
    - Label the refnum `Weakly Typed`.
    - Right-click the refnum and choose **Select VI Server Class**. Verify that **VI** is checked.
  - ☐ Place a VI Refnum to the right of the Boolean switch.
    - Label the refnum `Strictly Typed`.
    - Right-click the refnum and choose **Select VI Server Class» Browse**. Navigate to <Exercises>\LabVIEW Connectivity\Dynamically Calling VIs directory and select the Pop up VI. Click the **OK** button. The refnum adapts to the connector pane of the Pop up VI.



**Figure 2-7. Weakly versus Strictly Typed Ref VI Front Panel**

3. Complete the False case as shown in Figure 2-8.

Wire the **Weakly Typed** VI refnum to the **type specifier** input of the Open VI Reference function in the False case as shown in Figure 2-8.



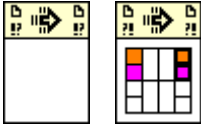
**Figure 2-8.** Weakly VS Strictly Typed Ref VI Block Diagram False Case

The False case contains a VI reference to the Pop up VI. This VI reference opens the front panel of the VI using the Front Panel Window:Open property. The Set Control Value method passes values to the **Numeric** and **String** controls of the Pop up VI.

The Run VI method runs the VI and waits until it completes execution. The Get All Control Values method returns the values of the front panel indicators of the Pop up VI. These values display on the front panel of this VI. Finally, the Close VI Reference function closes the front panel of the Pop up VI and releases the VI Reference.

- Complete the True case on the block diagram as shown in Figure 2-9.

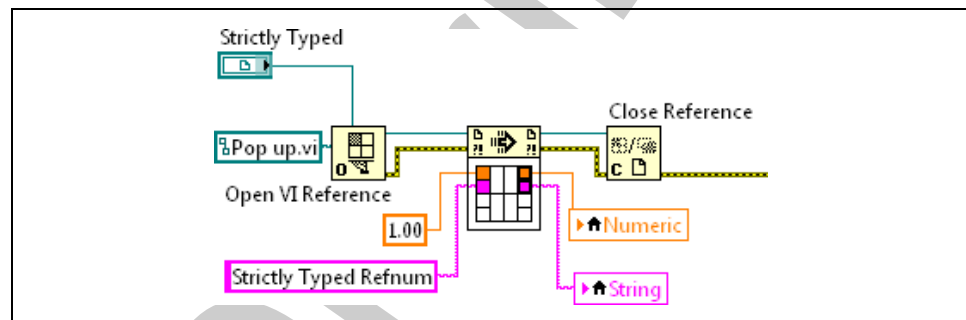
- ☐ Wire the **Strictly Typed** VI refnum to the **Type Specifier** input of the Open VI Reference function.
- ☐ Place a Call By Reference Node on the block diagram.



- Wire the **VI reference** output of the Open VI Reference function to the **reference** input of the Call By Reference Node. The node adopts the connector pane of the Pop up VI, as shown at left.



**Note** When you wire the strictly typed VI refnum for the Pop up VI to the Open VI Reference function, a strictly typed VI reference is generated that you can wire to the Call By Reference Node.



**Figure 2-9.** Block Diagram Code Inside the True Case

- Save the VI.

## Testing

- Run the VI.
- Select the strictly typed reference and click the **GO!** button.

The Pop up VI appears. It returns the value it receives or allows you to change the data. When you finish with the Pop up VI and click the **DONE** button. The front panel of the Dynamically Calling VIs VI shows the values of the indicators from the Pop up VI.

- Run the VI again and select the weakly typed reference. Notice that the behavior is the same as the behavior of the strictly typed reference.

Although both calling methods produce the same result, the Run VI method provides more flexibility and allows you to call a VI asynchronously. If you call a VI asynchronously by passing a false value to the **Wait Until Done** parameter of the Run VI method, the

dynamically called VI executes independently of the calling VI. The calling VI continues its dataflow progression without waiting for the called VI to complete.

