LabVIEW™ Connectivity Course Manual

Course Software Version 2010
May 2011 Edition
Part Number 325627A-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

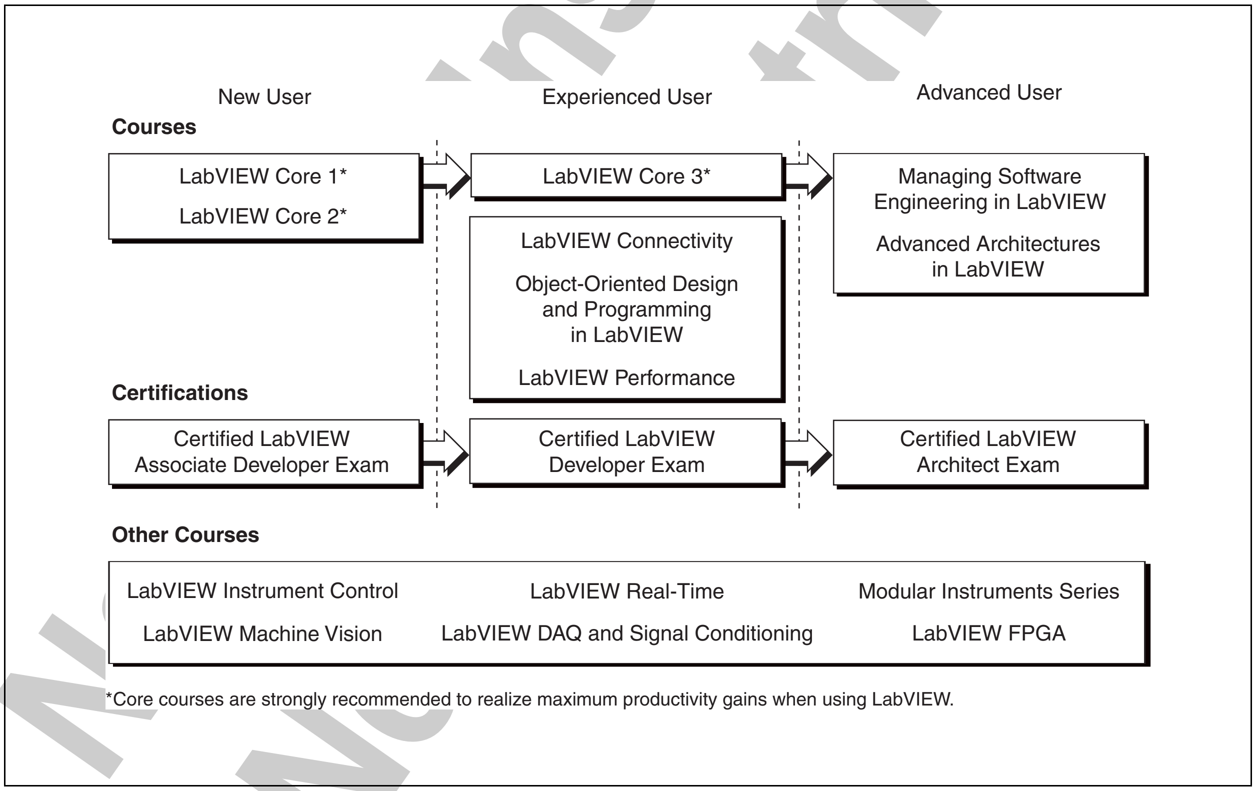
Student Guide

Thank you for purchasing the *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.



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

B. Course Description

The *LabVIEW Connectivity* course teaches you how to use advanced connectivity in VIs. This course 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

❑ LabVIEW Professional Development System 2010 or later

❑ Database Connectivity Toolkit

❑ *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
- 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

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.

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.
`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.
`monospace italic`	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.

1

Calling Shared Libraries in LabVIEW

You can use LabVIEW to call code written in other languages in the following ways:

- Using platform-specific protocols.
- Using the Call Library Function Node to call the following types of shared libraries:
  - Dynamic Link Libraries (DLL) on Windows
  - Frameworks on Mac OS
  - Shared Libraries on Linux

The scope of this course covers using the Call Library Function Node to call Dynamic Link Libraries (shared libraries) in Windows. Refer to the *LabVIEW Help* for information about other ways to call code written in other languages from LabVIEW.

Topics

A. Shared Libraries Overview

On Windows, a shared library is called a DLL. A shared library is a library of executable functions or data that can be used by a Windows application. A shared library provides one or more functions that a program can access by creating a static or dynamic link to the shared library. A static link remains constant during program execution and a dynamic link is created by the program as needed. The shared library is stored as a binary file.

You can use any language to write shared libraries as long as the shared libraries can be called using one of the calling conventions LabVIEW supports, either `stdcall` or `C`. Examples and troubleshooting information help you build and use shared libraries and successfully configure the Call Library Function Node in LabVIEW. The general methods described here for DLLs also apply to other types of shared libraries.

LabVIEW loads shared libraries in a unique application instance. Opening a shared library in a unique application instance prevents naming conflicts with VIs in the shared library, and VIs outside of the shared library.

Refer to the `labview\examples\dll` directory for examples of using shared libraries.

B. Calling Shared Libraries

This section describes the process of using a shared library in LabVIEW.

Describing and Defining Shared Libraries

You can call most standard shared libraries with the Call Library Function Node. On Windows, these shared libraries are DLLs, on Mac OS, they are Frameworks, and on Linux, they are Shared Libraries. The Call Library Function Node supports a large number of data types and calling conventions. You can use the Call Library Function Node to call functions from most standard and custom-made libraries.

Purposes, Advantages, and Limitations of DLLs

Most modern development environments provide support for creating DLLs.

In some cases, you may want to perform additional tasks at certain execution times. For this purpose, the Call Library Function Node provides three entry points—Reserve, Unreserve, and Abort. For example, you may want to initialize data structures at reserve time for the Call Library Function Node or free private data when the Call Library Function Node is unreserved or aborted. For these situations, you can write and export routines from your

DLL that LabVIEW calls at these predefined times. You can configure these entry points on the **Callbacks** page of the **Call Library Function** dialog box.

The Call Library Function Node is most appropriate when you have existing code you want to call, or if you are familiar with the process of creating standard shared libraries. Because a library uses a format standard among several development environments, you can use almost any development environment to create a library that LabVIEW can call. Refer to the documentation for your compiler to determine whether you can create standard shared libraries.

The LabVIEW compiler can generate code fast enough for most programming tasks. You can call shared libraries from LabVIEW to complete tasks for which another language might be better suited.

Shared libraries execute synchronously, so LabVIEW cannot use the execution thread used by these objects for any other tasks.

LabVIEW cannot interrupt object code that is running, so you cannot reset a VI that is running a shared library until execution completes. If you want to write a shared library that performs a long task, be aware that LabVIEW cannot perform other tasks in the same thread while these objects execute.

Method for Calling Shared Libraries

Use the Call Library Function Node to directly call a Windows DLL, a Mac OS Framework, or a Linux Shared Library function. With this node, you can create an interface in LabVIEW to call existing libraries or new libraries specifically written for use with LabVIEW. National Instruments recommends using the Call Library Function Node to create an interface to external code.

Note Be aware when using the Call Library Function Node or writing code that is called by the Call Library Function Node that LabVIEW reserves Windows messages WM_USER through WM_USER+99 for internal use only.

Right-click or double-click the Call Library Function Node and select **Configure** from the shortcut menu to display the **Call Library Function** dialog box. Use the **Call Library Function** dialog box to specify the library, function, parameters, return value for the node, calling conventions, and function callbacks on Windows. When you click the **OK** button in the **Call Library Function** dialog box, LabVIEW updates the Call Library Function Node according to your settings, displaying the correct number of terminals and setting the terminals to the correct data types.

Note If you want to run applications or shared libraries created with different versions of LabVIEW on the same computer, the computer must have a version of the LabVIEW Run-Time Engine that is compatible with each version of LabVIEW used to create the applications or shared libraries.

Setting the Calling Convention

Calling conventions define the way to pass information from a piece of code to a function. Use the **Calling convention** control on the **Function** tab of the **Call Library Function** dialog box to select the calling convention for the function. The default calling convention is `C`. The `C` calling convention allows variable-length parameter lists and passes parameters onto the stack in reverse order. This can incur a slight speed decrease.

(Windows) You also can use the standard Windows calling convention, `stdcall`. When you use `stdcall`, parameters are passed by a function onto the stack in the same order as they appear in the function declaration. The number of parameters passed to the function is fixed.

Refer to the documentation for the DLL you want to call for the appropriate calling conventions.

Caution Using the incorrect calling convention can cause an irregular shutdown of LabVIEW.

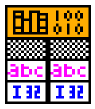
Configuring Parameters

To configure parameters for the Call Library Function Node, navigate to the **Parameters** tab of the **Call Library Function** dialog box. Initially, the Call Library Function Node has no parameters and has a return type of **Void**.

As you configure parameters, the **Function prototype** text box displays the C prototype for the function you are building. This text box is a read-only display.

Note If a type library is found, the parameters are updated to match the parameters found in the type library for the selected function. The order of the parameters must match the prototype of the function found in the library.

The return type for the Call Library Function Node returns to the right terminal of the top terminal. If the return type is **Void**, the top terminal is unused. Each additional pair of terminals corresponds to a parameter in the **Parameters** list of the Call Library Function Node. To pass a value to the Call Library Function Node, wire to the left terminal of a terminal pair. To read the value of a parameter after the Call Library Function Node call, wire from the right terminal of a terminal pair. The following illustration shows a Call Library Function Node that has a return type of **Void**, a string parameter, and a numeric parameter.



Return Type

For **return type**, you can set **Type** to **Void**, Numeric, or String. **Void** is only available for return type and is not available for other parameters. Use **Void** for the return type if your function does not return any values.

Even if the function you call returns a value, you can use **Void** for the return type. When the function returns a value and you select **Void** as the return type, the value returned by the function is ignored.

Note The function you are calling can return a C string pointer. If you want to deallocate the pointer, you must do so explicitly as LabVIEW does not automatically deallocate the C string pointer for you.

Tip If the function you are calling returns a data type not listed, choose a return data type the same data size as the one returned by the function. For example, if the function returns a char data type, use an 8-bit unsigned integer. A call to a function in a DLL cannot return a pointer because there are no pointer types in LabVIEW. However, you can specify the return type as an integer that is the same size as the pointer. LabVIEW then treats the address as a simple integer, and you can pass it to future DLL calls.

Adding and Deleting Parameters

To add parameters to the Call Library Function Node, navigate to the **Parameters** tab of the **Call Library Function** dialog box. Click the **Add a parameter** button. To remove a parameter, click the **Delete the selected parameter** button. To change the order of the parameters, use the **Move the selected parameter up one** and **Move the selected parameter down one** buttons to the right of the parameter list.

Editing Parameters

Select the parameter from the **Parameters** list to edit the data type or parameter name. You can edit the parameter name to something more descriptive, which makes it easier to distinguish between parameters. The parameter name does not affect the call, but it is propagated to output wires. Also, you can edit all fields in the **Current parameter** section for the selected parameter.

Selecting the Parameter Type

Use the **Type** pull-down menu to indicate the data type of each parameter. You can select from the following parameter types:

- Numeric
- Array
- String
- Waveform
- Digital Waveform
- Digital Data
- ActiveX
- Adapt to Type
- Instance Data Pointer

After you select an item from the **Type** pull-down menu, you see more items you can use to indicate details about the parameter and about how to pass the data to the library function. The Call Library Function Node has a number of different items for parameter types because of the variety of data types required by different libraries. Refer to the documentation for the library you call to determine which parameter types to use.

The following sections discuss the different parameter types available from the **Type** pull-down menu.

(Windows) Refer to the `labview\examples\dll\data passing\Call Native Code.llb` for an example of using data types in shared libraries.

Numeric

For numeric data types, you must indicate the exact numeric type by using the **Data Type** pull-down menu. You can choose from the following data types:

- 8-, 16-, 32-, 64-bit, and pointer-sized signed and unsigned integers
- 4-byte, single-precision numbers
- 8-byte, double-precision numbers

If you use pointer-sized integers, the Call Library Function Node adapts to the bitness of the version of LabVIEW it is being executed on and passes data of the appropriate size to and from the library function. LabVIEW represents the data in 64 bits and, on 32-bit LabVIEW, translates the numeric data types to 32-bit integer types.

Note You can pass extended-precision numbers and complex numbers by selecting **Adapt to Type** from the **Type** pull-down menu. However, standard libraries generally do not use extended-precision numbers and complex numbers.

Use the **Pass** pull-down menu to indicate whether you want to pass the value or a pointer to the value.

Array

Use the **Data Type** pull-down menu to indicate the data type of the array. You can choose from the same data types available for numeric parameters.

Specify the dimensions of the array in **Dimensions**.

Use the **Array format** pull-down menu to make one of the following choices:

- **Array Data Pointer**—passes a pointer to the array data, allowing the called library to access the array data as the specified data type of the array data.
- **Array Handle**—passes a pointer to a pointer to a four-byte value for each dimension, followed by the data.
- **Array Handle Pointer**—passes a pointer to an array handle.

Use the **Minimum size** control to have LabVIEW check at run time that the memory LabVIEW allocated for an array data pointer is at least the **Minimum size**. To indicate the **Minimum size** of a 1D array, you can enter a numeric value, or, if you configure an integer parameter in the **Parameters** list, you can select the parameter from the pull-down menu. This option is available only for one dimensional array data pointers.

Note If you pass in an array that is smaller than the **Minimum size**, LabVIEW enlarges the size of the array to the minimum. However, if you pass in an array that is bigger than the minimum, the array retains the larger size.

String

Use the **String format** pull-down menu to indicate the string format. You can choose from the following string formats:

- **C String Pointer**—a string followed by a null character.
- **Pascal String Pointer**—a string preceded by a length byte.
- **String Handle**—a pointer to a pointer to four bytes for length information, followed by string data.
- **String Handle Pointer**—a pointer to an array of string handles.

Select a string format that the library function expects. Most standard libraries expect either a C string or a Pascal string. If the library function you are calling is written for LabVIEW, you might want to use the **String Handle** format. When configuring a Pascal string pointer, you must wire a value to the string input on the block diagram. That value must be initialized with enough characters to hold any new string that may be written to that Pascal string. When configuring a C string pointer, you have two options:

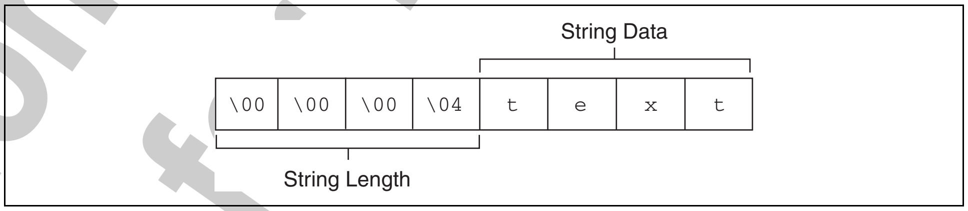
- Wire a value to the string input that is initialized with enough characters to hold any new string that may be written to that string.
- Specify the string size in the **Minimum size** pull-down menu on the **Parameters** tab of the **Call Library Function** dialog box.

Use the **Minimum size** control to have LabVIEW check at run-time that the memory LabVIEW allocated for a C string pointer is at least the **Minimum size**. To indicate the **Minimum size** of a string, you can enter a numeric value, or, if you configure an integer parameter in the **Parameters** list, you can select the parameter from the pull-down menu. This option is available only for C string pointers.

Note If you pass in a string that is smaller than the **Minimum size**, LabVIEW enlarges the size of the string to the minimum. However, if you pass in a string that is bigger than the minimum, the string retains the larger size.

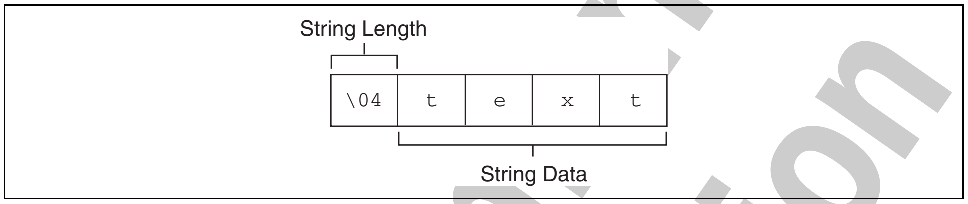
String Options

LabVIEW stores strings as arrays, that is, structures pointed to by handles. The Call Library Function Node works with C and Pascal-style string pointers or LabVIEW string handles. The following illustration shows an example of a LabVIEW string handle.

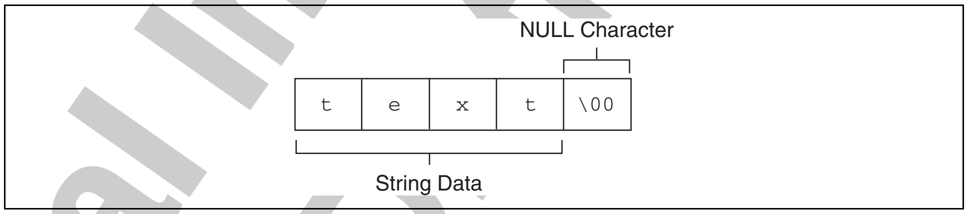


Think of a string as an array of characters. Assembling the characters in order forms a string. LabVIEW stores a string in a special format in which the first four bytes of the array of characters form a 32-bit signed integer that stores how many characters appear in the string. Thus, a string with n characters requires n + 4 bytes to store in memory. For example, the string `text` contains four characters. When LabVIEW stores the string, the first four bytes contain the value 4 as a 32-bit signed number, and each of the following four bytes contains a character of the string. The advantage of this type of string storage is that NULL characters are allowed in the string. Strings are virtually unlimited in length, up to 2^{31} characters.

The Pascal string format is similar to the LabVIEW string format, but instead of storing the length of the string as a 32-bit signed integer, it is stored as an 8-bit unsigned integer. This limits the length of a Pascal-style string to 255 characters. A graphical representation of a Pascal string appears in the following illustration. A Pascal string that is *n* characters long requires *n* + 1 bytes of memory to store.



C strings are probably the type of strings you deal with most often. The similarities between the C-style string and normal numeric arrays in C becomes much clearer when you notice that C strings are declared as `char *`. Unlike LabVIEW and Pascal strings, C strings do not contain information that indicates the length of the string. Instead, C strings use a special NULL character to indicate the end of the string, as shown in the following illustration. NULL has a value of zero in the ASCII character set. Notice that this is the number zero and not the character `0`.



In C, a string containing *n* characters requires *n* + 1 bytes of memory to store, *n* bytes for the characters in the string and one additional byte for the NULL termination character. The advantage of C-style strings is that they are limited in size only by available memory. However, if you acquire data from an instrument that returns numeric data as a binary string, as is common with serial or GPIB instruments, values of zero in the string are possible. For binary data where NULLs might be present, use an array of 8-bit unsigned integers or use a string handle. If you treat the string as a C-style string, when the instrument returns a numeric value of zero, the program assumes incorrectly that the end of the string has been reached.

Waveform

When you call a shared library that includes a waveform data type, you do not have to specify a numeric value from the **Data Type** pull-down menu; the default is **8-byte Double**. However, you must specify **Dimensions**. If the parameter is a single waveform, specify **Dimensions** as `0`. If the parameter is an array of waveforms, specify **Dimensions** as `1`. LabVIEW does not support an array of waveforms with more than one dimension.

Digital Waveform

Specify **Dimensions** as `0` if the parameter is a single digital waveform. Specify **Dimensions** as `1` if the parameter is an array of digital waveforms. LabVIEW does not support an array of digital waveforms with more than one dimension.

Digital Data

Specify **Dimensions** as `1` if the Parameter is an array of digital data. Otherwise, specify Dimensions as `0`. LabVIEW does not support an array of digital data with more than one dimension.

Note You can pass waveforms, digital waveforms, and digital data through shared libraries, but you cannot access the data inside the shared libraries.

ActiveX

Select one of the following items from the **Datatype** pull-down menu:

- **ActiveX Variant Pointer**—passes a pointer to ActiveX data.
- **IDispatch* Pointer**—passes a pointer to the IDispatch interface of an ActiveX Automation server.
- **IUnknown* Pointer**—passes a pointer to the IUnknown interface of an ActiveX Automation server.

Adapt to Type

Use **Adapt to Type** to pass arbitrary LabVIEW datatypes to DLLs. The arbitrary LabVIEW data types are passed to DLLs in the following ways:

- Scalars are passed by reference. A pointer to the scalar is passed to the library.
- Arrays and strings are passed according to the **Data format** setting. You can choose from the following **Data format** settings:
  - **Handles by Value** passes the handle to the library. The handle is not NULL.
  - **Pointers to Handles** passes a pointer tothe handle to the library. If the handle is NULL, treat the handle as an empty string or array. To set a value when the handle is NULL, you must allocate a new handle.
  - **Array Data Pointer** passes a pointer to the first element of the array, allowing the called library to access the array data as the data type of the array data.
- Clusters are passed by reference.

- Scalar elements in arrays or clusters are in line. For example, a cluster containing a numeric is passed as a pointer to a structure containing a numeric.
- Clusters within arrays are in line.
- Strings and arrays within clusters are referenced by a handle.

Note When one or more of the parameters of the function you want to call in a DLL are of types that do not exist in LabVIEW, ensure that each parameter is passed to the function in a way that allows the DLL to correctly interpret the data. Create a skeleton `.c` file from the current configuration of the Call Library Function Node. By viewing the `.c` file, you can determine whether LabVIEW will pass the data in a manner compatible with the DLL function. You then can make any necessary adjustments.

Instance Data Pointer

Use **Instance Data Pointer** to access data allocated for each instance of the Call Library Function Node. The **Instance Data Pointer** references a pointer sized allocation that you may use at your own discretion. This allocation is also passed to each of the callback functions on the **Callbacks** tab.

Working with Unusual Data Types

You might encounter a function that expects data in a form that the Call Library Function Node cannot pass. Specifically, the Call Library Function Node does not support structures or arrays containing a pointer to other data or structures containing flat arrays that can be variably sized. You can call a function that expects an unsupported data type in the following ways:

- If the data contains no pointers, you might be able to use the Flatten To String function to create a string containing the binary image of the data required and pass this string as a C string pointer. You will probably want to use the **byteorder** input to Flatten To String to specify that the data be flattened in native byte order.
- Write a library function that accepts the data in the form used by LabVIEW and builds the data structure expected by the other library. This function then can call the other library and retrieve any returned values before returning. Your function will probably accept the data from the block diagram as Adapt to Type, so that any block diagram data type can be passed.

Thread-Safe and Thread-Unsafe DLLs

You can select the thread to execute the library call from the **Thread** section on the **Function** tab of the **Call Library Function** dialog box. The thread options are **Run in UI thread** and **Run in any thread**. If you select **Run in UI thread**, the Call Library Function Node switches from the thread the VI is currently executing in to run in the LabVIEW user interface thread. If you select **Run in any thread**, the Call Library Function Node continues in the currently executing thread. By default, all Call Library Function Nodes run in the LabVIEW user interface thread.

Before you configure a Call Library Function Node to run in any thread, make sure that multiple threads can call the function simultaneously. In a shared library, code can be considered thread-safe when:

- It does not store any global data, such as global variables, files on disk, and so on.
- It does not access any hardware. In other words, the code does not contain register-level programming.
- It does not make any calls to any functions, shared libraries, or drivers that are not thread safe.
- It uses semaphores or mutexes to restrict access to global resources.
- It is called by only one non-reentrant VI.

Refer to the *Execution Properties Page* topic of the *LabVIEW Help* for more information about reentrancy. Refer to the *Benefits of Multithreaded Applications* topic of the *LabVIEW Help* for more information about multithreading in LabVIEW.

To practice the concepts in this section, complete Exercise 1-1.

C. Using the Import Shared Library Wizard

Use the **Import Shared Library** wizard to create or update a LabVIEW project library of wrapper VIs for functions in a Windows `.dll` file, a Linux `.so` file, or a Macintosh `.framework` file. The **Import Shared Library** wizard supports most C and C++ header files. The wrapper VIs the wizard creates use the Call Library Function Node, which does not support the C++ `this` pointer or calling methods in C++ classes.

Note If you want to import a shared library file for an instrument driver, you can download the LabVIEW Instrument Driver Import Wizard from the Instrument Driver Network on `ni.com`.

The **Import Shared Library** wizard parses the header file, lists the functions in the shared library, converts data types in the shared library to LabVIEW data types, and generates a wrapper VI for each function. The wizard saves the VIs in a LabVIEW project library that you can edit and creates an HTML report about the generated library that you can launch when you complete the wizard.

In the wizard, you can specify include paths and preprocessor definitions, configure the individual VIs that wrap each function, and configure memory allocation and error handling. The wizard also creates custom controls for structure elements in the original functions and adds the controls to the project library. You can use the custom controls to modify all the VIs in the library that contain the corresponding data type.

You can run the wizard multiple times on the same shared library file. If you select the **Update VIs for a shared library** option on the **Specify Create or Update Mode** page, the wizard overwrites the previous version of the project library file and the existing VIs within that file. If you choose not to re-import generated VIs within the project library file, the VIs remain unchanged in the directory. The wizard retains the most recent settings for each individual function in a particular shared library. For example, if you have a shared library that contains three functions, you might update only the second function. The next time you run the wizard on that shared library file, it retains the original settings for functions one and three and uses the new settings for function two.

Select **Tools»Import»Shared Library** to launch the **Import Shared Library** wizard. Follow the prompts to create wrapper VIs for shared library files. You must provide the name of a shared library file and a header (`.h`) file for the wizard to parse.

Refer to the *Importing Functions from a Shared Library File* topic of the *LabVIEW Help* for step-by-step instructions for importing a shared library and creating wrapper VIs.

Refer to the Import Shared Library Tutorial GUI VI in the `labview\examples\dll\regexpr` directory for examples of using the **Import Shared Library** wizard.

Self-Review: Quiz

1. Which of the following do you need to know to call a function in a shared library?
   a. Data type returned by the function
   b. Calling convention
   c. Development environment that created the shared library
   d. Order of parameters to be sent to the function

2. You have inherited a DLL that accesses hardware and is known not to be thread safe. Which configuration would you choose for your Call Library Function Node?
   a. Run in UI thread
   b. Run in any thread

3. True or False? The only file the **Import Shared Library** wizard requires is a shared library.

Self-Review: Quiz Answers

1. Which of the following do you need to know to call a function in a shared library?

   a. Data type returned by the function

   b. Calling convention

   c. Development environment that created the shared library

   d. Order of parameters to be sent to the function

2. You have inherited a DLL that accesses hardware and is known not to be thread safe. Which configuration would you choose for you Call Library Function Node?

   a. Run in UI thread

   b. Run in any thread

3. True or False? The only file the **Import Shared Library** wizard requires is a shared library.

   False. The Import Shared Library requires both a shared library and its header file.

Notes

2

Using VI Server

The VI Server allows you to programmatically control VIs and LabVIEW. You can access the VI Server through block diagrams, ActiveX technology, and the TCP protocol. You can perform VI Server operations on a local computer or remotely across a network.

Topics

A. Capabilities of the VI Server

The VI Server is an object-oriented, platform-independent technology that provides programmatic access to LabVIEW and LabVIEW applications. The VI Server is a set of functions that allows you to dynamically control front panel objects, VIs, and the LabVIEW environment. These functions are located in the **Functions»Application Control** subpalette. Use the VI Server to perform the following programmatic operations:

- Edit the properties of a VI and LabVIEW. For example, you can dynamically determine the location of a VI window or scroll a front panel so that a part of it is visible. You also can programmatically save any changes to disk.
- Dynamically load VIs into memory when another VI needs to call them, rather than loading all subVIs when you open a VI.
- Call a VI dynamically.
- Call a VI remotely.
- Control a LabVIEW application from another program.
- Update the properties of multiple VIs rather than manually using the **File»VI Properties** dialog box for each VI.
- Retrieve information about an application instance, such as the version number and edition. You also can retrieve environment information, such as the platform on which LabVIEW is running.
- Create a plug-in architecture for the application to add functionality to the application after you distribute it to customers. For example, you might have a set of data filtering VIs, all of which take the same parameters. By designing the application to dynamically load these VIs from a plug-in directory, you can ship the application with a partial set of these VIs and make more filtering options available to users by placing the new filtering VIs in the plug-in directory.

Note The VI Server also allows you to programmatically inspect, modify and create block diagram options using VI Scripting. Refer to the *Programatically Scripting VIs* in *LabVIEW* topic of the *LabVIEW Help*.

VI Server Clients

The VI Server has a set of methods and properties that are accessible through the different clients, as shown in Figure 2-1. The different client interfaces are ActiveX Automation client, TCP/IP client, and LabVIEW functions that allow LabVIEW VIs to access the VI Server.

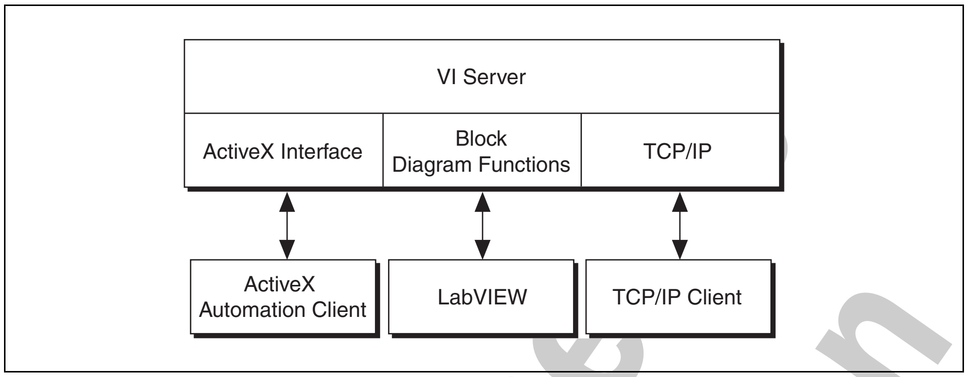


Figure 2-1. VI Server Clients

- **Block Diagram Access**—LabVIEW includes a set of built-in functions located on the **Application Control** palette you can use to access the VI Server on a local or remote computer.
- **Network Access**—If you are using the VI Server on a remote computer, LabVIEW uses the TCP/IP protocol as the means of communication.
- **ActiveX Interface**—ActiveX clients such as Visual Basic, Visual C++, and Excel applications can use the VI Server to run LabVIEW applications.

Application and VI Objects

You access VI Server functionality through references to two main classes of objects—the Application object and the VI object. After you create a reference to one of these objects, you can pass the reference to a VI or function that performs an operation on the object.

An Application reference refers to a local or remote LabVIEW application instance. You can use Application properties and methods to change LabVIEW preferences and return system information. A VI refnum refers to a VI in an application instance.

With a reference to an application instance, you can retrieve information about the LabVIEW environment, such as the platform on which LabVIEW is running, the version number, or a list of all VIs currently in memory. LabVIEW opens a new application instance when you create a LabVIEW project or a target for a LabVIEW project. You also can set information, such as the list of VIs exported to other application instances. Because you can open multiple application instances at once, you must use an application reference when you are using VI Server properties and methods in one application instance, and you want to interact with a different application instance.

When you create a refnum to a VI, LabVIEW loads the VI into memory. With a refnum to a VI, you can update all the properties of the VI available in the **File»VI Properties** dialog box as well as dynamic properties, such as the position of the owning pane. You also can print the VI documentation programmatically, save the VI to another location, and export and import its strings to translate into another language.

For more information about VI references, refer to the *VI Properties* and *VI Methods* topics of the *LabVIEW Help*.

B. VI Server Programming Model

The programming model for VI Server applications is based on refnums. Refnums also are used in file I/O, network connections, and other objects in LabVIEW.

Typically, you open a refnum to an application instance or to a VI. You then use the refnum as a parameter to other VIs. The VIs get (read) or set (write) properties, execute methods, or dynamically load and call a referenced VI. Finally, you close the refnum, which releases the referenced object.

Figure 2-2 illustrates the VI Server programming model.

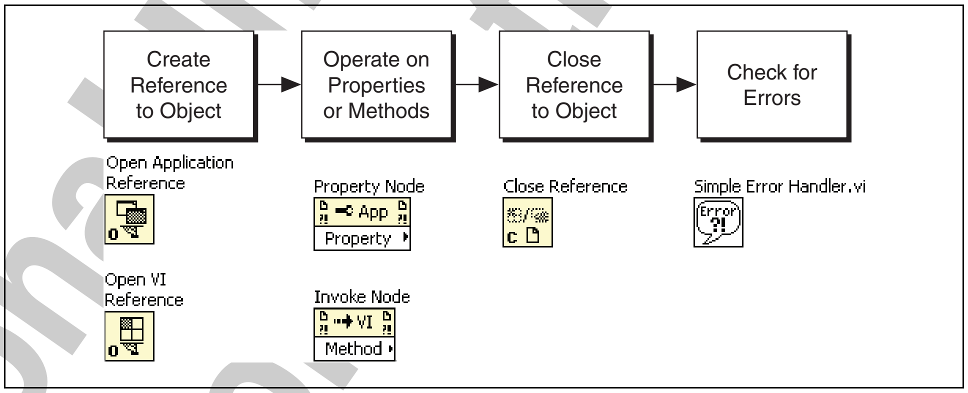


Figure 2-2. VI Server Programming Model

Complete the following steps to create a VI Server application.

1. (Optional) Configure the VI Server to allow the TCP/IP protocol. This allows other LabVIEW applications or other application instances in the LabVIEW application to communicate with this LabVIEW.
2. (Optional) Use the Open Application Reference function to open a reference to a local or remote application instance.

Note If you have multiple application instances open simultaneously, such as if you are working with a LabVIEW project or targets of a LabVIEW project, there can be multiple VI Servers listening on different ports. Open an application reference to a specific application instance by stating the machine name and the port or service name.

3. Use the Open VI Reference function to open a reference to a VI on the local or remote computer that already exists in memory for the application instance, to dynamically load a VI from disk, or to dynamically load a VI from disk.
4. Use the Property Node to get or set properties or the Invoke Node to invoke methods.

   You also can use a Call By Reference Node to call a dynamically loaded VI. The Call By Reference Node requires a strict VI reference.
5. Use the Close Reference function to close any open references.
6. Check for errors.

Refer to `labview\examples\viserver` for examples of using the VI Server.

C. VI Server Functions

Use the following **Application Control** functions and nodes to build VI Server applications:

- **Open Application Reference**—Opens a reference to a local or remote application instance.
- **Open VI Reference**—Opens a reference to a VI on the local or remote computer or dynamically loads a VI from disk.
- **Property Node**—Gets and sets VI, object, or application properties.
- **Invoke Node**—Invokes methods on a VI, object, or application.
- **Call By Reference Node**—Calls a dynamically loaded VI.
- **Close Reference**—Closes open references to the VI, object, or application you accessed using the VI Server.
- **Static VI Reference**—Maintains a static reference to a VI.

Open Application Reference Function

The Open Application Reference function returns a reference to a VI Server application instance running on the specified computer. If you specify an empty string for **machine name**, the function returns a reference to the local LabVIEW application instance in which this function is running. If you do specify a **machine name**, the function attempts to establish a TCP connection with a remote VI Server on that machine on the specified port. **machine name** can be in dotted decimal notation, such as `123.23.45.100`, or domain name notation, such as `remotemachine.ni.com`.

The **port number or service name** input can accept a numeric or a string input. The default is numeric. **port number or service name** is the port number on which the remote LabVIEW application is listening. If you specify a service name, LabVIEW queries the NI Service Locator for the port number that the server registered. The default is to use the default VI Server listener port number (3363).

You use the **application reference** output of the function as an input to Property Nodes and Invoke Nodes to get or set properties and invoke methods on the LabVIEW application. You also use **application reference** as an optional input to the Open VI Reference function to get references to VIs that are running in the LabVIEW application.

Open VI Reference Function

The Open VI Reference function returns a reference to a VI, custom control, or global variable specified by a name string or path to the location of the VI on disk.

You can get references to VIs in another LabVIEW application instance by wiring an **application reference**, obtained from the Open Application Reference function, to this function. If you do not wire data to the **application reference** input unwired, the Open VI Reference function refers to the application instance the Open VI Reference function is running in.

Property Nodes

Use the Property Node to get and set various properties of an application or VI. Select properties from the node by using the Operating tool to click the property terminal or by right-clicking the white area of the node and selecting **Properties** from the shortcut menu. You also can create an implicitly linked Property Node by right-clicking a front panel object, selecting **Create»Property Node**, and selecting a property from the shortcut menu.

The following are examples of how properties can enhance ease of use in an application or VI:

- Set the front panel window size.
- Set the VI window title.

You can read or write multiple properties using a single node. However, some properties are not readable and some are not writable. Use the Positioning tool to resize the Property Node to add new terminals. A small direction arrow to the right of the property indicates a property you read. A small direction arrow to the left of the property indicates a property you write. Right-click the property and select **Change to Read** or **Change to Write** from the shortcut menu to change the operation of the property.

The Property Node executes from top to bottom. The node does not execute if an error occurs before it executes, so always check for the possibility of errors. If an error occurs in a property, LabVIEW ignores the remaining properties and returns an error. If you right-click the Property Node and select **Ignore Errors inside Node**, LabVIEW executes the remaining properties on the Property Node. The Property Node returns only the first error. The **error out** cluster contains information about which property caused the error.

Invoke Nodes

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

Close Reference Function

The Close Reference function releases the application or VI reference. If you close a reference to a specified VI and there are no other references to that VI, LabVIEW can unload the VI from memory.

This function does not prompt you to save changes to the VI. By design, VI Server actions should avoid causing user interaction. You must use the Save:Instrument method to save the VI programmatically.

The VI stays in memory until you close the reference and until the VI meets the following conditions:

1. There are no other open references to the referenced VI.
2. The front panel of the VI is not open.
3. The VI is not a subVI of another VI in memory.
4. The VI is not a member of an open project library.

Note If you do not close the application or VI reference with this function, the reference closes automatically when the top-level VI associated with this function finishes execution. However, it is a good programming practice to conserve the resources involved in maintaining the connection by closing the reference when you finish using it.

Static VI Reference

The Static VI Reference function acts as a subVI and appears in the VI hierarchy of the top-level VI. By default, the output is a generic VI reference.

You can change the output of this function to a strictly typed VI reference. Right-click the function and select **Strictly Typed VI Reference** from the shortcut menu to change the output. A red star in the center of the function icon indicates the reference is strictly typed. The strictly typed VI reference identifies the connector pane of the VI you are calling. You can create a strictly typed VI reference only from a VI or VI template, not from a polymorphic VI or other non-VI file such as a global variable or custom control.

Use a strictly typed VI reference if you want to call the referenced VI with the Call By Reference Node. When you create a strictly typed VI reference, you cannot wire vi reference to the Run VI method. You cannot use the Run VI method to run a VI that is already reserved for execution by another VI. A strictly typed static VI reference also reserves any subVIs when a top-level VI is reserved, thus making it ineligible for the Run VI method. Refer to the *Run VI Method* topic of the *LabVIEW Help* for more information.

LabVIEW loads the referenced VI into memory when you load the top-level VI. When the Static VI Reference function outputs a strictly typed VI reference, LabVIEW reserves the referenced VI as long as the top-level VI is running. LabVIEW closes this reference when the top-level VI is no longer in memory.

Properties of the Application or VI Classes

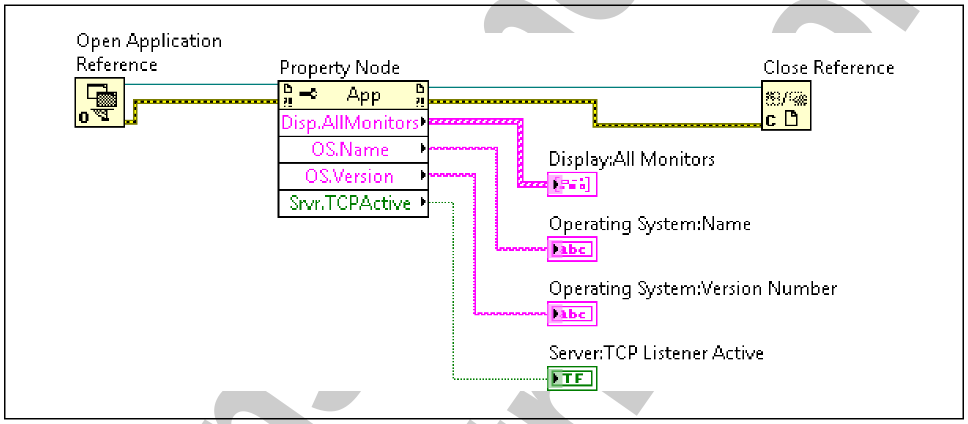
Use Property Nodes to modify properties of the LabVIEW application instance or VI object defined with the Open Application Reference or Open VI Reference function. There are different properties available for the Application class and the VI class.

Application Class Properties

Most Application class properties are read-only. They allow you to check a whole range of parameters, such as what VIs are loaded into memory, the operating system, and so on. For more information, refer to the *Application Properties* topic of the *LabVIEW Help*.

The following block diagram opens an application reference to the calling LabVIEW application instance and reads four properties of the LabVIEW application instance:

- The display properties of all the display monitors
- The name of the operating system, such as Windows 2000, Windows XP, and so on
- The version of the operating system
- Whether the LabVIEW VI Server is active



You can use this information to ensure that the computer running LabVIEW is configured correctly and prompt the user to reconfigure the computer if necessary.

VI Class Properties

Many of the properties of the VI class correspond to the properties available in the **VI Properties** dialog box. You can access the **VI Properties** dialog box by selecting **File»VI Properties**.

Tip After you select a property, you can access its help topic by right-clicking the property and selecting **Help for *Property Name*** from the shortcut menu, where ***Property Name*** is the name of the property.

Methods of the Application or VI Classes

Invoke Nodes are used to perform functions on the LabVIEW application instance or VI object defined with the Open Application Reference or Open VI Reference function. There are different methods available for the Application class and the VI class.

Application Class Methods

When you place the Invoke Node on the block diagram you can access the Application class methods. You do not have to wire an Application reference to the Invoke Node to access the Application class methods because the node defaults to the current Application if **reference** is unwired.

Some of the important methods available are Mass Compile, Get VI Version, and Bring To Front. The Mass Compile method loads and compiles all the VIs in a directory and all its subdirectories. The Get VI Version method gets the version of LabVIEW in which the VI was last saved. The Bring To Front method brings the application window to the front.

Refer to the *Application Methods* topic of the *LabVIEW Help* for more information.

VI Class Methods

When you wire the VI reference to the Invoke Node, you can access the VI class methods. If you do not wire a VI reference in, it will refer to the VI that called the method.

Some of the important methods available to the VI Server are Export VI Strings, Set Lock State, Run VI, and Save Instrument. The Export VI Strings method exports strings pertaining to VI and front panel objects to a tagged text file. The Set Lock State method sets the lock state of a VI. The Run VI method starts VI execution. The Save:Instrument method saves a VI.

Tip After you select a method, you can access its help topic by right-clicking the method and selecting **Help for *Method Name*** from the shortcut menu, where ***Method Name*** is the name of the method.

VI Server Configuration

To configure the VI Server, select **Tools»Options** on the server computer and select **VI Server** from the **Category** list to display the **VI Server** page. If a target in a LabVIEW project supports the VI Server, you also can right-click the target, such as **My Computer**, select **Properties** from the shortcut menu, and select **VI Server** from the **Category** list to display this page.

Use this page to configure the VI Server. If you are using a project, display this page from the **Options** dialog box to configure the VI Server for the main application instance. To configure the VI Server for a target, display this page from the **Properties** dialog box for the target.

The default VI Server settings are **ActiveX** enabled and **TCP/IP** disabled.

This page includes the following components.

- **Protocols**—Sets the protocols for the VI Server.
  - **TCP/IP**—Enables VI Server support for TCP/IP. If you allow remote applications to connect using TCP/IP, you also should specify which machine addresses can access the VI Server on the **Machine Access** section. This checkbox is unchecked by default.
    - **Port**—Sets the TCP/IP port at which the VI Server listens for requests. By default, this port number is 3363, which is a registered port number reserved for use by LabVIEW. For targets, the default is 0, causing the operating system to dynamically select a port. If you want to run multiple application instances on the machine, each with its own VI Server running, you must have a unique VI Server port number. You also can use the **Server:Port** property to set the LabVIEW VI Server port programmatically.

Note The VI Server settings in the **Options** dialog box apply to the main application instance, or VIs not in a project. To set VI Server settings for a project application instance, right-click the target in the **Project Explorer** window.

- **Service name**—Sets the service name for the VI Server TCP Instance. To retrieve an application reference without the port number, use service name in conjunction with the **Open Application Reference** function by wiring a service name to the polymorphic port number or Service name input. If you display this page from the **Options** dialog box, this service name is `Main Application Instance/VI Server` by default. If you display this page from the **Properties** dialog box for a target, the service name is `target name/VI Server` by default. You can use the **Server:Service Name** property to set the service name programmatically.
  - **Use default**—Sets **Service name** to its default value. This checkbox is checked by default. To edit **Service name**, remove the checkmark from the checkbox.

- **ActiveX**—**(Windows)** Enables VI Server support for ActiveX Automation. This checkbox is only available from the **Tools» Options** navigation. This checkbox is checked by default.

- **VI Scripting**—Use this section to enable VI Scripting.
  - **Show VI Scripting functions, properties and methods**—Enables VI Scripting functions on the VI Scripting palette and additional VI Server properties and methods. All functions, properties, and methods you enable through VI Scripting display as blue.

  - **Display additional VI Scripting information in Context Help window**—Displays connector pane terminal numbers in the Context Help window. Place a checkmark in the Show VI Scripting functions, properties and methods checkbox to enable this option.

- **Accessible Server Resources**—Indicates the tasks that remote applications can accomplish.
  - **VI calls**—Allows remote applications to call VIs exported through the VI Server. If you allow remote applications access to VIs, specify which VIs can be exported. This checkbox is checked by default.
  - **VI properties and methods**—Allows remote applications to read and set the properties of VIs through the VI Server. If you allow remote applications access to VIs, specify which VIs can be exported. This checkbox is checked by default.
  - **Application methods and properties**—Allows remote applications to read and set the properties of the VI Server. This checkbox is checked by default.
  - **Control methods and properties**—Allows remote applications to read and set the properties of controls in exported VIs. You also can call methods that are usable with controls. This checkbox is checked by default.
- **Machine Access Configuration**—Use this section to control machine access to VIs through the VI Server.

  When you allow remote applications to access the VI Server using the TCP/IP protocol, you should specify which Internet hosts have access to the server.

  Use this section to control machine access through the VI Server. To control the machine access for the main application instance, display this section from the **Options** dialog box. To control machine access for a target, display this page from the **Properties** dialog box for the target.

  When a remote LabVIEW application attempts to open a connection to the VI Server, the VI Server compares the IP address to the entries in the **Machine access list** to determine whether it should grant access. If an entry in the **Machine access list** matches the IP address, the VI Server permits or denies access based on how you set up the entry. If no entry matches the IP address, the VI Server denies access. When you add new **Machine access list** entries, edit existing entries, or remove entries, use the correct syntax, order, and wildcards in the entries.

  An IP address, such as `130.164.15.138`, might have more than one domain name associated with it. The conversion from a domain name to its corresponding IP address is called name resolution. The conversion from an IP address to its domain name is called name lookup. A name lookup or a resolution can fail when the system does not have access to a DNS (Domain Name System) server or when the address or name is not valid.

- **Machine access list**—Lists machines that do and do not have access to the VI Server. You also can use the **Server:TCP/IP Access List** property to list programmatically the TCP/IP addresses of machines that may access the VI server.

Note If you change the **Machine access list**, machines that are currently connected to the VI Server will not be disconnected even if they are no longer allowed access to the server.

- **Machine name/address**—Enter the name or IP address of the machine you want to add to the **Machine access list**.
- **Allow Access**—Allows access to the machine(s) selected in **Machine access list**.
- **Deny Access**—Denies access to the machine(s) selected in **Machine access list**.
- **Add**—Adds a new entry to the **Machine access list**. The new entry appears below the selected entry in the **Machine access list**.
- **Remove**—Removes the selected entry from the **Machine access list**.

• **Exported VIs**—Use this section to add, edit, and remove VIs from the Exported VIs list.

When you allow remote applications to access VIs through the VI Server, you should specify which VIs these applications can access.