The Call By Reference Node simplifies calling a VI dynamically, particularly when passing data to the subVI. The Call By Reference Node requires a strictly typed reference that eliminates the possibility of a run-time type mismatch error. If you do not need the additional flexibility of the Run VI method, use the Call By Reference Node to reduce the complexity of your code.

## End of Exercise 2-4

## Notes

---

Not for Distribution

## Notes

---

Not for Distribution

## Using .Net and ActiveX Objects in LabVIEW Exercises

### Exercise 3-1 Font Dialog

#### Goal

Call a System .NET Assembly to display a Windows Common Dialog box.

#### Scenario

For many applications, you want to provide a familiar look and feel for your user. One technique for doing this is reusing Windows Common Dialogs whenever possible. For example, to have your user select a text font, you can call a Font dialog from the operating system. The Windows Common Font Dialog Box creates a font style and color selection dialog that is familiar to users of most Windows-based word processors, as shown in Figure 3-1.

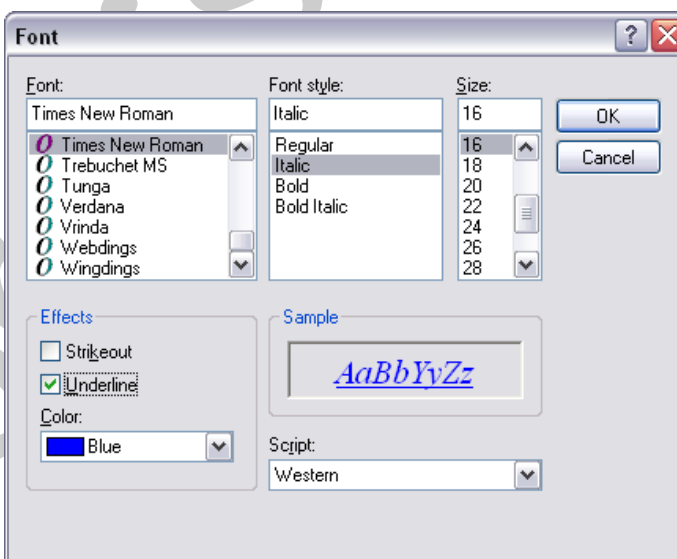


Figure 3-1. Windows Common Font Dialog Box

Create a subVI that calls a font dialog and returns references to the font and color selected by the user. The subVI should use proper error handling techniques and should return a value indicating if the user has canceled the dialog.

## Design

There are multiple ways to display Windows Common Dialogs. The Microsoft Common Dialog Control ActiveX server and ActiveX control provide access to the Color, Font, Help, Open, Printer, and Save dialog boxes. However, these ActiveX components require special licensing to use, which can be acquired through Microsoft Visual Studio. Alternately, you can create a FontDialog object from the System.Windows.Forms .NET assembly. Using the .NET assembly requires the .NET Framework to be installed, but does not require any additional licensing. Furthermore, the .NET assembly provides the newest version of the dialog, which has additional features and improved integration with other Windows components.

### FontDialog Inputs and Outputs

**Table 3-1.** FontDialog Inputs and Outputs

| Type              | Name         | Properties                                    | Default Value |
|-------------------|--------------|-----------------------------------------------|---------------|
| Cluster Control   | error in     | Error Cluster                                 | No Error      |
| Constructor Node  | Font         | .NET Reference,<br>System.Drawing.Font class  | Not a refnum  |
| Constructor Node  | Color        | .NET Reference,<br>System.Drawing.Color class | Not a refnum  |
| Enum Indicator    | DialogResult | Values defined by FontDialog object           | None          |
| Cluster Indicator | error out    | Error Cluster                                 | No Error      |

The FontDialog subVI should perform the following steps:

1. Create a FontDialog object using a .NET Constructor Node.
2. Set the ShowColor property to TRUE so that the font dialog allows the user to select a color.
3. Call the ShowDialog method to show the dialog and return a result. Return the result to the calling VI.
4. Use the Font and Color properties to obtain references to the selected font and color and return these references to the calling VI.
5. Close the reference to the FontDialog object.



## FontDialog Object Description

The FontDialog object creates a common dialog box that displays a list of fonts that are currently installed on the system. You can create a FontDialog class by selecting the FontDialog object from the **System.Windows.Forms** assembly using a .NET Constructor Node. You use the following properties and methods of the FontDialog object in this exercise. Full documentation for the FontDialog and other objects included in the .NET Framework can be found on MSDN or in the documentation for Microsoft Visual Studio .NET.

**ShowColor Property**—Setting this to TRUE instructs the FontDialog to show a selector for the font color. This property should be set before showing the dialog.

**ShowDialog Method**—Displays the font dialog. This method returns an enumerated type which indicates the user's response to the dialog. Notice that the enumerated type returned from this function is shared among many dialogs, and therefore not all the values are actually possible from a FontDialog. A FontDialog typically returns OK or Cancel.

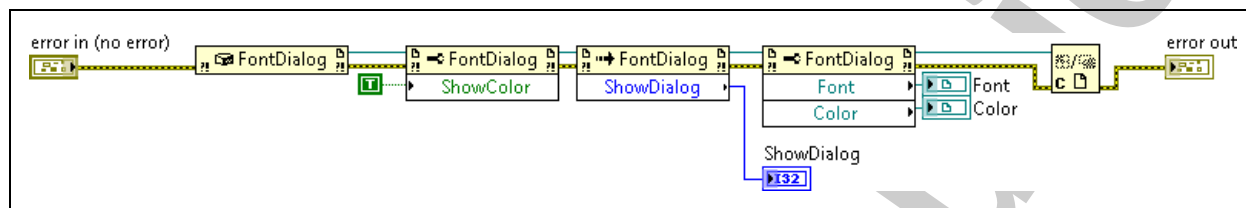
Two versions of the ShowDialog method exists. One takes no parameters and the other takes an IWin32Window object to designate the owning window for the dialog. For this exercise, use the version of the function with no parameters. This may occasionally cause the font dialog to show up behind the main application. You can solve this problem by using the IWin32Window version of the function. However, this requires getting a reference to the window handle of the LabVIEW front panel and converting it to an IWin32Window object, which is beyond the scope of this exercise.

**Font Property**—Returns a .NET reference to a System.Drawing.Font object. You can use this reference to get information about the font, such as the font name and size, or you can pass this reference to other objects that take .NET Font references, such as a .NET RichTextBox control.

**Color Property**—Returns a .NET reference to a System.Drawing.Color object. You can convert this color into a LabVIEW color by using the reference to get the **R**, **G** and **B** properties of the color and then using the RGB to Color VI. Alternately, you can pass this reference to any .NET object which uses colors.

## Implementation

1. Create the VI.
  - ❑ Create a blank VI and save it as `Font Dialog.vi` in the `<Exercises>\LabVIEW Connectivity\Font Dialog` directory.



**Figure 3-2.** Completed Font Dialog Block Diagram

2. Add the following item to the block diagram as shown in Figure 3-2 to open a .NET reference to the FontDialog object.
  - ☐ Place a Constructor Node on the block diagram to display the **Select .NET Constructor** dialog box.
  - ☐ Select **System.Windows.Forms** from the **Assembly** pull-down menu.



**Note** If more than one version of **System.Windows.Forms** is listed, select the latest one.

- Double-click the + to the left of the **System.Windows.Forms** item in the **Objects** list. Scroll down and select **FontDialog** to add it to the **Constructors** list.
- Click **OK**.