- **Exported VIs list**—Lists the VIs that can be exported. You also can use the Server:VI Access List property to list programmatically the VIs on the VI Server that are accessible by remote clients.
- **Exported VI**—Enter a VI to list in **Exported VIs**. You can use wildcards in the VI name or directory path you enter.
- **Allow Access**—Allows access to the VI(s) selected in **Exported VIs**. This option is selected by default.
- **Deny Access**—Denies access to the VI(s) selected in **Exported VIs**.
- **Add**—Adds a new entry to **Exported VIs**.
- **Remove**—Removes the selected entry from **Exported VIs**.

If an entry is allowed access to VIs, a checkmark appears next to the entry. If an entry is denied access to VIs, an `X` appears next to the entry. If no symbol appears next to the entry, the syntax of the entry is incorrect.

Each entry in the **Exported VIs** list describes a VI name or a VI path and can contain wildcard characters. When a remote client tries to access a VI, the VI Server examines the **Exported VIs** list to determine whether to grant access to the requested VI. If an entry in the list matches the requested VI, the server allows or denies access to that VI based on how you set up that entry. If a subsequent entry also matches the VI, its access permission is used in place of the previous permission. If there is not a VI in the list that matches the requested VI, access to the VI is denied.

You can use the ?, *, and ** characters as wildcard characters. The ? and * wildcards do not include the path separator. ** includes the path separator.

To practice the concepts in this section, complete Exercise 2-1.

To practice the concepts in this section, complete Exercise 2-2.

D. Remote Communication

An important aspect of both Application and VI refnums is their network transparency. This means you can open refnums to objects on remote computers in the same way you open refnums to those objects on your computer.

After you open a refnum to a remote object, you can treat it in exactly the same way as a local object, with a few restrictions. For operations on a remote object, the VI Server sends the information about the operation across the network and retrieves the results. The application looks almost identical regardless of whether the operation is remote or local.

To open an application reference to a remote version of LabVIEW, you must specify the **machine name** input of the Open Application Reference function. Then LabVIEW attempts to establish a TCP connection with a remote VI Server on that computer on the specified port.

Example—Accessing a Remote VI Property

Figure 2-3 shows how to access the properties of a VI object on a remote computer. The Open Application Reference function opens a connection to a remote computer. The Open VI Reference function opens a VI on the specified remote computer. The Property Node opens the front panel of the specified VI. The Invoke Node runs the specified VI and waits for execution to complete before exiting the function. Because the Auto Dispose Ref is set to FALSE, a Close Reference function must be used to free the reference.

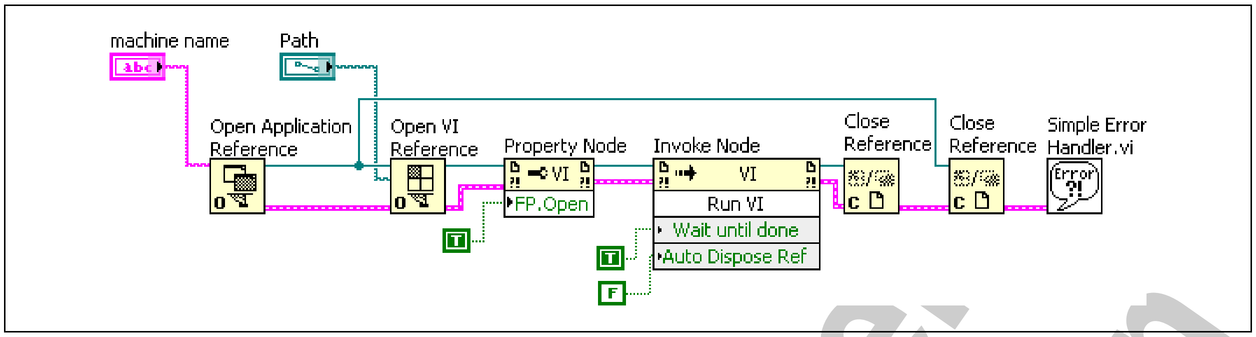


Figure 2-3. Accessing a Remote VI Property

Open VI Reference Inputs

VI path accepts a string containing the name of the VI you are referencing or a path containing the path to the VI you are referencing. If the VI is not in memory, the VI must be at the specified path for this function to succeed. If a path is wired, the function loads the specified VI into memory, if it is not already in memory, and returns a reference to this VI. If you wire a name string, the VI must already be in memory. If you wire a path and a VI of the same name is already in memory, the function returns a reference to the open VI, regardless of whether its path is the same as the input.

If the path is relative, the VI interprets the path as relative to the caller VI or to the application directory, if the caller VI is not saved.

application reference is a reference to a LabVIEW application instance. The default is a reference to an application instance on the local instance of LabVIEW. If wired and the reference is to a remote instance of LabVIEW, the remote instance of LabVIEW is queried to return the VI reference.

Refer to the *Open VI Reference Function* topic of the *LabVIEW Help* for more information about using the Open VI Reference function.

To practice the concepts in this section, complete Exercise 2-3.

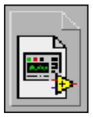
E. Dynamically Calling and Loading VIs

You can dynamically load VIs instead of using statically linked subVI calls. A statically linked subVI is one you place directly on the block diagram of a caller VI. It loads at the same time the caller VI loads.

Unlike statically linked subVIs, dynamically loaded VIs do not load until the caller VI loads them with the Open VI Reference. If you have a large caller VI, you can save load time and memory by dynamically loading the VI because the VI does not load until the caller VI needs it, and you can release it from memory after the operation completes. You also can use the **VI Call Configuration** dialog box to configure when to load the subVI.

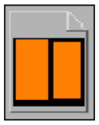
You use two types of VI refnums to dynamically load and call VIs in LabVIEW—strictly typed VI references and weakly typed VI references.

Types of VI References



- **Weakly Typed VI Reference**—Refers to a VI in memory. The type of the VI reference is not specified. If you create an indicator from the **vi reference** output of the Open VI Reference function, it looks like the control shown at left if it is weakly typed.

  Use a weakly typed VI reference to pass a VI reference from an Open VI Reference function or an Open Application Reference function to a Property Node or an Invoke Node. Create a weakly typed VI reference by not specifying the **type specifier VI Refnum** input.



- **Strictly Typed VI Reference**—Includes the connector pane information of the VI to be called. An indicator created from the **vi reference** output of the Open VI Reference function looks similar to the control shown at left if it is strictly typed.

  Use a strictly typed VI reference to dynamically call or load a VI. Wire a strictly typed VI refnum to the **type specifier** input of the Open VI Reference function to use the **vi reference** output with the Call By Reference Node. Strictly typed means that the connector terminals of a called VI and the data type they can accept are fixed. When you use the Open VI Reference function, it checks to see if the VI specified by vi path has the same connector pane and data type as defined in the **type specifier vi refnum** input. If it does not, the function returns an error.

Note The VI you use to specify the type of a strictly typed refnum provides only the connector pane information. That is, no permanent association is made between the refnum and the VI. In particular, avoid confusing selecting the VI connector pane with opening a reference to the selected VI. You specify a particular VI using the **vi path** input on the Open VI Reference function.

Dynamically Calling VIs with the Call By Reference Node

Use the Call By Reference Node to dynamically call VIs.

The Call By Reference Node requires a strictly typed VI refnum. The strictly typed VI refnum identifies the connector pane of the VI you are calling. You can wire the Call By Reference Node inputs and outputs just like you wire any other VI.

Strictly typed VI refnums allow you to save load time and memory because the subVI does not load until the caller VI needs it and you can release the subVI from memory after the operation completes.

Figure 2-4 shows how to use the Call By Reference Node to dynamically call the Frequency Response VI. The Call By Reference Node requires the use of the Open VI Reference and Close Reference functions, similar to the functions you use for the Property Node and the Invoke Node.

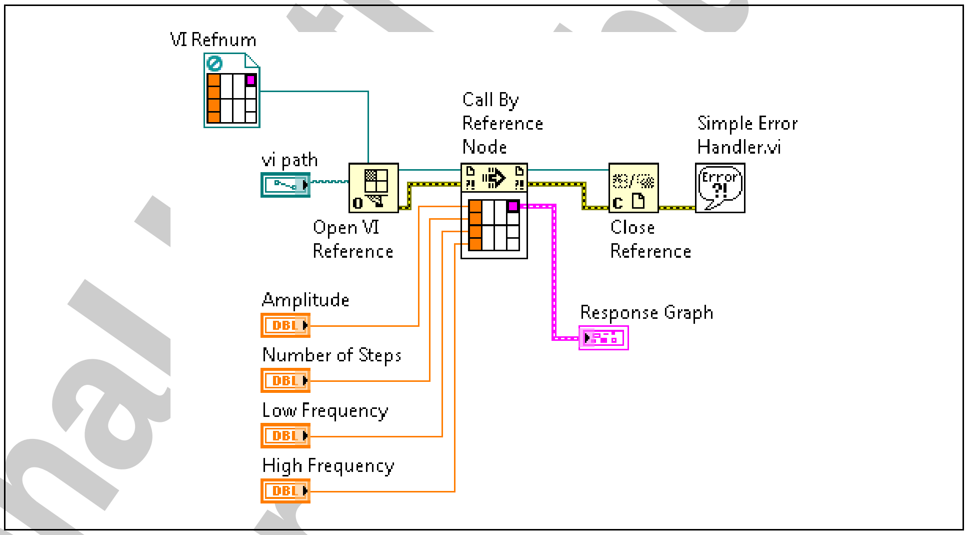


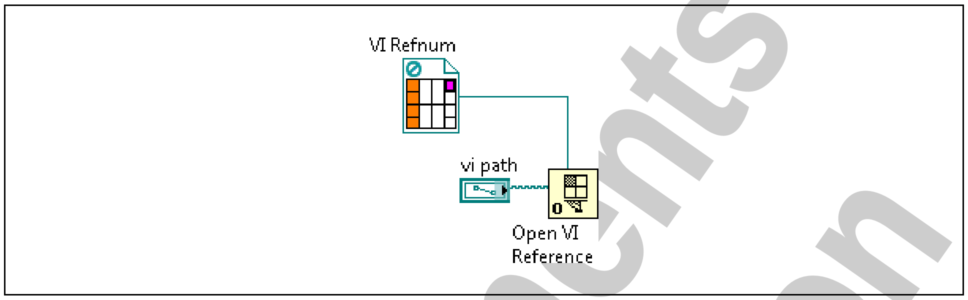
Figure 2-4. Call by Reference Node Dynamically Calls VIs

Call By Reference Programming Model

Complete the following steps to call a VI dynamically using the VI Server.

1. Use the Open VI Reference function to specify the VI you want to call. The Open VI Reference function needs a strictly typed refnum in order to call the VI dynamically. To create a strictly typed refnum, right-click the type specifier terminal of the Open VI Reference function and select **Create»Constant**.

   Right-click the refnum and select **Select VI Server Class»Browse** from the shortcut menu. Then **Choose the VI to Open** dialog box prompts you to select a VI. Select the VI that you want to replicate. Wire the strictly typed VI refnum to the **type specifier VI Refnum** input of the Open VI Reference function, as shown in the following figure.



Tip You also can create a strictly typed refnum by dragging and dropping a VI icon onto the refnum.

2. Use the Call by Reference Node to dynamically call the VI, as shown in the following figure. The Call By Reference Node works in the same way as calling a subVI. The only difference is that normally LabVIEW loads subVIs into memory when the application first opens, whereas with a Call by Reference Node, LabVIEW loads a VI into memory when the Open VI Reference function generates a reference to it.

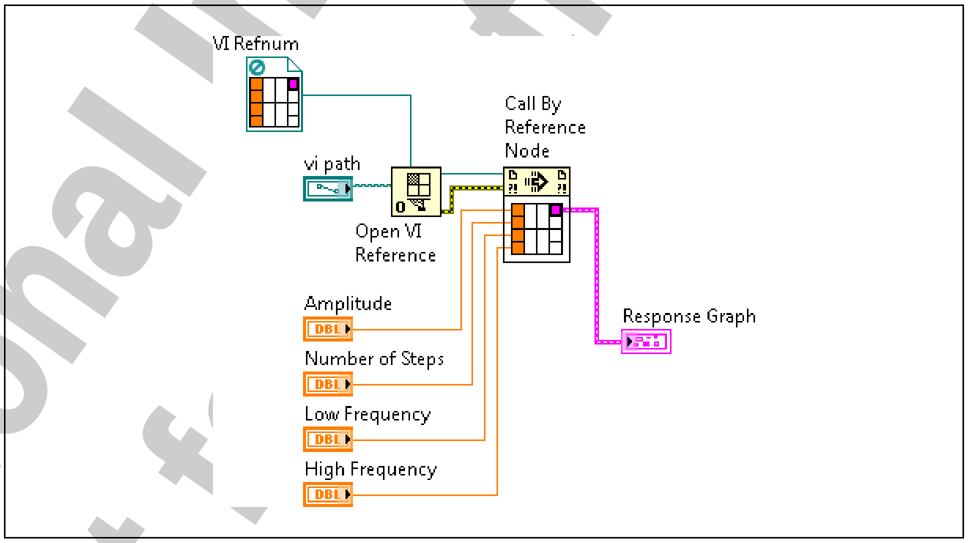


Figure 2-5. Use Call by Reference Node to Dynamically Call the VI

A strictly typed refnum stores the connector pane information of the VI to which it is strictly typed. The type specifier displays its connector pane. Notice that you are opening a reference to a VI that has a connector pane of the type you have just selected. It does not store any link to the VI you select.

3. Use the Close Reference function to close the reference to the VI and add a Simple Error Handler VI, as shown in the following figure.

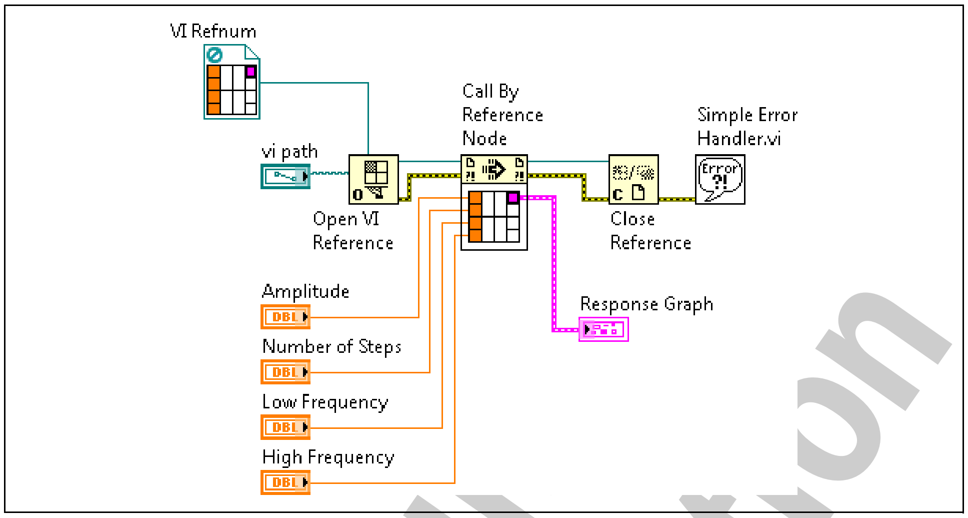


Figure 2-6. Use the Close Reference function to Close the Reference to the VI

Using Weakly Typed VI References

You can use methods in LabVIEW to modify the values of controls in a VI and read the values of indicators using a weakly typed VI refnum, as shown in Figure 2-7.

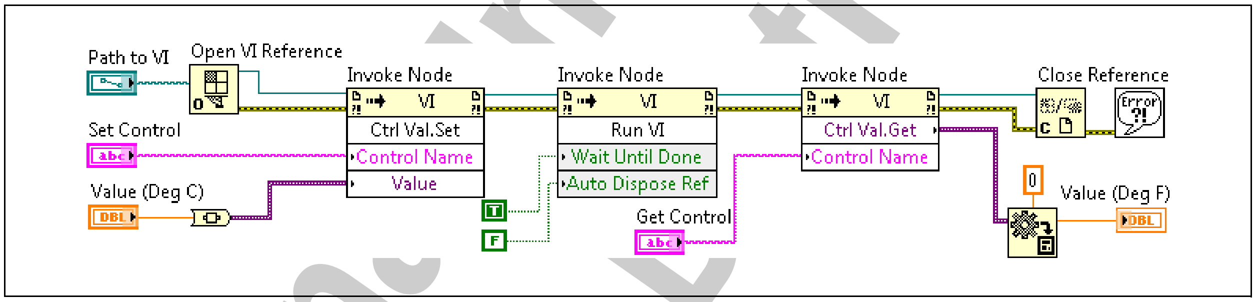


Figure 2-7. Use Methods to Modify and Read Values

Using a Call by Reference Node and a strictly typed VI refnum instead, as shown in Figure 2-8, you can write or read data to a VI in a much simpler manner.

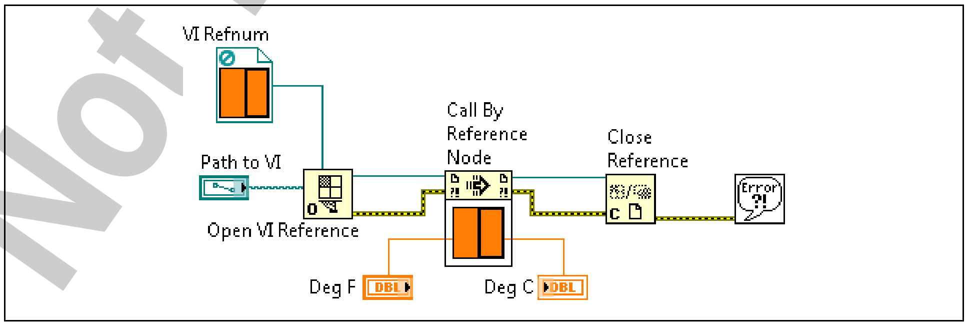


Figure 2-8. Use Call By Reference Node to Write or Read Data

Behavior of Strictly Typed VI References

When you open a strictly typed reference, the referenced VI is reserved for running and cannot be edited.

For example, you can open a VI reference to a target VI and edit the VI with Property and Invoke Nodes. While this reference is open, you can open another reference—such as a strictly typed reference—and call the target VI as a subVI through the Call By Reference Node. However, until you close the strictly typed reference, editing operations through the Property and Invoke Nodes fail because the VI to which they refer is reserved for running by the strictly typed reference.

Because opening a strictly typed VI reference puts the referenced VI in the reserved for running state, it means that the VI has been checked to make sure it is not corrupted, that it is not currently running as a top-level VI, that it has been compiled (if necessary), and a few other checks. A VI referenced by a strictly typed VI reference can be called using the Call By Reference Node at any moment without having to check all these conditions again. Thus, in the reserved for running state you cannot edit the VI or do anything to it that would change the way it would execute.

To practice the concepts in this section, complete Exercise 2-4.

Self-Review: Quiz

1. Which of the following does VI Server allow you to do?
   a. Programmatically access and control VIs and LabVIEW
   b. Call a VI remotely
   c. Load VIs into memory dynamically
   d. Call a shared library

2. Which of the following VIs are necessary for accessing a remote VI?
   a. Open Application Reference
   b. Open VI Reference
   c. Call By Reference Node

3. True or False? Dynamically loaded subVIs do not load until the caller VI makes the call to the subVI.

Self-Review: Quiz Answers

1. Which of the following does VI Server allow you to do?
   a. **Programmatically access and control VIs and LabVIEW**
   b. **Call a VI remotely**
   c. **Load VIs into memory dynamically**
   d. Call a shared library

2. Which of the following VIs are necessary for accessing a remote VI?
   a. **Open Application Reference**
   b. **Open VI Reference**
   c. Call By Reference Node

3. True or False? Dynamically loaded subVIs do not load until the caller VI makes the call to the subVI.

   True. Dynamically loaded subVIs do not load until the caller VI makes the call to the subVI.

Notes

3

Using .NET and ActiveX Objects in LabVIEW

LabVIEW provides access to other Windows applications using .NET or ActiveX technologies.

.NET refers to Microsoft's .NET technology. The .NET Framework is the programming basis of the .NET environment you use to build, deploy, and run Web-based applications, smart client applications, and XML Web services. You must install the .NET Framework. Refer to the Microsoft Developer Network (MSDN) Web site for more information about .NET and installing the framework.

Note Creating and communicating with .NET objects requires the .NET Framework 2.0 or later. National Instruments strongly recommends that you always put VIs that use .NET objects in a LabVIEW project and not leave them as stand-alone VIs.

ActiveX refers to Microsoft's ActiveX technology and OLE technology. With ActiveX Automation, a Windows application, such as LabVIEW, provides a public set of objects, commands, and functions that other Windows applications can access. Refer to the MSDN documentation for more information about ActiveX.

1. *Inside OLE*, by Kraig Brockschmidt, second edition
2. *Essential COM*, by Don Box

Topics

A. Using .NET Objects in LabVIEW

The .NET Framework allows you to access Windows services such as the performance monitor, event log, and file system, as well as advanced Windows APIs such as the Speech Recognition and Generation service. The .NET Framework also provides access to Web protocols such as SOAP, WSDL, and UDDI.

You can use LabVIEW as a .NET client to access the objects, properties, methods, and events associated with .NET servers. You also can host .NET user interface controls on the front panel of a VI. Although LabVIEW is not a .NET server, you can communicate to LabVIEW remotely with the ActiveX server interface through .NET support for COM objects.

Note Creating and communicating with .NET objects requires the .NET Framework 2.0 or later.

Be sure to save .NET assemblies in appropriate locations to ensure that they load and behave as expected.

.NET Functions and Nodes

Use the following LabVIEW VIs, functions, and nodes located on the **.NET** palette to access the objects, properties, and methods associated with .NET servers:

- **Constructor Node**—selects a constructor of a .NET class from an assembly and creates an instance of that class to be used during VI execution. When you place this node on the block diagram, LabVIEW displays the Select .NET Constructor dialog box.
- **Property Node (.NET)**—gets (reads) and sets (writes) the properties associated with a .NET class. Many .NET classes have an EnableEvents property. If you are not receiving events, you might need to set an EnableEvents property.
- **Invoke Node (.NET)**—invokes methods associated with a .NET class.
- **Close Reference function**—closes all references to .NET objects when you no longer need the reference.
- **Register Event Callback function**—registers and handle .NET events.
- **Unregister For Events function**—unregisters a .NET event.
- **To More Generic Class function**—upcasts a .NET reference to a base class.
- **To More Specific Class function**—downcasts a .NET reference to one of a derived class.
- **To .NET Object VI**—converts a LabVIEW type to a .NET object.

- **.NET Object To Variant VI—**converts a .NET object to a LabVIEW variant.
- **Static VI Reference function—**creates a strictly typed VI reference to a callback VI you create.

B. Accessing .NET in LabVIEW

You can use .NET in LabVIEW in the following ways:

- Creating .NET objects
- Setting .NET properties and invoking .NET methods
- Registering .NET events
- Loading .NET assemblies
- Configuring a .NET client application

Creating .NET Objects

You can create .NET objects on the front panel or the block diagram. Use a Constructor Node to create a .NET object on the block diagram. Use a .NET container to create a .NET control on the front panel. You also can add .NET controls to the Controls palette for later use.

.NET objects can be visible or invisible to the user. For example, buttons, windows, pictures, documents, and dialog boxes are visible to users. Application objects are invisible to users.

Use the .NET functions to access the .NET objects and their associated properties and methods.

Setting.NET Properties and Invoking .NET Methods

.NET objects expose properties and methods that other applications can access. You access an application by accessing an object associated with that application and setting a property or invoking a method of that object.On the front panel or block diagram, right-click a .NET object and select **Create»Property Node** or **Create»Invoke Node**, and select a property or method from the shortcut menu to set a property or invoke a method for the object. You also can select .NET-specific properties or invoke .NET-specific methods for a .NET object. On the block diagram, right-click a .NET object and select **Create»Property Node** or **Create» Invoke Node**, and select a .NET-specific property or method from the shortcut menu.

Loading .NET Assemblies

If you reference a .NET object from the front panel or block diagram of a VI, ensure that LabVIEW can load the .NET assembly for that object. The Common Language Runtime (CLR) is responsible for locating .NET assemblies that you call. Refer to the Microsoft Developer Network (MSDN) Web site for more information about how the CLR locates assemblies. If the CLR cannot find the assembly, LabVIEW then searches for the assembly in the same manner it searches for missing VIs. LabVIEW searches for missing VIs in the directories you specify on the **Paths** page of the **Options** dialog box. If LabVIEW cannot find the .NET assembly for a .NET object referenced directly on the front panel or block diagram, LabVIEW generates a load-time error. If LabVIEW cannot load a dependent assembly needed during run-time, LabVIEW generates a run-time error.