3. Add the following items to the block diagram as shown in Figure 3-2 to show the font dialog.

- ❑ FontDialog Property Node.
- Right-click the **new reference** output of the Constructor Node and select **Create»Property for System.Windows.Forms.FontDialog Class»ShowColor** to create the FontDialog Property Node.
- Right-click the FontDialog Property Node and select **Change All To Write** from the shortcut menu.

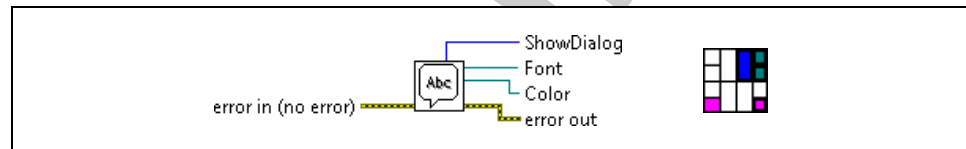
- ☐ **TRUE constant**—Right-click the **ShowColor** input of the FontDialog Property Node and select **Create»Constant**. Set the value of the constant to TRUE.
  - ☐ **FontDialog Invoke Node**—Right-click the **reference** output of the FontDialog Property Node and select **Create»Method for System.Windows.Forms.FontDialog Class»ShowDialog()** from the shortcut menu to create a FontDialog Invoke Node.
  - ☐ **Ring Indicator**—Right-click the **ShowDialog** output of the FontDialog Invoke Node and select **Create»Indicator** from the shortcut menu to create a ring indicator described in Table 3-1.
4. Add the following items to the block diagram as shown in Figure 3-2 to get .NET references to font and color.
- ☐ FontDialog Property Node
    - Right-click the **reference** output of the FontDialog Invoke Node and select **Create»Property for System.Windows.Forms.FontDialog Class»Font** to create another FontDialog Property Node.
    - Expand the second FontDialog Property Node to show two elements. Select **Color** as the second element.
  - ☐ Right-click the **Font** output of the FontDialog Property Node and select **Create»Indicator** from the shortcut menu to create the **Font** output described in Table 3-1.
  - ☐ Right-click the **Color** output of the FontDialog Property Node and select **Create»Indicator** from the shortcut menu to create the **Color** output described in Table 3-1.
5. Add the following items to the block diagram as shown in Figure 3-2 to close the reference and handle errors.
- ☐ Place a Close Reference function on the block diagram.
  - ☐ Right-click the **error in** input of the Constructor Node and select **Create»Control** from the shortcut menu to create the **error in** input described in Table 3-1.
  - ☐ Right-click the **error out** output of the Close Reference function and select **Create»Indicator** from the shortcut menu to create the **error out** output described in Table 3-1.

- ☐ Wire the error wire through the Error case of the Case structure to the **error out** indicator.
- 6. Create the icon and connector pane.
  - ☐ Switch to the VI front panel.
  - ☐ Organize the controls in a logical manner.
  - ☐ Right-click the **ShowDialog** indicator and select **Replace»Modern»Ring & Enum»Enum** from the shortcut menu.



**Note** Converting the ring indicator into an enumerated type indicator allows you to better control Case structures with the result of the dialog.

- ☐ Create an icon and connector pane similar to Figure 3-3.



**Figure 3-3.** Font Dialog icon and Connector Pane

- 7. Save the VI.

## Testing

Test the VI as a top-level VI.

- ☐ Run the VI. A font dialog should display. Notice that the font dialog may be behind the front panel. Minimize the front panel or press <Alt-Tab> to find the font dialog window.
- ☐ Click **OK** in the font dialog to finish the VI.

## Challenge

Test the VI as a subVI.

- ☐ Create a VI that calls the `FontDialog` VI.
- ☐ Check the **ShowDialog** to determine if the user clicked the **OK** button.
- ☐ Use the Font reference to display the selected Font Name.
- ☐ Use the Color reference to display the selected color in a LabVIEW **Color Box** indicator.



**Tip** Refer to the *Design* section for a suggestion on how to convert a .NET Color reference to a LabVIEW color.

## End of Exercise 3-1