The CLR uses the directory of the running executable as the default search path when it loads private .NET assemblies. If you reference a .NET object from a VI that does not belong to a LabVIEW project, the CLR considers `LabVIEW.exe` to be the running executable. The CLR therefore searches for private assemblies in the directory in which the `LabVIEW.exe` file is located. If you reference a .NET object from a VI that does belong to a LabVIEW project, the CLR considers the project to be the running executable. The CLR therefore searches for private assemblies in the project directory. If you reference a .NET assembly from a VI and the assembly does not belong to the .NET Framework, National Instruments strongly recommends that you store the VI in a project to avoid having to place files in the directory in which the `LabVIEW.exe` file is located.

If you call a .NET assembly from a VI that does not belong to a project, you technically can save the assembly in the same directory as its calling VI. LabVIEW searches certain VI directories, including the calling VI directory, for assemblies that the CLR cannot load by default. However, calling assemblies stored in this location can result in name conflicts and other unexpected .NET behavior. Therefore, National Instruments does not recommend that you save assemblies in this location.

Configuring a .NET Client Application

.NET provides administrative capability to an application through configuration files. For example, you can specify the Common Language Runtime (CLR) version of a .NET application in a configuration file. LabVIEW automatically loads the latest installed version of the CLR. If you develop a .NET application using an earlier version of .NET, it can work on machines with later versions of the CLR. However, if you develop the application with a later version of the CLR than is on the target machine, the application does not work. If you want to build an application using .NET 1.1 and you have both 1.1 and 2.0 on your machine, specify the CLR version with a configuration file.

You also can use configuration files for any of the following tasks:

- loading assemblies
- setting up remote .NET access
- setting logging options
- setting security

A configuration file contains XML content and typically has a `.config` extension. To configure a LabVIEW .NET client application, you can provide a configuration file for a saved project, shared library, or stand-alone application. Name the configuration file the name of the project, library, or application with a `.config` extension, for example, `MyApp.lvproj.config`, `MyApp.dll.config`, or `MyApp.exe.config`. Save the configuration file in the directory that contains the project, library, or application.

If you build a stand-alone application from a saved project with a configuration file, you must rename and save the configuration file in the directory that contains the stand-alone application. For example, if you build a stand-alone application from `foo.lvproj` that has the configuration file `foo.lvproj.config`, rename the file `foo.exe.config` and save it in the directory that contains `foo.exe`, the stand-alone application.

When working with .NET assemblies, you can use a `.config` file to store preconfiguration code before running the assembly or application. If you call an assembly from a VI within a project, library, or stand-alone application, the assembly references the corresponding configuration file. If you call an assembly from a VI that is not in a project, library, or stand-alone application, the assembly references the `LabVIEW.exe.config` configuration file.

Review the specific documentation for your .NET type to ensure you are using the `.config` file correctly.

To practice the concepts in this section, complete Exercise 3-1.

To practice the concepts in this section, complete Exercise 3-2.

C. Registering .NET Events

.NET events are the actions taken on a .NET object, such as clicking a mouse, pressing a key, or receiving notifications about things such as running out of memory or tasks completing. Whenever these actions occur to the object, the object sends both an event to alert the .NET container and the event-specific data. The .NET object defines the events available for an object.

To use .NET events in an application, you must register for the event and handle the event when it occurs. .NET event registration is similar to dynamic event registration. However, the architecture of a .NET event VI is different from the architecture of an event-handling VI. The following components make up a typical .NET event VI:

- .NET object for which you want to generate an event.
- Register Event Callback function to specify and register for the type of event you want to generate. The Register Event Callback function is a growable node capable of handling multiple events, similar to the Register For Events function.
- Callback VI that contains the code you write to handle the event you specify.

When you wire a .NET object to the Register Event Callback function and specify the event you want to generate for that object, you are registering the .NET object for that event. After you register for the event, create a callback VI that contains the code you write to handle the event. Different events might have different event data formats so changing the event after you create a callback VI might break wires on the block diagram. Select the event before you create the callback VI.

You can handle events on .NET controls in a container. For example, you can place a calendar control in a .NET container and specify that you want to handle a DoubleClick event for the items displayed in the calendar.

Handling .NET Events

A callback VI contains the code you write to handle an ActiveX or .NET event you specify. You must create a callback VI to handle events from ActiveX controls or .NET objects when the controls or objects generate the registered events. The callback VI runs when the event occurs. When you create a callback VI, LabVIEW creates a reentrant VI that you can open and edit to handle an event. A callback VI contains the following elements:

- **Event Common Data** contains the following elements:
  - **Event Source** is a numeric control that specifies the source of the event, such as LabVIEW, ActiveX, or .NET. A value of `1` indicates an ActiveX event. A value of `2` indicates a .NET event.
  - **Event Type** specifies which event occurred. This is an enumerated type for user interface events and a 32-bit unsigned integer type for ActiveX, .NET, and other event sources. For ActiveX events, the

event type represents the method code, or ID, for the event registered. You can ignore this element for .NET.

- **Time Stamp** is the time stamp in milliseconds that specifies when the event was generated.

- **Control Ref** is a reference to the ActiveX or automation refnum or .NET object on which the event occurred.
- **Event Data** is a cluster of the parameters specific to the event the callback VI handles. LabVIEW determines the appropriate **Event Data** when you select an event from the Register Event Callback function. If an event does not have any data associated with it, LabVIEW does not create this control in the callback VI.
- **Event Data Out** is a cluster of the modifiable parameters specific to the event the callback VI handles. This element is available only if the ActiveX or .NET event has output parameters.
- (Optional) **user parameter** is data that you want to pass to the callback VI when the ActiveX or .NET object generates the event.

Note You can use an existing VI as a callback VI as long as the connector pane of the VI you intend to use matches the connector pane of the event data. The callback VI must be reentrant, and the reference to the callback VI must be strictly typed.

To allow callback VIs to execute without interruption, LabVIEW delays the processing of operating system messages until any callback VIs stop execution or until you load a modal window. When LabVIEW delays the processing of operating system messages, you cannot interact with any LabVIEW front panels. A *modal* window is a type of window that remains active or remains on top of all other LabVIEW windows until you close the window or open another modal window. You cannot interact with other windows while a modal window is open. Most dialog boxes in LabVIEW are modal windows.

You cannot open a non-modal window from a LabVIEW callback VI nor a DLL while any other process is running. Refer to Calling Non-Modal Windows Programmatically for more information about calling a non-modal window from a callback VI or DLL.

To practice the concepts in this section, complete Exercise 3-3.

D. Using ActiveX Objects in LabVIEW

You can use LabVIEW as an ActiveX client to access the objects, properties, methods, and events associated with other ActiveX-enabled applications. LabVIEW also can act as an ActiveX server, so other applications can access LabVIEW objects, properties, and methods.

ActiveX Objects, Properties, Methods, and Events

ActiveX-enabled applications include objects that have exposed properties and methods that other applications can access. Objects can be visible to the users, such as buttons, windows, pictures, documents, and dialog boxes, or invisible to the user, such as application objects. You access an application by accessing an object associated with that application and setting a property or invoking a method of that object.

Events are the actions taken on an object, such as clicking a mouse, pressing a key, or receiving notifications about things such as running out of memory or tasks completing. Whenever these actions occur to the object, the object sends an event to alert the ActiveX container along with the event-specific data. The ActiveX object defines the events available for an object.

Data Type Mapping

LabVIEW converts the data types of some ActiveX properties and methods into LabVIEW data types so LabVIEW can read and interpret the data. The following table lists the ActiveX data types and the corresponding converted LabVIEW data types.

ActiveX Data Type	LabVIEW Data Type
char	I8 U8
short	I16 U16
long	I32 U32
hyper	I64 U64
float	SGL EXT
double	DBL
BSTR	abc ⊶

ActiveX VIs, Functions, Controls, and Indicators

Use the following VIs, functions, controls, and indicators to access the objects, properties, methods, and events associated with other ActiveX-enabled applications:

- **Automation Refnum control**—creates a reference to an ActiveX object. Right-click this control on the front panel to select an object from the type library you want to access.
- **Automation Open function**—opens an ActiveX object.
- **ActiveX container**—accesses and display an ActiveX object on the front panel. Right-click the container, select **Insert ActiveX Object** from the shortcut menu, and select the object you want to access.
- **Property Node**—gets (reads) and sets (writes) the properties associated with an ActiveX object.

- **Invoke Node**—invokes the methods associated with an ActiveX object.
- **Register Event Callback function**—handles events that occur on an ActiveX object.
- **Variant control and indicator**—passes data to or from ActiveX controls.

Tip Select **Tools».NET & ActiveX»Add ActiveX Controls to Palette** to add custom control files to the ActiveX Controls palette.

E. Using LabVIEW as an ActiveX Client

When LabVIEW accesses the objects associated with another ActiveX-enabled application, it is acting as an ActiveX client. You can use LabVIEW as an ActiveX client in the following ways:

- Open an application, such as Microsoft Excel, create a document, and add data to that document.
- Embed a document, such as a Microsoft Word document or an Excel spreadsheet, on the front panel in a container.
- Place a button or other object from another application, such as a **Help** button that calls the other application help file, on the front panel.
- Link to an ActiveX control you built with another application.

LabVIEW accesses an ActiveX object with the automation refnum control or the ActiveX container, both of which are front panel objects. Use the automation refnum control to select an ActiveX object. Use the ActiveX container to select a displayable ActiveX object, such as a button or document, and place it on the front panel. Both objects appear as automation refnum control terminals on the block diagram.

The implementation, or the object you select, is necessary when COM creates the implementation. However, COM can only return interfaces. When LabVIEW creates an implementation, LabVIEW matches the implementation to the correct interface. A single implementation can implement multiple interfaces. In that case, LabVIEW picks the default interface on the object and returns that interface. The Automation Open function returns the default interface and you can use Variant To Data to convert to a different interface. You cannot return the implementation, only the interface.

Note The block diagram displays the default interface wired to the Automation Open function, not the implementation class. If you open the Select Object From Type Library dialog box, LabVIEW selects the implementation class by default.

Figure 3-1 shows how LabVIEW, acting as an ActiveX client, interacts with server applications.

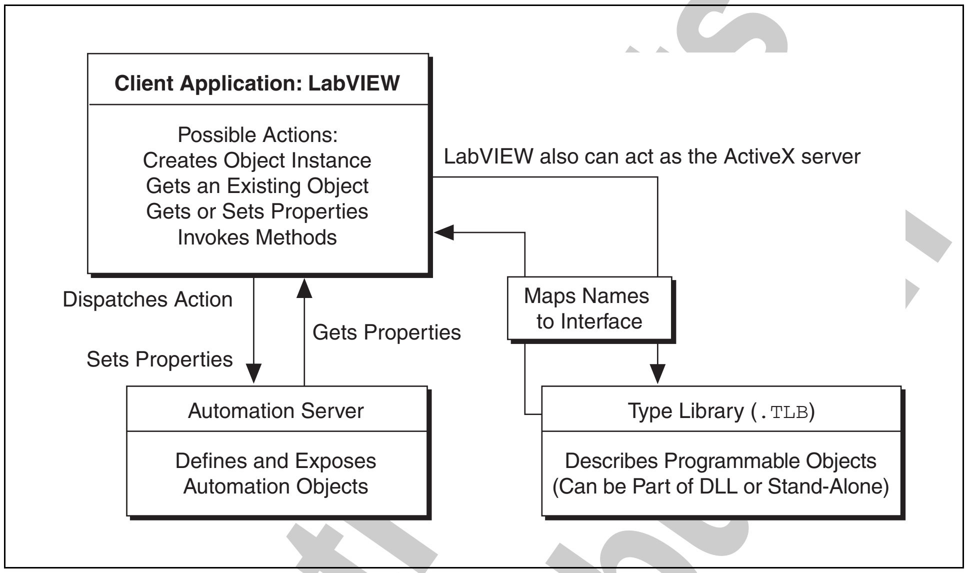


Figure 3-1. LabVIEW as an ActiveX Automation Client

LabVIEW accesses the server type library to obtain information about its objects, methods, and properties. LabVIEW can perform actions such as invoking methods, getting or setting properties, and so on.

Accessing an ActiveX-Enabled Application

To access an ActiveX-enabled application, use the automation refnum control terminal on the block diagram to create a reference to an application. Wire the control terminal to the Automation Open function, which opens the calling application. Use the Property Node to select and access the properties associated with the object. Use the Invoke Node to select and access the methods associated with the object. Close the reference to the object using the Close Reference function. Closing the reference removes the object from memory.

For example, you can build a VI that opens Microsoft Excel so it appears on the user's screen, creates a workbook, creates a spreadsheet, creates a table in LabVIEW, and writes that table to the Excel spreadsheet.

Refer to the Write Table To XL VI in the `labview\examples\comm\ExcelExamples.llb` for an example of using LabVIEW as an Excel client.

Inserting an ActiveX Object on the Front Panel

To insert an ActiveX object on the front panel, right-click the ActiveX container, select **Insert ActiveX Object** from the shortcut menu, and select the ActiveX control or document you want to insert. You can set the properties for an ActiveX object using the ActiveX Property Browser or property pages, or you can set the properties programmatically using the Property Node.

Use the Invoke Node to invoke the methods associated with the object.

For example, you can display a Web page on the front panel by using an ActiveX container to access the Microsoft Web Browser control, selecting the Navigate class of methods, selecting the URL method, and specifying the URL.

If you use the ActiveX container, you do not have to wire the automation refnum control on the block diagram to the Automation Open function or close the reference to the object using the Close Reference function. You can wire directly to the Invoke Node or Property Node because the ActiveX container embeds the calling application in LabVIEW. However, if the ActiveX container includes properties or methods that return other automation refnums, you must close these additional references.

Design Mode for ActiveX Objects

Right-click an ActiveX container and select **Advanced»Design Mode** from the shortcut menu to display the container in design mode while you edit the VI. When you select the design mode option, the ActiveX container owns the references to the controls and is responsible for creating and releasing new controls. There are distinct design and run modes for controls. Each time you run a VI, LabVIEW removes the design mode control and replaces it with a new control in run mode. When the VI stops, LabVIEW removes the run mode control and creates a new design mode control. In a built application there is no design mode control. LabVIEW creates the run mode control when the VI starts and releases the control when the VI stops. In design mode, events are not generated and event procedures do not run. The default mode is run mode, where you interact with the object as a user would.

Setting ActiveX Properties

After you open an ActiveX server or insert an ActiveX control or document, you can set the properties associated with that control or document using the ActiveX Property Browser, property pages, and the Property Node.

ActiveX Property Browser

Use the ActiveX Property Browser to view and set all the properties associated with an ActiveX control or document in an ActiveX container. To access the ActiveX Property Browser, right-click the control or document in the container on the front panel and select **Property Browser** from the shortcut menu. You also can select **View»ActiveX Property Browser**. The ActiveX Property Browser is an easy way to set the properties of an ActiveX object interactively, but you cannot use the browser to set properties programmatically, and you can use the ActiveX Property Browser only with ActiveX objects in a container. The ActiveX Property Browser is not available in run mode or while a VI runs.

ActiveX Property Pages

Many ActiveX objects include property pages, which organize the properties associated with the object on separate tabs. To access ActiveX property pages, right-click the object in the container on the front panel and select the name of the object from the shortcut menu.

Like the ActiveX Property Browser, ActiveX property pages are an easy way to set the properties of an ActiveX object interactively, but you cannot use them to set properties programmatically, and you can use property pages only with ActiveX objects in a container. Property pages are not available for all ActiveX objects. Property pages also are not available in run mode or when the VI is running.

Property Nodes

Use the Property Node to set ActiveX properties programmatically. For example, if you use an ActiveX object to warn a user when a temperature exceeds a limit, use the Property Node to set the Value property of the object to specify the limit.

The block diagram in Figure 3-2 shows an example that changes the Value property of the CWButton ActiveX control, which is part of the National Instruments Measurement Studio User Interface ActiveX Library, when the temperature reaches 85 degrees Fahrenheit or higher.

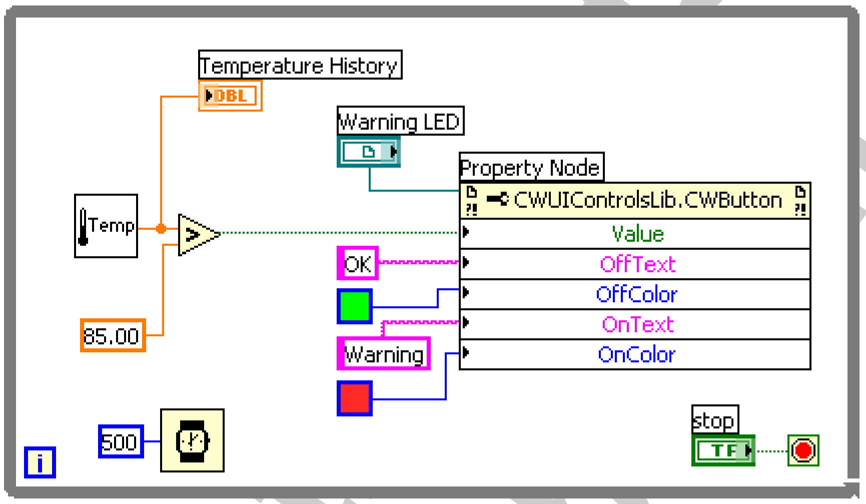


Figure 3-2. Using a Property Node to Set ActiveX Properties Programatically

In this case, the CWButton control acts as an LED, changes colors, and displays `Warning` when the temperature reaches the limit, which is the on state of the CWButton control.

Note In this example, you could use the ActiveX Property Browser or property pages to set the OffText, OffColor, OnText, and OnColor properties for the CWButton control because you do not need to set those properties programmatically.

To practice the concepts in this section, complete Exercise 3-4.

F. Using LabVIEW as an ActiveX Server

The LabVIEW application, VIs, and control properties and methods are available through ActiveX calls from other applications. Other ActiveX-enabled applications, such as Microsoft Excel, can request properties, methods, and individual VIs from LabVIEW, and LabVIEW acts as an ActiveX server.

For example, you can embed a VI graph in an Excel spreadsheet and, from the spreadsheet, enter data in the VI inputs and run the VI. When you run the VI, the data plot to the graph.

Refer to the `labview\examples\comm\freqresp.xls` for an example of using LabVIEW properties and methods in an Excel spreadsheet.

LabVIEW supports an ActiveX interface to the VI Server. This server interface allows client applications to programmatically access LabVIEW. Figure 3-3 shows how LabVIEW acts as an ActiveX server to interact with client applications.

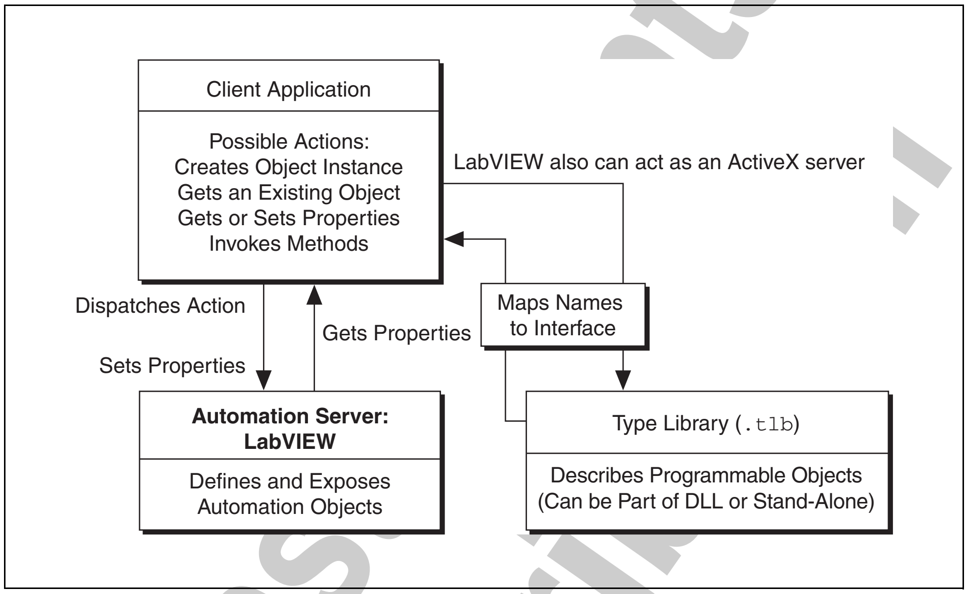


Figure 3-3. LabVIEW ActiveX Automation Server

The LabVIEW type library, `labview.tlb`, provides information about LabVIEW objects, methods, and properties. Client applications can access methods and set or get properties of the LabVIEW ActiveX server.

You can use LabVIEW as an ActiveX server, enabling other ActiveX-enabled clients to request properties and methods from LabVIEW and individual VIs. To configure LabVIEW as an ActiveX server, select **Tools»Options** to display the **Options** dialog box. Select **VI Server:Configuration** from the top pull-down menu and place a checkmark in the **ActiveX** checkbox.

Support for Custom ActiveX Automation Interfaces

If you are writing an ActiveX client that accesses properties and methods from an ActiveX server using LabVIEW, you can access custom interfaces exposed by the server. You do not need to use IDispatch to do so. However, the developer of the ActiveX server must make sure the parameters of the properties and methods in these custom interfaces have Automation (IDispatch) data types. The developer of the server must do so to expose multiple interfaces from one object, rather than through multiple objects. You still can use the interfaces in the LabVIEW environment. Refer to your server development programming environment documentation for more information about accessing custom interfaces.

G. ActiveX Events

To use ActiveX events in an application, you must register for the event and handle the event when it occurs. ActiveX event registration is similar to dynamic event registration. However, the architecture of an ActiveX event VI is different than the architecture of an event-handling VI. The following components make up a typical ActiveX event VI:

- ActiveX object for which you want to generate an event.
- Register Event Callback function to specify and register for the type of event you want to generate.
- Callback VI that contains the code you write to handle the event you specify.

You can generate and handle events on ActiveX objects in a container or on ActiveX objects you specify by using an automation refnum. For example, you can call a Windows tree control from an ActiveX container and specify that you want to generate a Double Click event for the items displayed in the tree control.

The Register Event Callback function is a growable node capable of handling multiple events, similar to the Register For Events function.

When you wire an ActiveX object reference to the Register Event Callback function and specify the event you want to generate for that object, you are registering the ActiveX object for that event. After you register for the event, create a callback VI that contains the code you write to handle the event. Different events may have different event data formats so changing the event after you create a callback VI might break wires on the block diagram. Select the event before you create the callback VI.

National Instruments recommends that you unregister for events when you no longer need to handle them using the Unregister For Events function. If you do not unregister for events, LabVIEW continues to generate and queue the events as long as the VI runs, even if no Event structure is waiting to handle them, which consumes memory and can hang the VI if you enable front panel locking for the events.

Handling ActiveX Events

A callback VI contains the code you write to handle an ActiveX or .NET event you specify. You must create a callback VI to handle events from ActiveX controls or .NET objects when the controls or objects generate the registered events. The callback VI runs when the event occurs. When you create a callback VI, LabVIEW creates a reentrant VI that you can open and edit to handle an event. A callback VI contains the following elements:

- **Event Common Data** contains the following elements:
  - **Event Source** is a numeric control that specifies the source of the event, such as LabVIEW, ActiveX, or .NET. A value of `1` indicates an ActiveX event. A value of `2` indicates a .NET event.
  - **Event Type** specifies which event occurred. This is an enumerated type for user interface events and a 32-bit unsigned integer type for ActiveX, .NET, and other event sources. For ActiveX events, the event type represents the method code, or ID, for the event registered. You can ignore this element for .NET.
  - **Time Stamp** is the time stamp in milliseconds that specifies when the event was generated.
- **Control Ref** is a reference to the ActiveX or automation refnum or .NET object on which the event occurred.
- **Event Data** is a cluster of the parameters specific to the event the callback VI handles. LabVIEW determines the appropriate **Event Data** when you select an event from the Register Event Callback function. If an event does not have any data associated with it, LabVIEW does not create this control in the callback VI.
- **Event Data Out** is a cluster of the modifiable parameters specific to the event the callback VI handles. This element is available only if the ActiveX or .NET event has output parameters.
- (Optional) **user parameter** is data that you want to pass to the callback VI when the ActiveX or .NET object generates the event.

Note You can use an existing VI as a callback VI as long as the connector pane of the VI you intend to use matches the connector pane of the event data. The callback VI must be reentrant, and the reference to the callback VI must be strictly typed.

To allow callback VIs to execute without interruption, LabVIEW delays the processing of operating system messages until any callback VIs stop execution or until you load a modal window. When LabVIEW delays the processing of operating system messages, you cannot interact with any LabVIEW front panels. A *modal* window is a type of window that remains active or remains on top of all other LabVIEW windows until you close the window or open another modal window. You cannot interact with other windows while a modal window is open. Most dialog boxes in LabVIEW are modal windows.

You cannot open a non-modal window from a LabVIEW callback VI nor a DLL while any other process is running. Refer to the *Calling Non-Modal Windows Programmatically* topic of the *LabVIEW Help* for more information about calling a non-modal window from a callback VI or DLL.

Self-Review: Quiz

1. For each of the following functions, identify if it is used in ActiveX applications, .NET applications or both.
   a. Constructor Node
   b. Invoke Node
   c. Property Node
   d. Automation Open
   e. Register Event Callback
   f. Close Reference

2. True or False? LabVIEW can act as an ActiveX client and/or an ActiveX server.

3. Which of the following are ways to call objects in LabVIEW?
   a. ActiveX Automation
   b. ActiveX Container
   c. User parameter
   d. Constructor Node
   e. .NET Container

Self-Review: Quiz Answers

1. For each of the following functions, identify if it is used in ActiveX applications, .NET applications or both.
   a. Constructor Node—**.NET**
   b. Invoke Node—**Both**
   c. Property Node—**Both**
   d. Automation Open—**ActiveX**
   e. Register Event Callback—**Both**
   f. Close Reference—**Both**

2. **True** or False? LabVIEW can act as an ActiveX client and/or an ActiveX server.

3. Which of the following are ways to call objects in LabVIEW?
   a. ActiveX Automation
   b. ActiveX Container
   c. User parameter
   d. Constructor Node
   e. .NET Container

Notes

4

Connecting to Databases

Databases are useful for storing large, complex systems of data and make enterprise connectivity to the data possible. This lesson defines database terminology and demonstrates the fundamentals of database programming in LabVIEW. The LabVIEW Database Connectivity Toolkit allows you to use LabVIEW to connect to a database, view its contents, insert data into it, and execute SQL statements on it.

Topics

A. What is a Database?

A database is an organized collection of data. A Database Management System (DBMS) is a group of software for organizing the information in a database. This software may include routines for data input, verification, storage, retrieval, and combination. Microsoft Access, Oracle, SQL Server, and MySQL are a few examples of DBMSs. Most modern DBMSs store data in tables. These tables are organized into records, also known as rows, and fields, also known as columns. Every table in a database must have a unique name and every field within a table must have a unique name.

Database tables have many uses. Table 4-1 is an example table that you could use with a simple test executive program to record test sequence results. It contains columns for the unit under test number, the test name, the test result, and two measurements. The data in the table is not inherently ordered. Ordering, grouping, and other manipulations of the data occur when you use a SELECT statement to retrieve the data from the table. A row can have empty columns, which means that the row contains NULL values. NULL values for databases are not exactly the same as NULL values in the C programming language. Refer to *Handling Null Values* in the *Database Connectivity Toolkit User Manual* for more information about how LabVIEW handles NULL values.

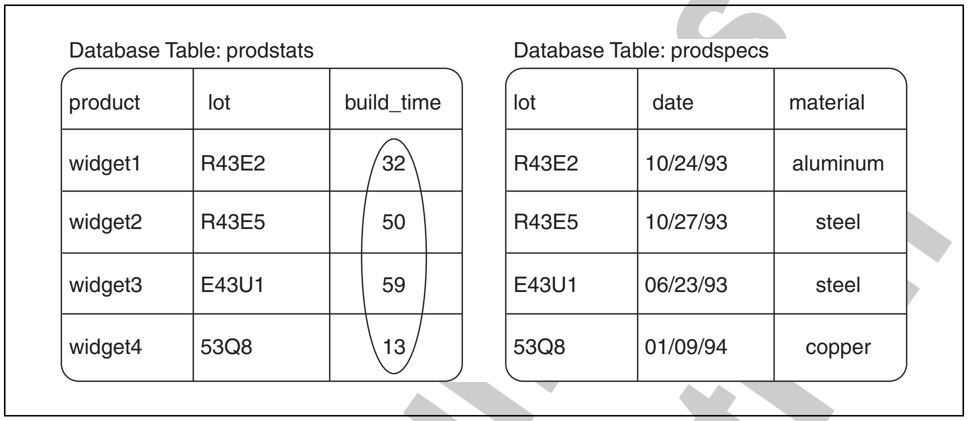
Note This manual refers to NULL values in tables as SQL NULL values, to distinguish them from standard NULL values.

Table 4-1. Sample Test Sequence Results

UUT_NUM	TEST_NAME	RESULT	MEAS1	MEAS2
20860B456	TEST1	PASS	0.5	0.6
20860B456	TEST2	PASS	1.2	—
20860B123	TEST1	FAIL	–0.1	0.7
20860B789	TEST1	PASS	0.6	0.6
20860B789	TEST2	PASS	1.3	—

Non-relational and Relational Databases

Non-relational databases allow you to store all of your information in one large table like Table 4-1. This method is sometimes inefficient, because all of the information is in one table, and searching for a specific piece of data can be difficult and time-consuming. Relational databases store information in multiple structures, or tables, where each table can be smaller and contain a specific subset of information. Figure 4-1 shows two database tables that are related to each other through the lot field.



Database Table: prodstats

product	lot	build_time
widget1	R43E2	32
widget2	R43E5	50
widget3	E43U1	59
widget4	53Q8	13

Database Table: prodspecs

lot	date	material
R43E2	10/24/93	aluminum
R43E5	10/27/93	steel
E43U1	06/23/93	steel
53Q8	01/09/94	copper

Figure 4-1. Relational Database Table Concept

Each column in a table has a data type, such as CHARACTER (fixed and variable length), NUMBER, DECIMAL, INTEGER, FLOAT, REAL, DOUBLE PRECISION,DATE, LONG, and RAW. The available data types vary depending on the DBMS. The LabVIEW Database Connectivity Toolkit uses a set of common data types. The toolkit automatically maps these data types into the appropriate type in the underlying database. By using the common data types, the toolkit can access a variety of databases with little or no modification.

B. Database Standards

The LabVIEW Database Connectivity Toolkit contains a set of VIs with which you can perform both common database tasks and more advanced and customized tasks.

The LabVIEW Database Connectivity Toolkit works with any database driver that complies with one of the following standards:

- Open Database Connectivity (ODBC)
- Object Linking Embedding Database (OLE DB)

ODBC Standard

The SQL Access Group, including representatives of Microsoft, Tandem, Oracle, Informix, and Digital Equipment Corporations, developed the Open Database Connectivity (ODBC) standard as a uniform method for applications to access databases. ODBC 1.0 released in September 1992. The standard consists of a multilevel API definition, a driver packaging standard, an SQL implementation based on ANSI SQL, and a means for defining and maintaining Data Source Names (DSN). A DSN is a quick way to refer to a specific database. You specify a DSN with a unique name and by the ODBC driver that communicates with the physical database, local or remote. You must define a DSN for each database to which an application program connects.

OLE DB Standard

The ODBC standards was designed to access only relational databases. Microsoft realized this as a limitation and developed a platform called Universal Data Access (UDA) where applications can exchange relational or non-relational data across intranets or the Internet, essentially connecting any type of data with any type of application. OLE DB is the Microsoft system-level programming interface to diverse sources of data. The Microsoft ActiveX Data Object (ADO) standard is the application-level programming interface.

The Microsoft Data Access Components (MDAC) are the practical implementation of the Microsoft UDA strategy. MDAC includes ODBC, OLE DB, and ADO components. MDAC also installs several data providers you can use to open a connection to a specific data source, such as an MS Access database.

OLE DB specifies a set of Microsoft Component Object Model (COM) interfaces that support various database management system services. These interfaces enable you to create software components that comprise the UDA platform. OLE DB is a C++ API that allows for lower-level database access from a C++ compiler. Three general types of COM components for OLE DB include:

- **OLE DB Data Providers**—Data source-specific software layers that access and expose data.
- **OLE DB Consumers**—Data-centric applications, components, or tools that use data through OLE DB interfaces. Using networking terms, OLE DB consumers are the clients, and the OLE DB data provider is the server.
- **OLE DB Service Providers**—Optional components that implement standard services to extend the functionality of data providers. Examples of these services include cursor engines, query processors, and data conversion engines.

All data access in the LabVIEW Database Connectivity Toolkit occurs through an OLE DB provider. If you do not specify a provider, the toolkit uses the OLE DB provider for ODBC provider. Microsoft provides some relational data providers as part of the MDAC installation.

To practice the concepts in this section, complete Exercise 4-1.

C. Connecting to a Database

Before you can access data in a table or execute SQL statements, you must establish a connection to a database. The LabVIEW Database Connectivity Toolkit supports multiple simultaneous connections to a single database or to multiple databases. Use the DB Tools Open Connection VI to establish the connection to a database.

Determining how to connect to a database can be a complex process because each DBMS uses different parameters for a connection and different levels of security. The different standards also use different methods of connecting to databases. For example, ODBC uses Data Source Names (DSN) for the connection, whereas the Microsoft ActiveX Data Object (ADO) standard uses Universal Data Links (UDL) for the connection. Fortunately, the DB Tools Open Connection VI supports all methods for connecting to a database.

DSNs and Data Source Types

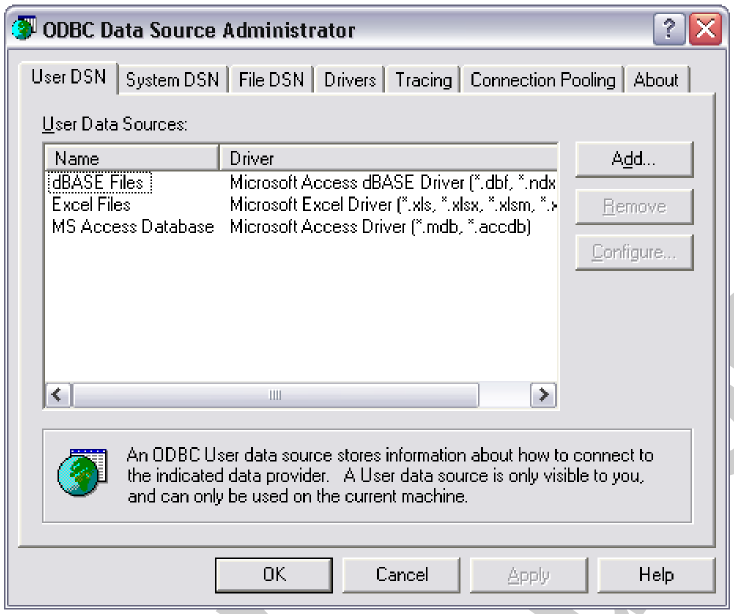
A DSN is the name of the data source, or database, to which you are connecting. The DSN also contains information about the ODBC driver and other connection attributes including paths, security information, and read-only status of the database. You can use the ODBC Data Source Administrator to create and configure DSNs. Two main types of DSNs exist: machine DSNs and file DSNs.

Machine DSNs are a restrictive type of DSN that reside in the system registry. Machine DSNs that apply to all users of a computer system are called system DSNs. These DSNs are restrictive because they are limited to only one system. Machine DSNs that apply to single users are called user DSNs. These DSNs are restrictive because they are limited to individual users.

The second main type of DSNs, File DSNs, are not restrictive. A file DSN is a text file with a `.dsn` extension and is accessible to anyone with proper permissions. File DSNs are not restricted to a single user or computer system.

ODBC Data Source Administrator

Use the ODBC Data Source Administrator to register and configure drivers to enable them as data sources for applications. In the Windows Control Panel, select **Administrative Tools»Data Sources (ODBC)** to display the ODBC Data Source Administrator as shown in Figure 4-2. The system saves the configuration for each data source in the registry or in a file.



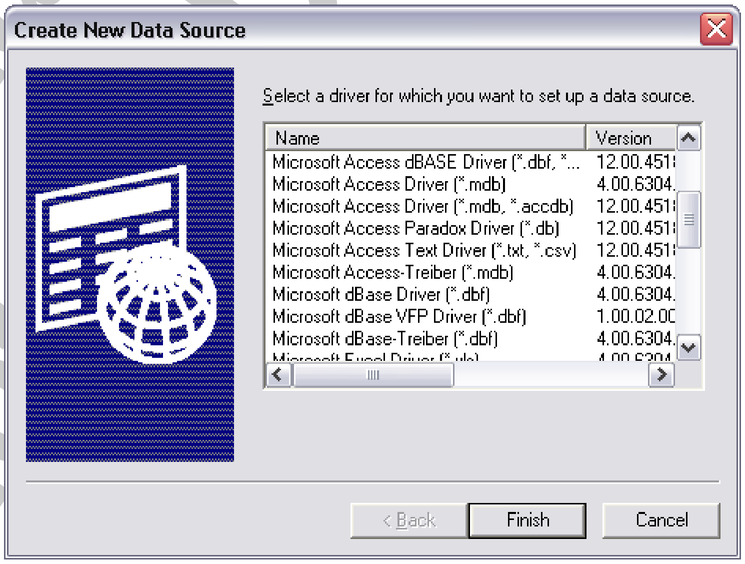
User Data Sources:

Name	Driver
dBASE Files	Microsoft Access dBASE Driver (*.dbf, *.ndx
Excel Files	Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.x
MS Access Database	Microsoft Access Driver (*.mdb, *.accdb)

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

Figure 4-2. Data Sources Dialog Box

The ODBC Data Source Administrator dialog box lists all the registered ODBC data sources. On various tabs you can select the type of DSN—User, System, or File. You then can click **Add** or **Configure** to display a driver-specific dialog box where you can configure a new or an existing data source. The system then saves the configuration for the data source in the registry or to a file. When you create a new DSN, the Create New Data Source dialog box displays a list of all the ODBC drivers for your system, as shown in Figure 4-3.



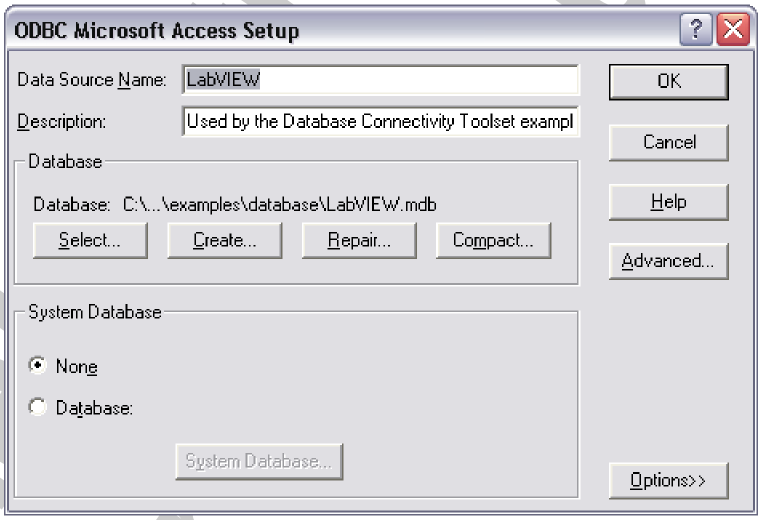
Select a driver for which you want to set up a data source.

Name	Version
Microsoft Access dBASE Driver (*.dbf, *...	12.00.451
Microsoft Access Driver (*.mdb)	4.00.6304.
Microsoft Access Driver (*.mdb, *.accdb)	12.00.451
Microsoft Access Paradox Driver (*.db)	12.00.451
Microsoft Access Text Driver (*.txt, *.csv)	12.00.451
Microsoft Access-Treiber (*.mdb)	4.00.6304.
Microsoft dBase Driver (*.dbf)	4.00.6304.
Microsoft dBase VFP Driver (*.dbf)	1.00.02.00
Microsoft dBase-Treiber (*.dbf)	4.00.6304.
Microsoft Excel Driver (*.xls)	4.00.6304

Figure 4-3. Available ODBC Drivers

The Database Connectivity Toolkit complies with the ODBC standard, so you can use the toolkit with any ODBC-compliant database drivers. The Database Connectivity Toolkit does not provide custom ODBC drivers. However, Microsoft Data Access Components (MDAC) includes several ODBC drivers. Database system vendors and third-party developers also offer large selections of ODBC drivers. Refer to the vendor documentation for information about registering the specific database drivers in the ODBC Data Sources Administrator.

After you select a particular driver, a second dialog box displays the specific settings for that driver. ODBC drivers for databases such as SQL Server and Oracle contain settings and additional dialog boxes for configuring items such as server information, user identification, and passwords. Figure 4-4 shows the ODBC Access Driver Setup dialog box for the system DSN named LabVIEW that is used for the Database Connectivity Toolkit examples in the Example Finder.



Data Source Name: LabVIEW
Description: Used by the Database Connectivity Toolset exampl
Database: C:\...\examples\database\LabVIEW.mdb

Figure 4-4. ODBC Access Driver Setup Dialog Box

Connecting to Databases Using DSNs

You can use the DB Tools Open Connection VI to connect to various databases that you specify with DSNs.

You can use a string to specify a system DSN or user DSN. The VI in Figure 4-5 specifies a DSN called `MS Access` to open a connection to that specific database.

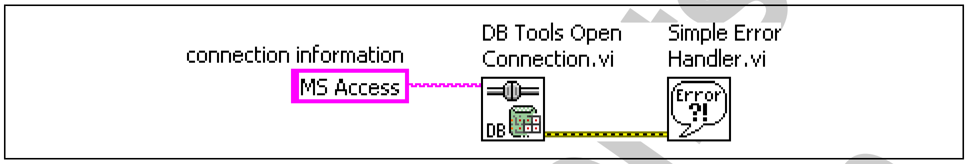


Figure 4-5. Connecting to an Access Database Using a System DSN

You can use a path to specify a file DSN. The VI in Figure 4-6 specifies a path to a file DSN called `access.dsn` to open a connection to the database.

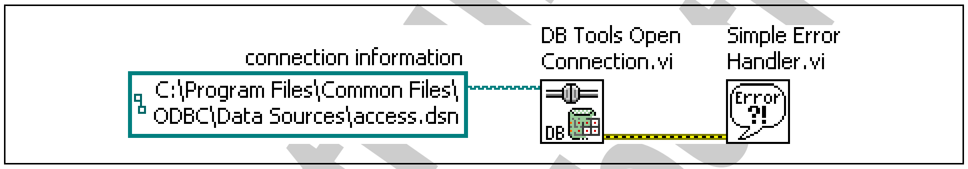

connection information
C:\Program Files\Common Files\ODBC\Data Sources\access.dsn
DB Tools Open Connection.vi
Simple Error Handler.vi

Figure 4-6. Connecting to an Access Database Using a File DSN

Notice that the connection information input of the DB Tools Open Connection VI is polymorphic. This VI accepts either a string or path for the DSN.

The VI in Figure 4-7 connects to an Oracle database using a system DSN. Notice that the **userID** and **password** parameters are wired. Some DBMS require that these parameters be set in order to connect to a database.

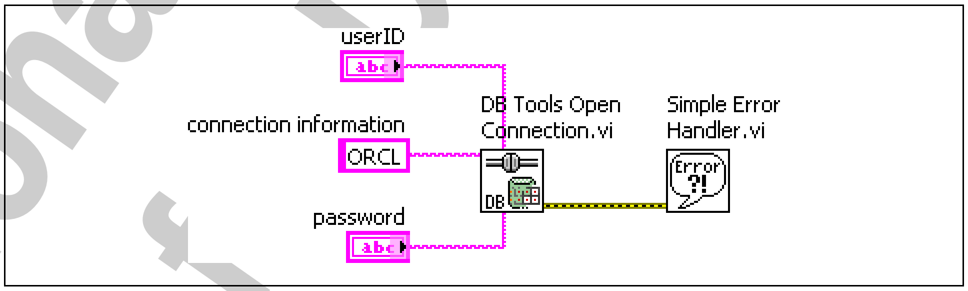


Figure 4-7. Connecting to an Oracle Database Using a System DSN

As shown in the previous examples, connecting to a database using the DB Tools Open Connection VI requires only a string or path value specifying the DSN along with optional user ID and password strings depending upon the DBMS. Therefore, the majority of problems in defining a connection occur when creating the DSN. Some ODBC drivers have an option to test the connection. Test the connection between the DSN and the database before you try to do anything with the Database Connectivity Toolkit.

UDLs

A UDL is similar to a DSN in that it describes more than just the data source. A UDL specifies what OLE DB provider is used, server information, the user ID and password, the default database, and other related information.

You can create a UDL in one of the following three ways:

- Use the **prompt?** input of the DB Tools Open Connection VI, as shown in Figure 4-8.

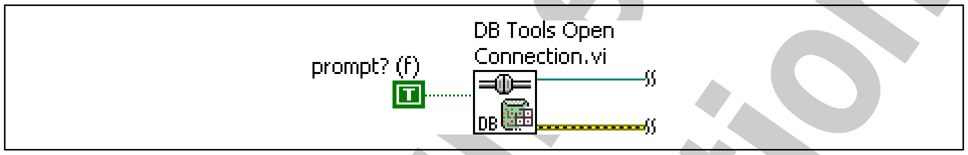


Figure 4-8. Using a Prompt to Create a UDL

  The **prompt?** input displays the **Data Link Properties** dialog box when the DB Tools Open Connection VI runs. You can select the appropriate options in this dialog box to make the database connection.

- Select **Tools»Create Data Link** in LabVIEW to display the **Data Link Properties** dialog box.

Note The Database Connectivity Toolkit installer creates a directory called `data links` inside the `labview/Database` directory. Save all UDL files and file DSNs to this directory so you can find them easily.

- In Windows Explorer, right-click an empty location in a folder and select **New»Text Document** from the shortcut menu. Change the file extension of this document from `.txt` to `.udl`. You then can double-click the UDL file to display the **Data Link Properties** dialog box.

Configuring a UDL

Any method of creating a UDL involves the **Data Link Properties** dialog box. Select a data provider from the **Provider** page of this dialog box.

After you select a data provider from the list on the **Provider** page, you can configure the database connection on the **Connection** page. The options on the **Connection** page are different depending upon which provider you choose. For example, the **Connection** page for an ODBC provider contains a selection for a DSN or connection string along with user name and password information. Click the **Test Connection** button to test the database connection after you configure the various properties. Be sure the connection test passes before you click the **OK** button to exit.

Connecting to Databases Using UDLs

Use a path control or constant to specify the path to a UDL file unless you set the **prompt?** input of the DB Tools Open Connection VI to TRUE. The VI in Figure 4-9 uses a path constant to specify the UDL for a Microsoft Access database.

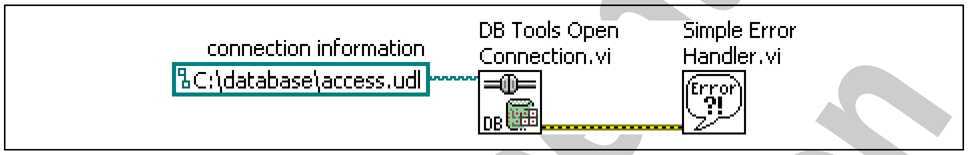


Figure 4-9. Connecting to a Microsoft Access Database Using a UDL

Although you may have created the DSN or UDL correctly, you still may not be able to connect to a specific database because of situations beyond your control. The following situations can prevent you from connecting to a database:

- The requested server is down.
- The network is down.
- All server connections are full, and no other users can connect.
- The maximum number of user licenses has been reached.
- You do not have permission to access the specified database.
- The specified DSN does not exist. Either you are on a different machine, or the specified DSN was deleted.
- The selected data provider is the wrong one for the database.

If the DB Tools Open Connection VI returns errors, you can open the UDL file manually and click the **Test Connection** button on the **Data Link Properties** dialog box to verify that you have the correct settings and that you have access to the database. If the test connection fails, you cannot connect to that database with the Database Connectivity Toolkit. Contact the database administrator for help.

To practice the concepts in this section, complete Exercise 4-2.

D. Performing Standard Database Operations

You can use the Database VIs and function to write data to or read data from, as well as update and delete data from databases.

Writing Data to a Database

Writing data to a database with the LabVIEW Database Connectivity Toolkit is similar to writing data to a file. You open a connection, insert the data, and close the connection when you are finished.

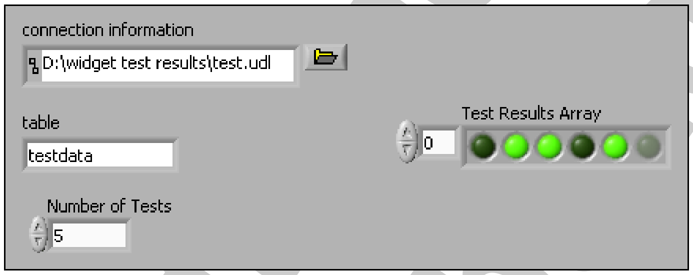


Figure 4-10. Writing Data to a Database Table Front Panel

Figures 4-10 and 4-11 show the front panel and block diagram of a VI that writes the test information to a database table. The connection information is a path to the UDL called `test.udl` and the table name is `testdata`.

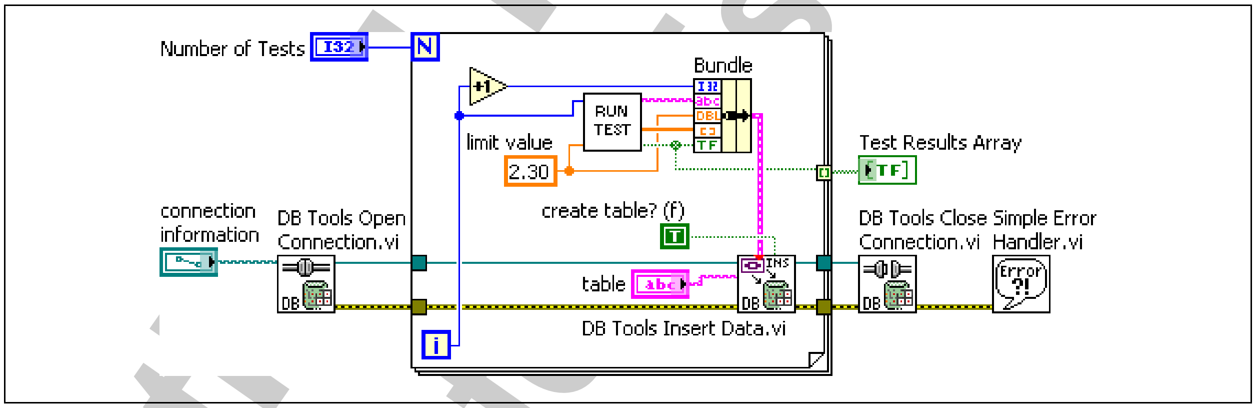
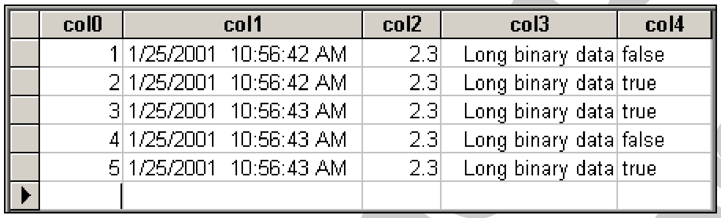


Figure 4-11. Writing Data to a Database Table Block Diagram

Figure 4-11 uses three Database VIs: the DB Tools Open Connection VI, the DB Tools Insert Data VI, and the DB Tools Close Connection VI. The **create table?** input of the DB Tools Insert Data VI is set to TRUE to create the specified table if it does not already exist. If this table does exist, then the data is appended to the existing table. The DB Tools Insert Data VI accepts any type for the data input. If the input type is a cluster, each cluster element is placed into a different field. The LabVIEW data types are converted to the appropriate database data types.

Figure 4-12 shows the `testdata` table as it appears in Microsoft Access. Note that the front panel and block diagram shown in Figures 4-10 and 4-11 respectively do not specify the type of database to use. That configuration occurs when the `test.udl` is created.



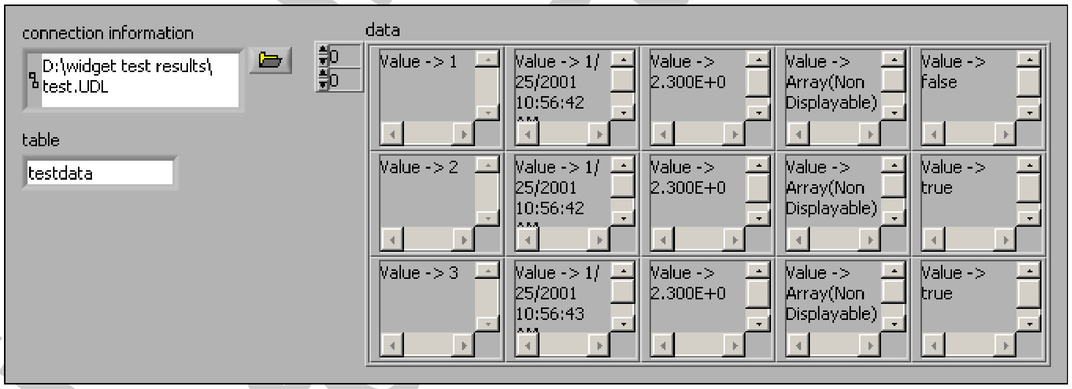
	col0	col1	col2	col3	col4
	1	1/25/2001 10:56:42 AM	2.3	Long binary data	false
	2	1/25/2001 10:56:42 AM	2.3	Long binary data	true
	3	1/25/2001 10:56:43 AM	2.3	Long binary data	true
	4	1/25/2001 10:56:43 AM	2.3	Long binary data	false
	5	1/25/2001 10:56:43 AM	2.3	Long binary data	true
▶					

Figure 4-12. Database Table Displayed in Microsoft Access

Notice that the column names are not specified in the VI, so the table uses default column names. You can specify column names using the **columns** input of the DB Tools Insert Data VI. Wiring an empty array to this input assumes that all columns in the table are to be used.

Reading Data From a Database

Reading data from a database table is similar to writing data to the database. You open a connection to the database, select the data from a table, and then close the connection.



connection information
D:\widget test results\test.UDL

table
testdata

data
0
0

Value -> 1	Value -> 1/25/2001 10:56:42 AM	Value -> 2.300E+0	Value -> Array(Non Displayable)	Value -> false
Value -> 2	Value -> 1/25/2001 10:56:42 AM	Value -> 2.300E+0	Value -> Array(Non Displayable)	Value -> true
Value -> 3	Value -> 1/25/2001 10:56:43 AM	Value -> 2.300E+0	Value -> Array(Non Displayable)	Value -> true

Figure 4-13. Reading Data from a Database Table Front Panel

Figures 4-13 and 4-14 show how you can read the data back from the `testdata` table used in the previous example.

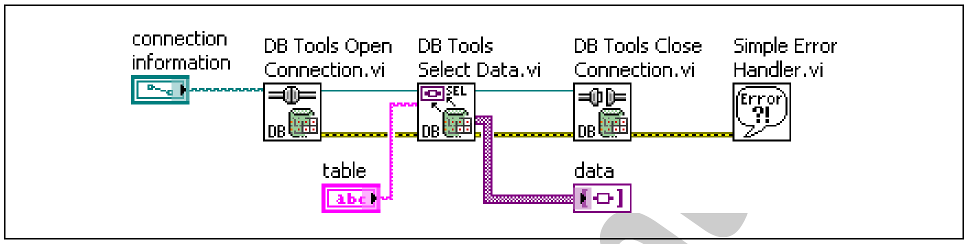
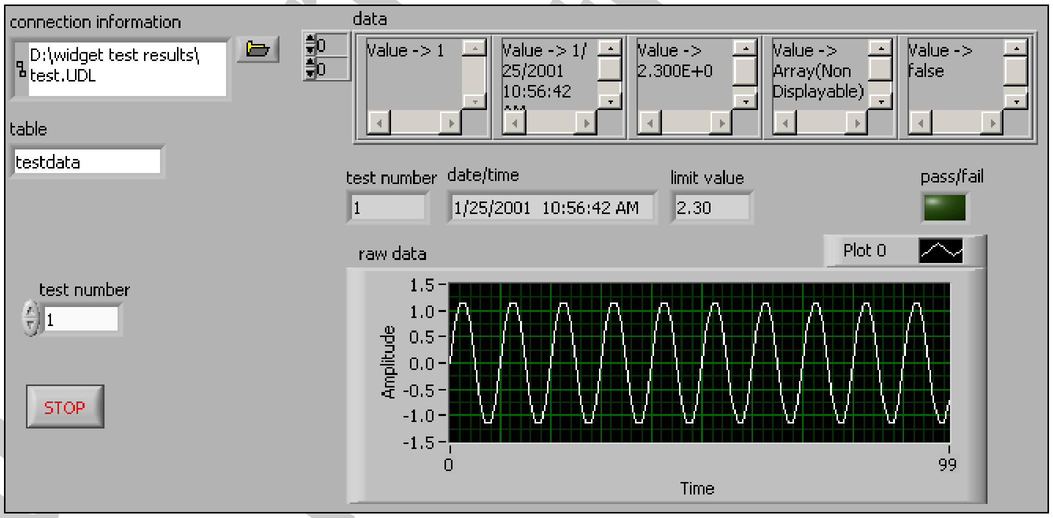


Figure 4-14. Reading Data from a Database Table Block Diagram

Notice in Figures 4-13 and 4-14 that the database data is returned as a two-dimensional array of variants. As the name implies, the Microsoft ActiveX Data Object (ADO) standard is based on ActiveX, which defines variants as its data types. Variants work well in languages such as Visual Basic that are not strongly typed. Because LabVIEW is strongly typed, you must use the Database Variant To Data function to convert the variant data to a LabVIEW data type before you can display the data in standard indicators such as graphs, charts, and LEDs.

Figures 4-15 and 4-16 show the front panel and block diagram for a VI that reads all data from a database table and then converts the data to appropriate data types in LabVIEW.


connection information: D:\widget test results\test.UDL
table: testdata
test number: 1
STOP

data: Value -> 1 | Value -> 1/25/2001 10:56:42 AM | Value -> 2.300E+0 | Value -> Array(Non Displayable) | Value -> false

test number: 1
date/time: 1/25/2001 10:56:42 AM
limit value: 2.30
pass/fail

Figure 4-15. Reading and Converting Data from a Database Table Front Panel

In Figure 4-15, notice that the fourth column of data that does not display properly in either Microsoft Access or the variant is now displayed in a waveform graph.

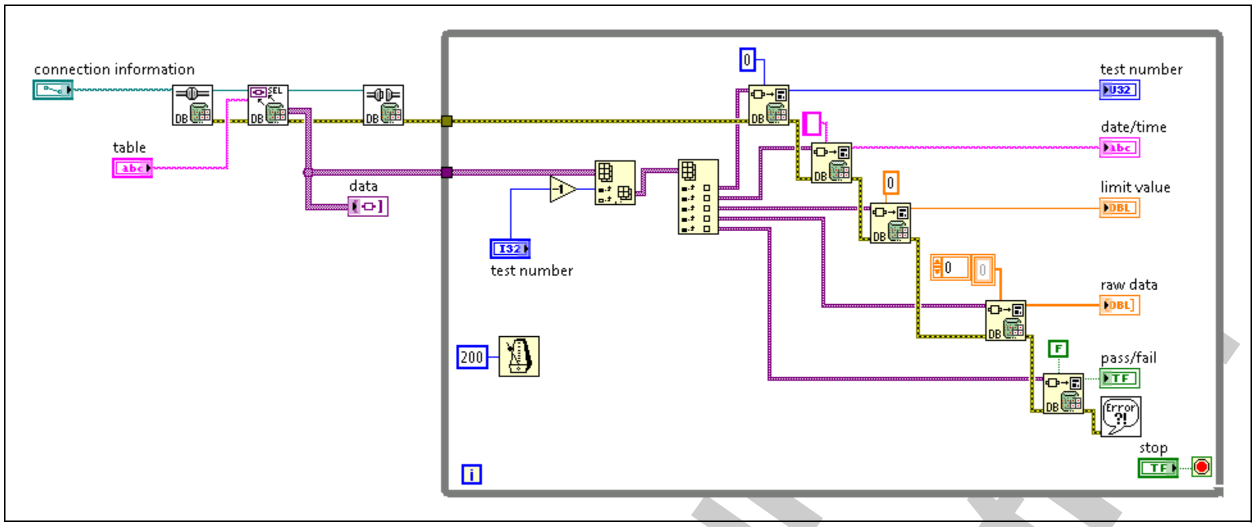


Figure 4-16. Reading and Converting Data from a Database Table Block Diagram

You can use the **table** input of the DB Tools Select Data VI to read data from more than one table in a database. Figure 4-17 shows how you can use a comma-delimited string to specify multiple table names. The **data** array includes all rows and columns from both tables in the order they appear in the **table** string.

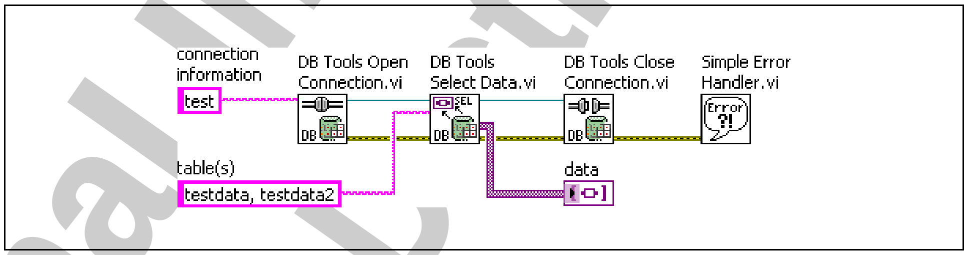


Figure 4-17. Specifying Multiple Database Tables for Reading Data

Limiting Data to Read

If you are reading data from a large table or set of tables, it might take several seconds to return all the data. There is no limit to the size of the database table you can read other than your computer resources, memory, and speed. Read only the necessary fields or perform an SQL query to limit the amount of information to read into LabVIEW at one time. Figure 4-18 shows how you can use the **columns** string array to specify which columns to read to limit the returned data.

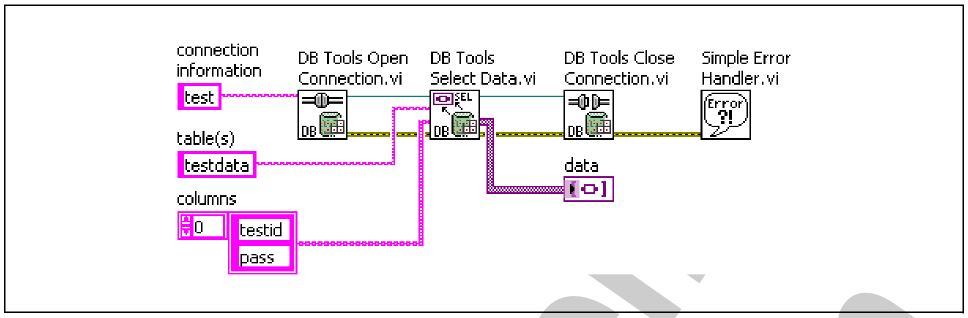


Figure 4-18. Specifying Column Names for Reading Data

In Figure 4-18, only the `testid` and `pass` fields are returned from a table named `testdata`. You can limit the returned data further by specifying conditions using the **optional clause** string. Figure 4-19 shows how you can limit the results from the previous example by returning the `testid` and `testdate` fields for the records where the `pass` field equals `TRUE`.

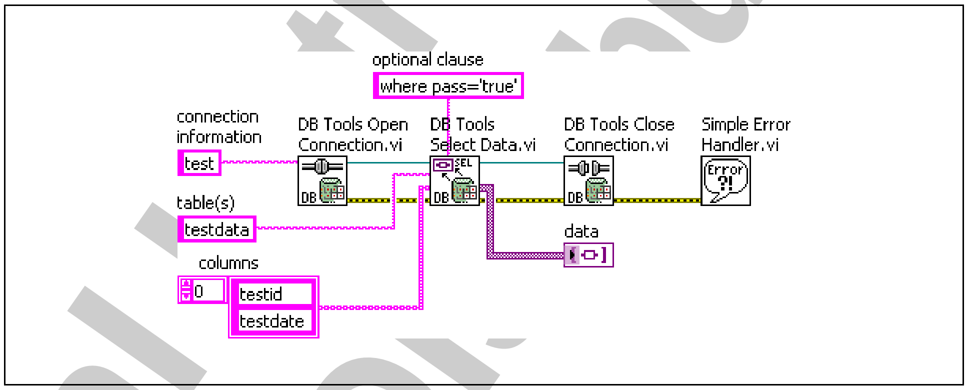


Figure 4-19. Specifying Conditions for Reading Data

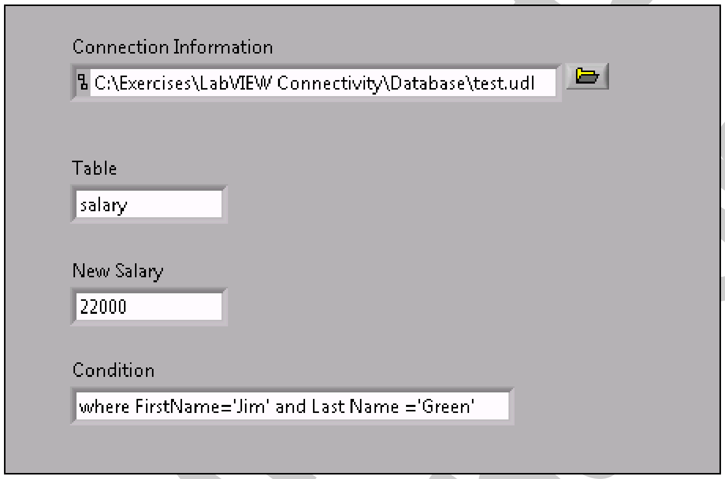
The statement `where pass='true'` is part of an SQL query. Refer to the *Structured Query Language* section of this lesson for information about SQL queries.

Note If you receive an error while using the DB Tools Select Data VI, either a specified field in the **columns** string array does not exist in the table, or a column name contains characters such as a space, `-`, `\`, `/`, or `?`. Do not use these characters when naming tables in a database. However, if an existing database contains such characters, enclosing the column name in double quotes, or single quotes for mysql, often solves the problem.

Updating Data in a Database

You can use the DB Tools, Update Data VI to update data records in a database. You open a connection, update data, and close the connection when you are finished.

Figure 4-20 and Figure 4-21 show the front panel and block diagram of a VI that updates the data record in a database table with specific condition. The connection information is a path to the UDL called `test.udl` and the table name is `salary`.



Connection Information: C:\Exercises\LabVIEW Connectivity\Database\test.udl

Table: salary

New Salary: 22000

Condition: where FirstName='Jim' and Last Name ='Green'

Figure 4-20. Update Data in a Database Table Front Panel

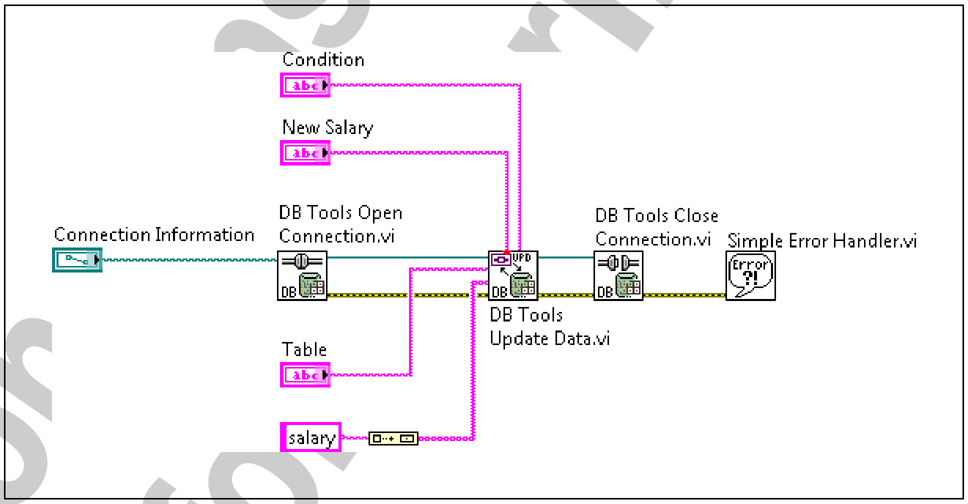
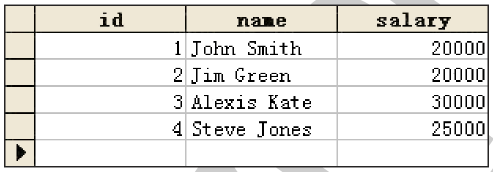


Figure 4-21. Update Data in a Database Block Diagram

The DB Tools Update Data VI accepts any type for the data input. If the input type is a cluster, each cluster element is replaced into a different field. The LabVIEW data types are converted to the appropriate database data types. The **Condition** input specifies an SQL clause that this VI uses to filter the selection criteria. If you do not specify a value for this input or if the input is an empty string, this VI updates all rows in **Table**.

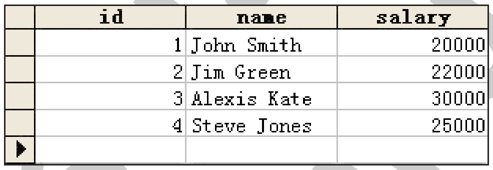
Figures 4-22 and 4-23 show the salary table before and after update as it appears in Microsoft Access.

Note The front panel and block diagram shown in Figure 4-20 and Figure 4-21 do not specify the type of database to use. That configuration occurs when the `test.udl` is created.



id	name	salary
1	John Smith	20000
2	Jim Green	20000
3	Alexis Kate	30000
4	Steve Jones	25000

Figure 4-22. Database Table Before Update



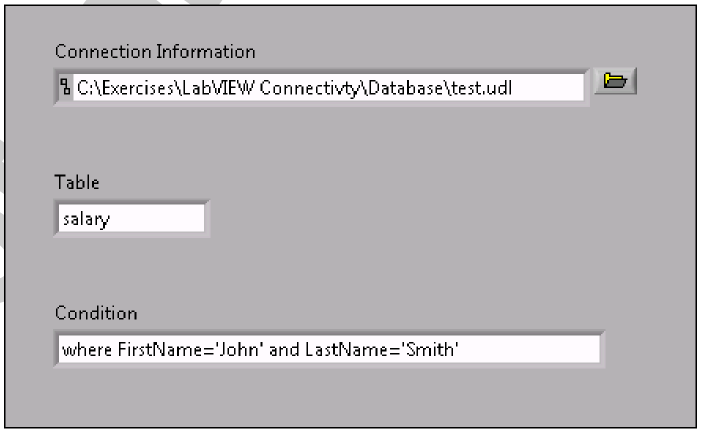
id	name	salary
1	John Smith	20000
2	Jim Green	22000
3	Alexis Kate	30000
4	Steve Jones	25000

Figure 4-23. Database Table After Update

Deleting Data in a Database

You can use the DB Tools Delete Data VI to delete data records in a database. You open a connection, delete data, and close the connection when you are finished.

Figures 4-24 and 4-25 show the front panel and a block diagram of a VI that deletes the data records in a database table with specific condition. The **Connection Information** is a path to the UDL called `test.udl` and the table name is `salary`.



Connection Information
C:\Exercises\LabVIEW Connectivty\Database\test.udl

Table
salary

Condition
where FirstName='John' and LastName='Smith'

Figure 4-24. Delete Data in a Database Table Front Panel

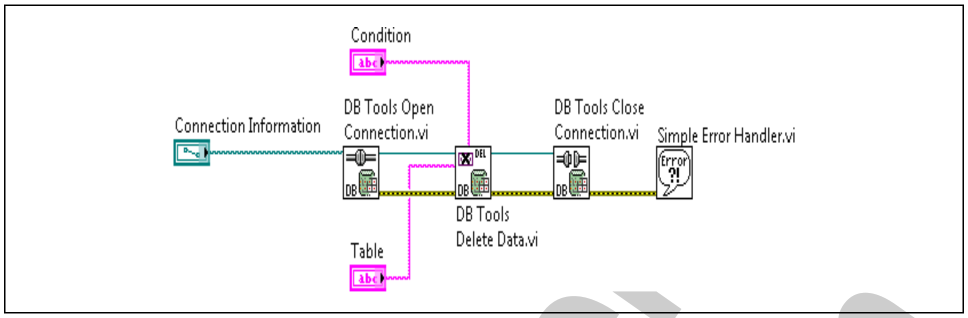
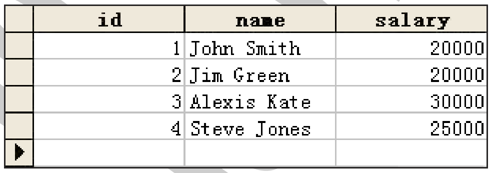


Figure 4-25. Delete Data in a Database Table Block Diagram

Figure 4-25 uses three Database VIs: the DB Tools Open Connection VI, the DB Tools Delete Data VI, and the DB Tools Close Connection VI.

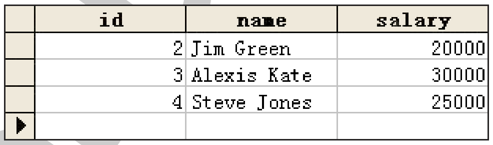
The DB Tools Delete Data VI deletes data from a database identified by connection reference. The condition input specifies an SQL clause that this VI uses to filter the selection criteria. If you wire an empty string to this input, this VI deletes all data from table.

Figures 4-26 and 4-27 show the salary table before and after delete as it appears in Microsoft Access.



id	name	salary
1	John Smith	20000
2	Jim Green	20000
3	Alexis Kate	30000
4	Steve Jones	25000

Figure 4-26. Database Table Before Deleting Data



id	name	salary
2	Jim Green	20000
3	Alexis Kate	30000
4	Steve Jones	25000

Figure 4-27. Database Table After Deleting Data

Note The front panel and block diagram shown in Figures 4-16 and 4-17 respectively do not specify the type of database to use. That configuration occurs when the `test.udl` is created.

Database Variant to Data

You can use Database Variant to Data VI to convert a database variant to the LabVIEW data type specified in type so you can use the data in another function or VI.

Database Variant to Data is designed so you can convert a single database variant or, row (or column) of data, or even a whole table of data at one time. Figure 4-28 shows the inputs and outputs of this function.

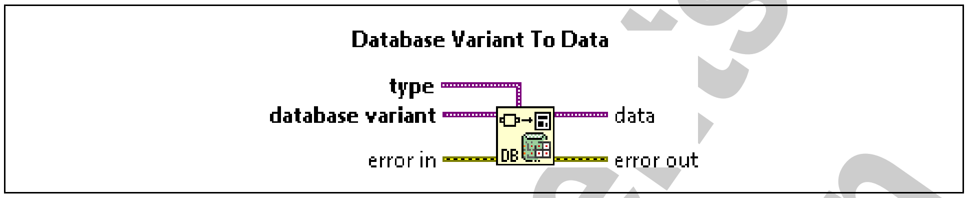


Figure 4-28. Database Variant to Data VI

You can wire a variant, a 1D array of variants or a 2D array of variants to the database variant input. Depending on the data type that you wire to the type and database variant inputs. The Database Variant to Data function performs the different conversions shown in Table 4-2.

Table 4-2. Database Variant to Data Function Conversions

Data Type of "database variant"	**Data Type of "type"**				
	Cluster	**1D Array of Non-Clusters**	**1D Array of Clusters**	**2D Array**	**Others**
Variant	This function converts the variant to the LabVIEW data type specified in **type**				
1D Array of Variants	Each element in the cluster defines a column. This function converts each element of the 1D array to a column in sequence.	This function converts each element of the 1D array to the element data type specified in **type**. The output is a 1D array.	This function converts each element of the 1D array to a cluster. The output is a 1D array of clusters.	The wire of **database variant** is broken.	The wire of **database variant** is broken.
2D Array of Variants	The wire of **database variant** is broken.	The wire of **database variant** is broken.	This function converts each row of the 2D array to a cluster. The output is a 1D array of clusters.	This function converts each element of the 2D array to the element data type specified in type. The output is a 2D array.	The wire of **database variant** is broken.
Others	The wire of **database variant** is broken.				

Using the Database Connectivity Toolkit Examples

The Database Connectivity Toolkit provides several examples that demonstrate how to perform common database operations with the Database VIs. These examples use a UDL called `LabVIEW.udl` to link to a Microsoft Access database named `LabVIEW.mdb`.

Using the Examples with Other Databases

You can use the Database Connectivity Toolkit example VIs by modifying the `LabVIEW.udl` file. Double-click this UDL file in the `labview/examples/database` directory to display the **Data Link Properties** dialog box. You then can select a different provider and set the connection properties for your DBMS. The default values for some of the example VIs assume the presence of a particular table in the database from which to read data or to which to add data. You must modify the example to fit the table names, column names, and data types required.

Using the Example without a Database

You do not need to have Microsoft Access or any other database installed to use the Database Connectivity Toolkit examples. If you run the examples with their default values, the data is read from or written to the `LabVIEW.mdb` file even if you do not have Microsoft Access installed. If you want to create a new database file to write data to and read data from, you can copy and rename the `LabVIEW.mdb` file and use the DB Tools Drop Table VI to remove the existing tables. You then can use the DB Tools Create Table VI to create new tables specific to the application.

To practice the concepts in this section, complete Exercise 4-3.

To practice the concepts in this section, complete Exercise 4-4.

E. Structured Query Language

The Structured Query Language (SQL) consists of a set of character string commands and is a widely supported standard for database access. SQL is the language that relational databases use. You can use the SQL commands to describe, store, retrieve, and manipulate the rows and columns in database tables.

SQL is a non-procedural language for processing sets of records in database tables. The following are three pertinent classes of SQL statements:

- **Data Definition/Control Language (DDL/DCL) statements—**Define and control the structure of the database. They also define and grant access privileges to database users. Use the statements to create, define, and alter databases and tables.
- **Data Manipulation Language (DML) statements**—Operate on the data contents of database tables. You use these statements to insert rows of data into a table, update rows of data in a table, delete rows from a table, and conduct database transactions.
- **Queries**—Are SQL SELECT statements that specify which tables and rows are retrieved from the database.

Table 4-3 describes some frequently used SQL commands.

Table 4-3. Common SQL Commands

Command	Function
CREATE TABLE	Creates a database table and specifies the name and data type for each column therein. The result is a named table in the database. CREATE TABLE is a DDL command. DROP TABLE is the complementary DDL command.
INSERT	Adds a new data row to the table, allowing values to be specified for each column. INSERT is a DML command.
SELECT	Initiates a search for all rows in a table whose column data satisfy specified combinations of conditions. The result is an active set of rows that satisfy the search conditions. SELECT is a query command.
UPDATE	Initiates a search as in SELECT, then changes the contents of specific column data in each row in the resulting active set. UPDATE is a DML command.
DELETE	Initiates a search as in SELECT, then removes the resulting active set from the table. DELETE is a DML command.

The following line is an example of a typical SQL statement:

```
SELECT product, lot FROM prodstats WHERE lot=R43E2
```

This SQL command retrieves the product and lot columns from the prodstats table, including only those rows where the value contained in the lot column equals R43E2. This query creates a one-row, two-column active set as shown in Table 4-4.

Table 4-4. Example Query Results

widget	R43E2

SQL Dialects

Several database publishers use their own SQL dialects in their products. These dialects usually consist of extensions to the standard SQL commands that perform higher-level or database-specific functions. Conversely, certain accessible databases do not directly support standard SQL functions. Differences are most noticeable when using the CREATE TABLE and ALTER TABLE commands. Most of the databases use their own particular column type keywords. Many of the databases have different syntax for date-and-time formats. ODBC-compliant SQL Toolkit software helps to minimize the effects of these SQL variants.

Executing SQL Statements and Fetching Data

With SQL, you can perform common operations like creating and deleting tables, inserting data into databases, querying databases for particular recordsets, and manipulating data in tables. This section describes how you can use SQL statements with the LabVIEW Database Connectivity Toolkit and how you can fetch the data resulting from an SQL query.

Use the DB Tools Execute Query VI to send an SQL string to a database, as shown in Figure 4-29. You then can use the DB Tools Fetch Element Data VI, the DB Tools Fetch Next Recordset VI, or the DB Tools Fetch Recordset Data VI to return the results of a query.The SQL string does not have to specify only a query. You can enter any SQL statement in the SQL string.

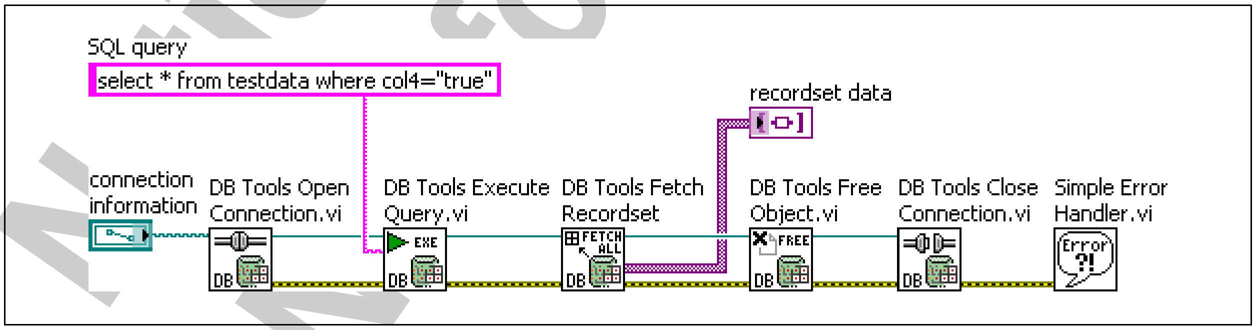

SQL query
select * from testdata where col4="true"
recordset data
connection information
DB Tools Open Connection.vi
DB Tools Execute Query.vi
DB Tools Fetch Recordset
DB Tools Free Object.vi
DB Tools Close Connection.vi
Simple Error Handler.vi

Figure 4-29. Fetching Query Results

The **SQL query** string shown in Figure 4-29 asks for all records in the testdata table where the fifth field contains a TRUE value. The DB Tools Fetch Recordset Data VI returns a two-dimensional array of variants for which all tests passed. Refer to the *Performing Standard Database Operations* section for more information about converting variant data to LabVIEW data types. Because the DB Tools Execute Query VI creates a Recordset reference, you then must use the DB Tools Free Object VI to release the Recordset reference value.

The DB Tools Fetch Recordset Data VI returns all records from a query such as the one shown in Figure 4-29.

Note A record is a single row of data and a recordset is a collection of records, or multiple rows, from a database table.

To practice the concepts in this section, complete Exercise 4-5.

Self-Review: Quiz

1. Match each item to its description:

Database	a. Contains records and fields
Tables	b. Set of string commands for database access
SQL	c. Organized collection of data

2. Which of the following are valid ways of connecting to a database using the OLE DB standard?
   a. Wire a TRUE constant to the Prompt input of the DB Tools Open Connection VI
   b. Wire the filepath of a valid UDL file to the Filepath input of the DB Tools Open Connection VI
   c. Wire the name of a DSN to the Filepath input of the DB Tools Open Connection VI

3. Which VI converts database data into a more usable LabVIEW data type?
   a. DB Tools Open Connection
   b. DB Tools Select Data
   c. DB Tools Insert Data
   d. Database Variant to Data

Self-Review: Quiz Answers

1. Match each item to its description:

Database	**c. Organized collection of data**
Tables	**a. Contains records and fields**
SQL	**b. Set of string commands for database access**

2. Which of the following are valid ways of connecting to a database using the OLE DB standard?
   a. Wire a TRUE constant to the Prompt input of the DB Tools Open Connection VI
   b. Wire the filepath of a valid UDL file to the Filepath input of the DB Tools Open Connection VI
   c. Wire the name of a DSN to the Filepath input of the DB Tools Open Connection VI

3. Which VI converts database data into a more usable LabVIEW data type?
   a. DB Tools Open Connection
   b. DB Tools Select Data
   c. DB Tools Insert Data
   d. Database Variant to Data

Notes

5

Broadcasting Data Using UDP and Serving Data to a Client Using TCP

You can use TCP/IP to communicate over single networks or interconnected networks. The individual networks can be separated by large geographical distances. TCP/IP routes data form one network or Internet-connected computer to another. Because TCP/IP is available on most computers, it can transfer information among diverse systems.

The Transport Control Protocol, or TCP, uses the Internet Protocol (IP) to provide reliable transmission across networks, delivering data in a sequence without errors, loss, or duplication. This lesson describes using a client/server model to reliably send data using TCP.

The User Data Protocol, or UDP, also uses the Internet Protocol to transfer data but does so in a lighter weight way than TCP. UDP provides a means for communicating short packets of data to one or more recipients. This lesson describes using UDP and LabVIEW to implement the broadcast model.

Topics

A. Broadcasting Data Overview

Broadcasting data is a method of placing data onto a network. Broadcasting data is not concerned with the network entities who are listening to the data being broadcast. Also, broadcasting data cannot guarantee that the data reaches its destination. The User Datagram Protocol, otherwise known as UDP, is the networking protocol that implements broadcasting.

Broadcast Communication Model Definition

UDP provides simple, low-level communication among processes on computers. Processes communicate by sending datagrams to a destination computer or port. A port is the location where you send data. Internet Protocol (IP) handles the computer-to-computer delivery. After the datagram reaches the destination computer, UDP moves the datagram to its destination port. If the destination port is not open, UDP discards the datagram. UDP shares the same delivery problems of IP.

Use UDP in applications in which reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic.

The broadcast model is very similar to a radio tower broadcasting a signal. The radio station doesn’t need to know who is listening to the broadcast signal. Also, there is the possibility that the listener is not able to receive the signal due to interference from other signals. The advantage to the broadcast model is there is no limit to the number of listeners, similar to a radio broadcast, as shown in Figure 5-1.

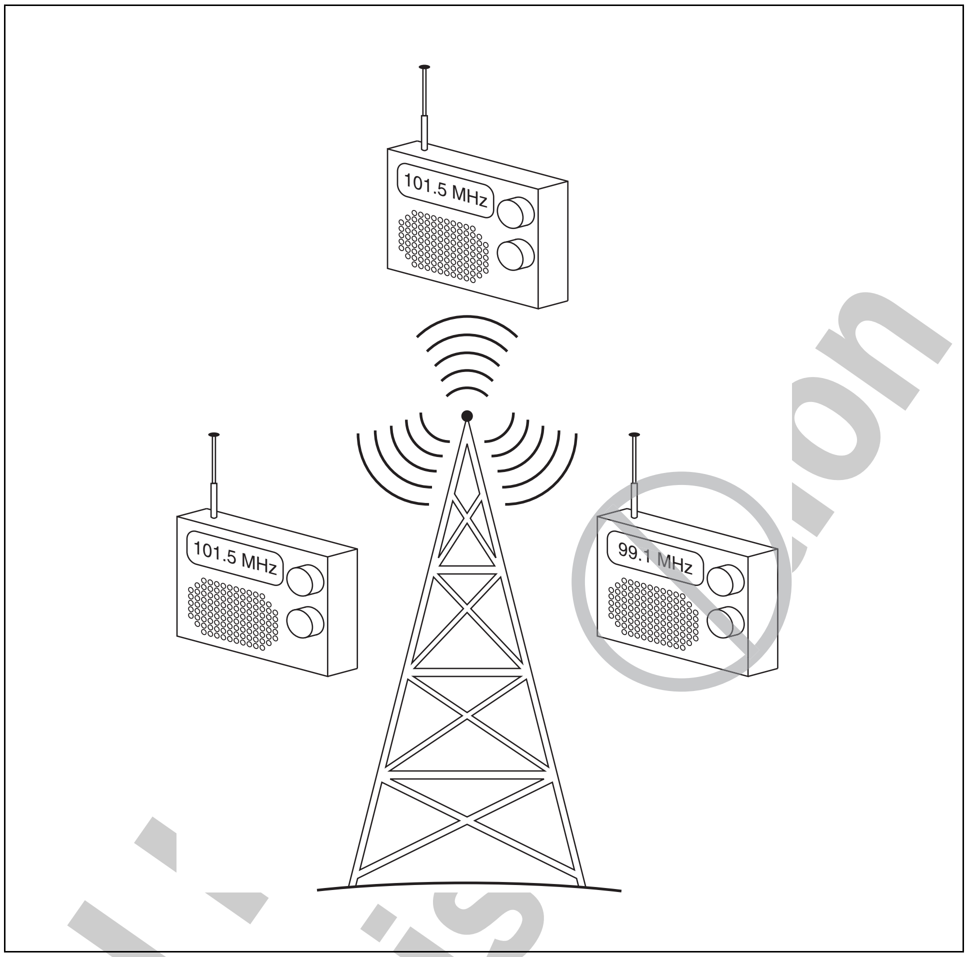


Figure 5-1. Broadcast Model

UDP Communication Capabilities and Issues

Unlike TCP, UDP does not guarantee the safe arrival of data to the destination. Furthermore, data sent in multiple packets may not arrive at the destination in the order they were sent. Therefore you should use UDP only to send short, non-critical messages to one or more destinations.

Because UDP has little communication control, you do not need an explicit connection to the other side of communication to send or receive data. A receiver listens on a specified UDP port to receive any data broadcast to that port.

In the example shown in Figure 5-2 and Figure 5-3, any data received is printed on a string indicator, along with the source address and port. If a timeout occurs on the UDP read function (after 250 ms) the VI ignores the error, nothing prints to the indicator, and the loop returns to the beginning and listens for data on the UDP port. Notice that the UDP connection must be opened and closed with the corresponding UDP functions.

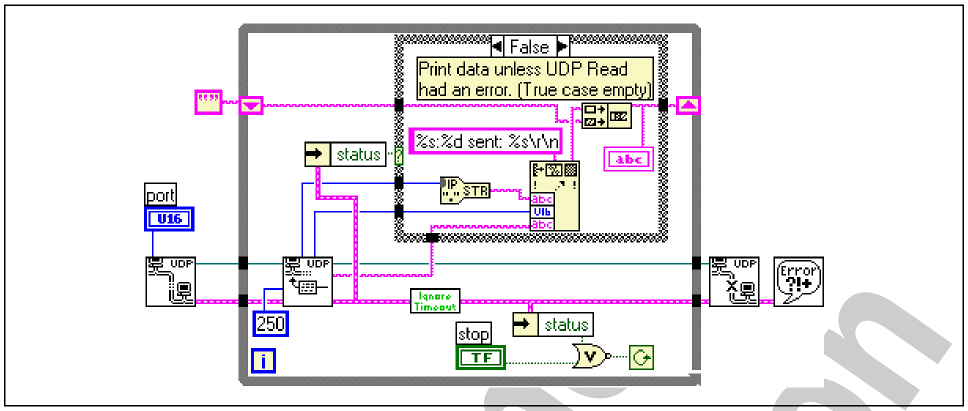


Figure 5-2. UDP Receiver

The broadcaster is simpler because it does not require a wait. The broadcaster opens the UDP port, sends the data, and closes the port.

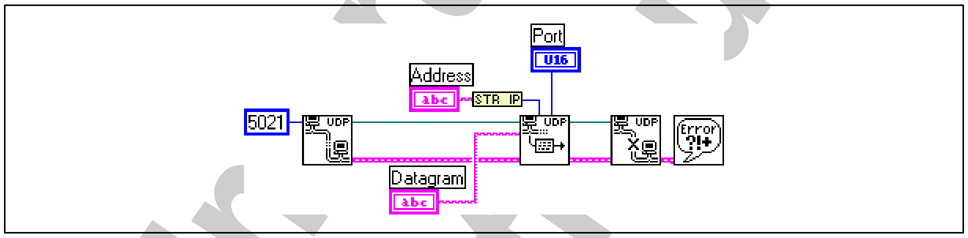


Figure 5-3. UDP Broadcaster

Even though UDP has reliability issues, some applications may be better suited to UDP for the following reasons:

- **No connection establishment**—Unlike TCP, UDP does not wait to confirm a connection before transferring data, so no delay is introduced.
- **No connection state**—Because UDP does not maintain a connection state, which includes send and receive buffers and congestion control settings, a UDP broadcaster can support more receivers.
- **Small overhead**—UDP segments have 8 bytes of overhead, compared to 20 bytes of overhead for TCP segments.
- **Unregulated send rate**—UDP send rate is only limited by the rate of data generation, CPU, clock rate, and access to Internet bandwidth

B. Implementing Broadcast Models

You can create a UDP-based distributed system by using the LabVIEW UDP VI and functions.

Creating a Broadcast Model Distributed System

You can use the UDP functions to communicate to a single client (single-cast) or to all computers on the subnet through a broadcast. If you want to communicate to multiple specific computers, you must write custom code for the UDP functions to iterate through a list of clients or use multicasting. Using this technique creates duplicate network traffic because LabVIEW sends a separate copy of the data to each client and maintains a list of clients interested in receiving the data.

Use multicasting to communicate between a single sender and multiple clients on a network without requiring the sender to maintain a list of clients or send multiple copies of the data to each client. To receive data broadcast by a multicast sender, all clients must join a multicast group. The sender does not have to join a group to send data. The sender specifies a multicast IP address, which defines a multicast group. Multicast IP addresses are in the `224.0.0.0` to `239.255.255.255` range. When a client wants to join a multicast group, it subscribes to the multicast IP address of the group. After subscribing to a multicast group, the client receives data sent to the multicast IP address.

To multicast in LabVIEW, use the **UDP Multicast Open** VI to open connections capable of reading, writing, or reading and writing UDP data. Specify the time-to-live (TTL) for writing data, the multicast address for reading data, and the multicast port number for reading and writing data. The default TTL is 1, which means LabVIEW sends the datagram only to the local subnet. When a router receives a multicast datagram, it decrements the datagram TTL. If the TTL is greater than 1, the router forwards the datagram to other routers. The following table lists what action occurs to a multicast datagram when you specify a value for the **time-to-live** parameter.

`0`	Datagram remains on the host computer.
`1`	Datagram sent to every client on the same local subnet that subscribes to that IP address. Hubs/repeaters and bridges/switches forward the datagram. Routers do not forward the datagram if the TTL is `1`.
`>1`	Datagram is sent and routers forward it through TTL –1 layers.

UDP VI and Functions

Use the UDP VI and functions to exchange data with devices on a remote UDP socket.

- **UDP Open**—Opens a UDP socket on the **port**. Close the socket with the UDP Close function. Use the UDP Multicast Open VI instead of this function to open connections capable of reading, writing, or reading and writing data to or from UDP Multicast sockets.
- **UDP Multicast Open**—Opens a UDP multicast socket on the **port**. You must manually select the polymorphic instance you want to use. Use the pull-down menu to select an instance of this VI. You can select from Read-Only, Read-Write and Write-Only instances.
- **UDP Read**—Reads a datagram from a UDP socket, returning the results in data out. The function returns data when it receives any bytes, and waits the full timeout ms only if it receives no bytes.
- **UDP Write**—Writes to a remote UDP socket.
- **UDP Close**—Closes a UDP socket.

C. TCP Overview

TCP ensures reliable transmission across networks, delivering data in sequence without errors, loss, or duplication. TCP retransmits the datagram until it receives an acknowledgment.

TCP is a connection-based protocol, which means that sites must establish a connection before transferring data. The data transmission occurs between a client and a server. TCP permits multiple, simultaneous connections.

You initiate a connection by waiting for an incoming connection or by actively seeking a connection with a specified address. In establishing TCP connections, you have to specify the address and a port at that address. Different ports at a given address identify different services at that address.

Client/Server Model Overview

The client/server model is a common model for networked applications. In this model, one set of processes (clients) requests services from another set of processes (servers) as shown in Figure 5-4.

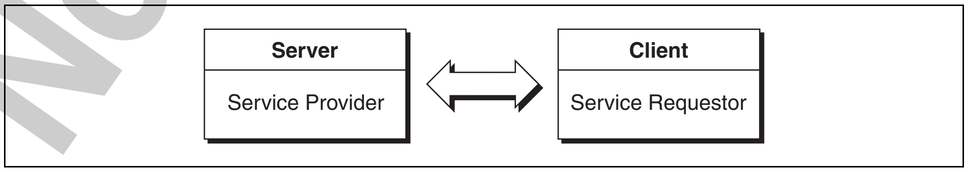


Figure 5-4. Client/Server Model

Client Model Overview

Figure 5-5 shows the flowchart model for TCP/IP communication in LabVIEW.

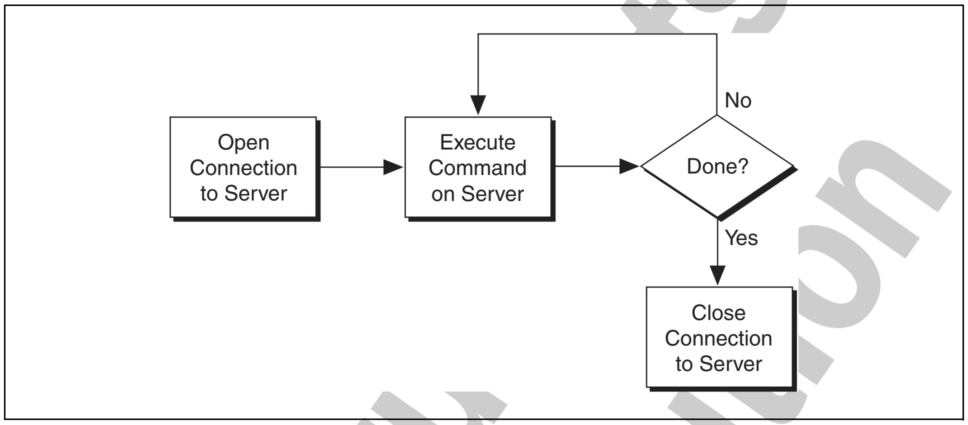

Open Connection to Server
Execute Command on Server
Done?
No
Yes
Close Connection to Server

Figure 5-5. LabVIEW TCP/IP Client Model

For higher performance, you can process multiple commands after you open the connection. After executing the commands, you can close the connection. This flowchart serves as a model to demonstrate implementing a given protocol in LabVIEW.

Client Model Communication Steps

The client model allows a client to communicate to a server by first opening a connection to the server, sending a command to the server, receiving a command from the server, and then closing the connection to the server.

Open a Connection to the Server

The first step is to open a connection to the server. The client must open a network port on the local machine, and then specify the IP address of the server to connect to.

Send Commands to the Server

After a connection is established to the server, the client sends a message that the server recognizes. This message causes the server to perform work.

Receive Responses from the Server

After a message is sent to the server, the client needs to know that the message was accepted and work was performed by the server. This occurs by the server responding to the client.

Close the Connection to the Server and Report Errors

After all of the commands are sent to the server, the client closes the connection, and reports errors.

Server Model Overview

Figure 5-6 shows a simplified model for a server in LabVIEW.

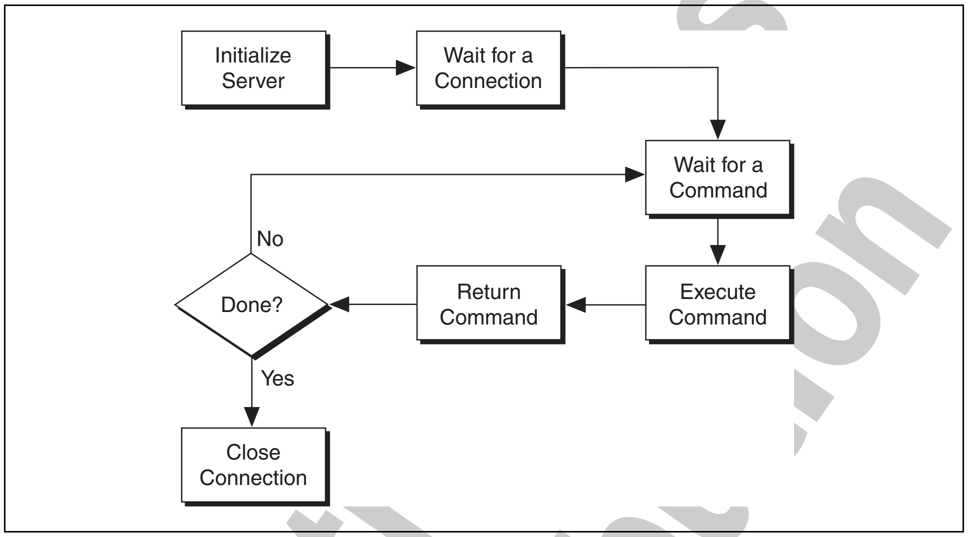


Figure 5-6. LabVIEW TCP/IP Server Model

LabVIEW repeats this entire process until it is shut down remotely by sending a command to end the VI. This VI does not report errors. It might return a response indicating that a command is invalid, but it does not display a dialog box when an error occurs. Because a server might be unattended, consider carefully how the server should handle errors. You probably do not want to display a dialog box on an error because it requires user interaction at the server. Someone would need to click the **OK** button. However, you might want LabVIEW to write a log of transactions and errors to a file or a string.

You can increase performance by allowing the connection to stay open. You can receive multiple commands this way, but it also blocks other clients from connecting until the current client disconnects. You can restructure the block diagram to handle multiple clients simultaneously, as shown in Figure 5-7.

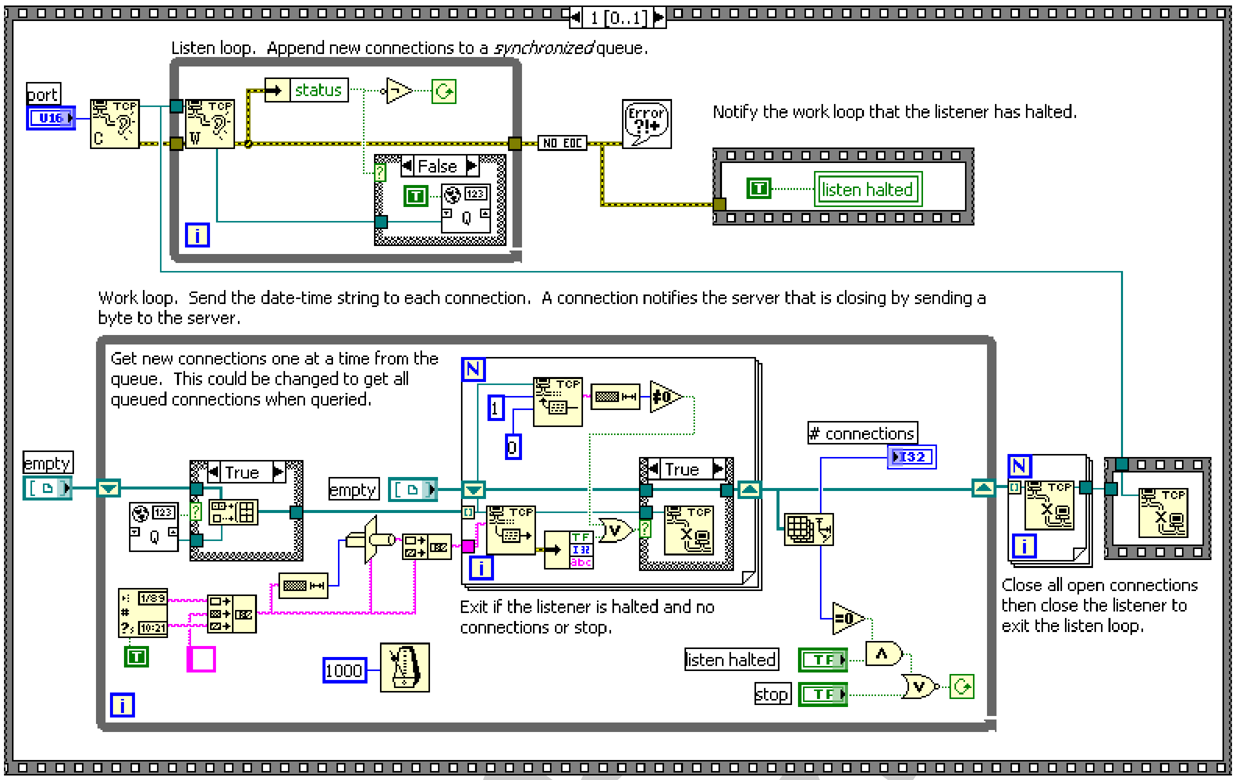

1 [0..1]
Listen loop. Append new connections to a *synchronized* queue.
port
U16
status
False
NO EOC
Notify the work loop that the listener has halted.
listen halted
Work loop. Send the date-time string to each connection. A connection notifies the server that is closing by sending a byte to the server.
Get new connections one at a time from the queue. This could be changed to get all queued connections when queried.
empty
True
# connections
I32
Exit if the listener is halted and no connections or stop.
1000
listen halted
stop
Close all open connections then close the listener to exit the listen loop.

Figure 5-7. Handling Multiple Clients Simultaneously

The block diagram uses the LabVIEW multitasking capabilities to run two loops simultaneously. The top loop continuously waits for a connection and then adds the connection to a synchronized queue. The bottom loop checks each of the open connections and executes any commands that have been received. If an error occurs on one of the connections, the connection is disconnected. When the user aborts the server, all open connections close.

Server Model Communication Steps

The server model is more complex to allow for multiple clients to access the server. The server must first initialize itself, and then wait for a connection from a client. Once the connection is established, the server will wait for a command. The server will process and execute any commands and return results back to the client. Once the client is finished sending commands to the server, the connection between the server and client can be closed.

Initialize the Server

The server is initialized by listening for a connection on a particular network port.

Wait for a Connection

The server model allows for multiple clients to connect to the server. This is accomplished by waiting for a connection from a client, and then placing the connection information into a queue using a producer/consumer design pattern. The consumer waits for a command to be received from the client.

Wait for a Command

Waiting for a command allows multiple commands to be received from multiple clients. This is accomplished by using a consumer to iterate through each of the clients to determine if a message has been sent by a client.

Execute the Command and Return Results

After a command is received by the server from a client, the server executes the command by performing work. Once the work is finished, the server responds with the results of the work or an error if it was not possible to perform the work.

Close the Connection

After all of the clients have finished sending messages to the server, the connection to the network port must be closed.

Client/Server Model Capabilities and Issues

The client/server model provides a reliable mechanism for sending data on a network. But, with the added reliability, the client/server model adds complexity over the broadcast model. This complexity is due to the extra amount of communication between the client and the server to verify that messages are being received. This extra amount of communication causes the client/server model to be slower.

To practice the concepts in this section, complete Exercise 5-1.

D. Implementing the Client/Server Model

Use the TCP/IP VI and functions located on the **TCP** palette to implement the client/server model using LabVIEW as a client or server application.

TCP/IP VI and Functions

The TCP/IP VIs are organized similar to other LabVIEW functions, where you must first open, then read/write, and close. The LabVIEW TCP/IP VIs have other functionality that is specific to the maintenance of a TCP/IP network connection such as the TCP Listen VI, and the IP to String, and String to IP functions.

TCP Open Connection Function (Client)

Use the TCP Open Connection function to actively establish a connection with a specific address and port. If the connection is successful, the function returns a network connection refnum that uniquely identifies that connection. Use this connection refnum to refer to the connection in subsequent VI calls.

If the connection is not established in the specified timeout period, the function completes and returns an error.

connection ID is a network connection refnum that uniquely identifies the TCP connection. **error in** and **error out** clusters describe any error conditions.

The **address** identifies a computer on the network and can be expressed in IP dot notation or as the hostname. The **remote port or service name** is an additional number that identifies a communication channel on the computer that the server uses to listen for communication requests. When you create a TCP server, you specify the port that you want the server to use for communication. If the connection is successful, the TCP Open Connection function returns a **connection ID** that uniquely identifies that connection. You use this connection ID to refer to the connection in subsequent VI calls.

TCP Read Function (Client and Server)

The TCP Read function reads a number of bytes from a TCP network connection, returning the results in **data out**. **mode** indicates the behavior of the read operation based on the following four options.

- **Standard (default)—**Waits until all bytes you specify in **bytes to read** arrive or until **timeout ms** runs out. Returns the number of bytes read so far. If fewer bytes than the number of bytes you requested arrive, returns the partial number of bytes and reports a timeout error.
- **Buffered—**Waits until all bytes you specify in **bytes to read** arrive or until **timeout ms** runs out. If fewer bytes than the number you requested arrive, returns no bytes and reports a timeout error.
- **CRLF—**Waits until all bytes you specify in **bytes to read** arrive or until the function receives a CR (carriage return) followed by a LF (linefeed) within the number of bytes you specify in **bytes to read** or until **timeout ms** runs out. The function returns the bytes up to and including the CR and LF if it finds them in the string.
- **Immediate—**Waits until the function receives any bytes from those you specify in **bytes to read**. Waits the full timeout only if the function receives no bytes. Returns the number of bytes so far. Reports a timeout error if the function receives no bytes.TCP Write Function (Client and Server).

The TCP Write function writes data to a TCP network connection.

All the data written or read is in a string data type. The TCP/IP protocol does not state the type or format of the data transferred, so a string type is the most flexible method. You can use the Flatten to String function to send binary or complicated data types.

If you send binary or complicated data types, you must inform the receiver of the exact type and representation of the data sent to reconstruct the original information. Also, when you use the TCP Read function, you must specify the number of bytes to read. A common method is to send a 32-bit integer first to specify the length of the data string that follows. The TCP example VIs provided with LabVIEW and the exercises in this lesson provide more information about these topics and about how the data typically is formatted for TCP/IP communications.

TCP Close Connection Function (Client and Server)

Use the TCP Close Connection function to close the connection to the remote application. If unread data remains and the connection closes, you might lose data. Use a higher level protocol for your computer to determine when to close the connection.

TCP Listen VI, TCP Create Listener Function and TCP Wait on Listener Function (Server)

A communication program needs the ability to wait for an incoming connection. The procedure is to create a listener and wait for an accepted TCP connection at a specified port. If the connection is successful, the function returns a **connection ID** and the address and port of the remote machine. Refer to the Data Server VI located in `examples\comm\TCP.llb` for an example of this architecture.

String to IP

The String to IP function converts a string to an IP network address or an array of IP network addresses. If String to IP is in single output mode, the **net address** is the first result returned by the operating system resolver. If String to IP is in multiple output mode, the result is an array of all IP network addresses returned by the operating system resolver. If the node fails to convert the string, the result is a value of zero in single output mode or an empty array in multiple output mode.

IP to String

The IP to String function converts an IP network address to a string.

To practice the concepts in this section, complete Exercise 5-2.

Self-Review: Quiz

1. Match the protocols to the statements that best describes them.

UDP	a. Connection based
TCP	b. Does not guarantee safe arrival
	c. Packets guaranteed to be received in same order they were sent
	d. Small overhead

2. Match each term to its description.

Client	a. Waits for a connection and then reads or writes data as instructed
Server	b. Initiates the connection and instructs to read and/or write data

3. For each of the following TCP functions, identify if it is used in **client** applications, **server** applications, or **both** client and server applications.
   a. TCP Open Connection
   b. TCP Create Listener
   c. TCP Wait on Listener
   d. TCP Write
   e. TCP Read
   f. TCP Close

Self-Review: Quiz Answers

1. Match the protocols to the statements that best describes them.
   a. Connection based—**TCP**
   b. Does not guarantee safe arrival—**UDP**
   c. Packets guaranteed to be received in same order they were sent—**TCP**
   d. Small overhead—**UDP**

2. Match each term to its description.

Client	b. Initiates the connection and instructs to read and/or write data
Server	a. Waits for a connection and then reads or writes data as instructed

3. For each of the following TCP functions, identify if it is used in **client** applications, **server** applications, or **both** client and server applications.
   a. TCP Open Connection—**client**
   b. TCP Create Listener—**server**
   c. TCP Wait on Listener—**server**
   d. TCP Write—**both**
   e. TCP Read—**both**
   f. TCP Close—**both**

Notes

6

Using LabVIEW Web Services

In Lesson 5, you learned to use the TCP and UDP VIs for network communication. These VIs are available for users who are interested developing networked applications at a lower-level. Low-level networked applications are beneficial for quickly establishing extremely simple data communication, but requires expertise in order to develop any advanced functionality.

In this lesson, you will learn how to use LabVIEW Web services to provide a standard, Web-based interface for communication with a LabVIEW application. LabVIEW can be used on both the server side and the client side. You can use LabVIEW Web services when LabVIEW is not available on the client machine.

Topics

A. Web Services

Use LabVIEW Web services to exchange data with VIs over a network. Any HTTP-capable Web client, including a standard Web browser, can invoke VIs and exchange data using a URL and standard HTTP methods such as GET and POST. You can conduct application-to-application data exchange between numerous HTTP-capable devices and software from both National Instruments and third parties

What is a Web Server?

A Web server is a software program running on a computer that listens on one or more ports for resource requests arriving from other computers, formatted as valid HTTP requests.

When a request arrives, the Web server breaks the request down into its component pieces, then uses the information in those pieces to determine exactly what service or resource the requester, commonly called a client, is requesting. The URL is the primary part of the request, as it is the effective "address" of the request. By looking at a requested URL and comparing it against the services and resources it is configured to supply, the Web server can tell instantly if it is equipped to service the request. If the server can satisfy the request, it locates whatever resource is being requested and returns it in a response, formatting the response according to the HTTP protocol.

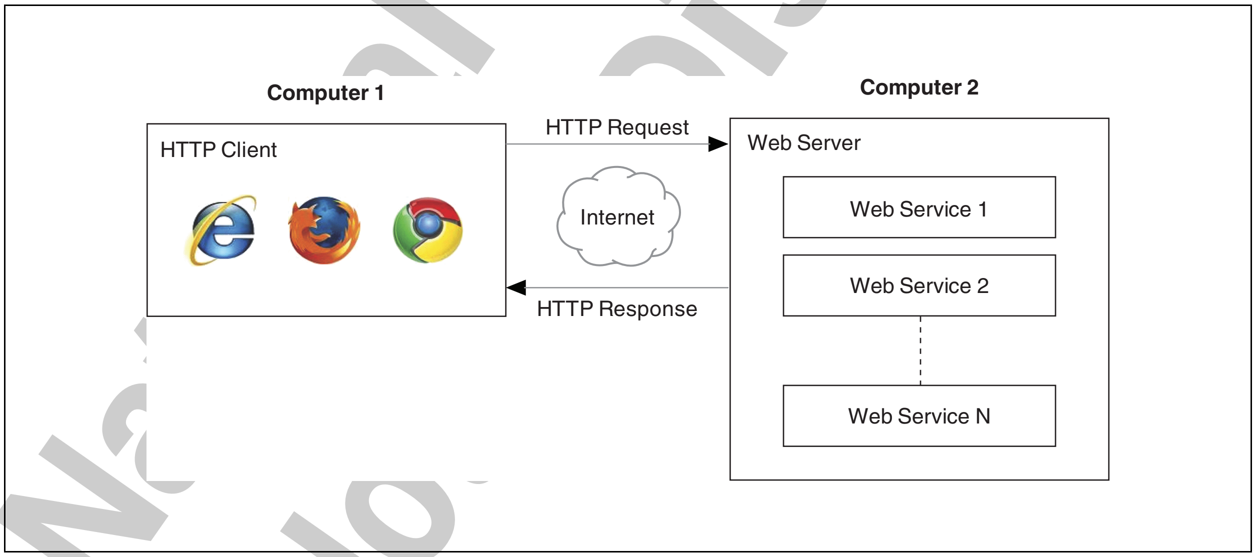


Figure 6-1. Web Server Example

A Web server can typically service multiple requests at one time. The largest Web servers can service thousands of simultaneous requests.

What is a Web Service?

Web services are Web-based APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services. Web services communicate using open protocols, such as HTTP.

LabVIEW Web services are special LabVIEW applications that can be loaded and run within the NI Application Web Server.

What is a Browser?

A browser is a software program that acts as an HTTP client to request resources, such Web pages, pictures, and so on, from HTTP servers. It graphically renders the resources to be read by a human, printed, and so on. Browsers are complex programs, relying on many different technologies including HTTP, SSL, XHTML, CSS, XML, audio/video codecs, and browser plugins (Flash, JavaScript, PDF, Silverlight).

What is HTTP?

HTTP protocol is an agreement between two communicating computers (typically a client and a server) for the formatting of request and response message sent across a network.

Hypertext Transport Protocol (HTTP) was invented in the late 80s, and first used globally during the birth of the World Wide Web in 1990.

HTTP was conceived as a way to access a distributed collection of documents on an internetwork of multiple computers. Any given document could be referenced (addressed) by a URL. This URL could be embedded within any document to refer or link to another document. If documents had these links embedded within them, they would allow a reader to start reading one document, then navigate to another, read through part of it, and navigate to yet another document. This is the basis of Hypertext.

This approach to distributing linked documents was so successful that it has become the basis for the global phenomenon we know today as the World Wide Web. All Web browsers use HTTP or a protocol derived from HTTP to communicate with Web servers.

What is a URL?

A Uniform Resource Locator (URL) is the address of a resource on a network. You can think of a URL as the name of a document on a network. That address could point to your corporate network or someplace on the other side of the planet via the World Wide Web.

Most URLs refer to files of some kind on a machine on a network, but URLs can also point to other resources on a network, such as database queries and command output.

The following is an example of a URL which addresses a file on the `www.mydomain.com` Web site:

```
http://www.mydomain.com:80/path/to/some/
resource?param1=something&param2=something_else
```

In this example, `http` identifies the protocol to be used on the network for communicating with the remote machine that has the resource. The colon specifies the end of the protocol string.

The double-slashes specify that host name follows (the name of the computer containing the target document). The host name in this case is `www.mydomain.com`.

The colon followed by a number specifies a port number on the computer where the target document can be found. Specific resources are served by servers that reside at particular port numbers within a given server computer. If the host name were an office building, the port number would be the number of a room within that building.

By convention, Web servers serve their resources over the HTTP protocol at port 80. This is so common that the protocol permits clients to skip the port number in the URL. Unless specified, the port number defaults to 80. When the port number is not specified, the colon and port number are both dropped from the URL. This means that the following two URLs are equivalent:

- `http://www.mydomain.com:80/aFile`
- `http://www.mydomain.com/aFile`

Following the host name (or port number if it is present) is a string beginning with a slash, including one or more elements, all separated by slashes. This string represents the pathname to a resource.

You can think of it like a pathname on the filesystem on the hard drive of modern PCs and laptops. The slash-delimited path is a representation of a hierarchical tree that sifts through hundreds or thousands of resources to identify a particular one.

After the resource path, there may be a question mark (?) followed by a list of parameters. These are optional for most URLs. If present, this parameter list is formed of key/value pairs separated by ampersands (&).

In the above example, there are two key/value pairs `param1` and `param2`, each with the value of that key specified after an equals (=). These parameters specify additional information that the Web server may use for various purposes while servicing the request.

Given what we've learned so far, we can now completely decode the URL given at the beginning of this section.

```
http://www.mydomain.com:80/path/to/some/
resource?param1=something&param2=something_else
```

protocol	`http`
host name	`www.mydomain.com`
port number	`80`
resource	`/path/to/some/resource`
request parameter 1	`something`
request paramter 2	`something_else`

Taken together, this is all the information that a Web server needs to service this particular request.

Static and Dynamic Resources

Static Resources

URLs may reference static documents. Issuing a request for a static document is like checking a book out of a library. The server has a collection of static documents, all addressed by URLs. It simply finds the book referenced by the URL, makes a copy of it, and gives it to the requester. The original resource remains on the server, unaffected. The client now has a snapshot copy of the resource as it existed at the time of the request. If the requester was a Web browser, the response may be a Web page, an image, or a file to be downloaded and stored to the remote user's hard drive.

Dynamic Resources

Requesters may also request dynamic content instead of static content. A dynamic resource specifies one that does not yet exist, but can be created specifically to match a particular request.

In this case, the Web server has to do more than simply find and return a copy of an original resource. It must examine the request and pass it on to other software, typically called Web applications or Web services. These programs examine the request much like the Web server did. Based on the request, they do whatever is required to come up with a response. They may search through databases, examine and report on other running programs, or perform any of an infinite variety of programmed functions.

Ultimately, once a Web application has completed everything necessary to satisfy the request, it formats a response according to HTTP and sends it back to the client.

What Can Web Services Do?

Web services can do anything any software program can do in response to an API call. For example, they can perform a service, such as look up and return values in a database, read a measurement device and return current values, or accept an uploaded file and store it on the file system. Web services can return a copy of a requested resource, such as the current value of some software variable or setting or a log file stored on disk.

Good Uses for Web Services

Web services are good for anything that fist the client-originated request/response model. The following situations are examples of good uses for Web services.

- Monitoring applications—one server, many clients retrieving the state of the application.
- Controlling applications—one server, many clients that might try to update the state of the application and/or hardware.
- Creating resources—uploading new applications or documents.
- Client machine has little or no processing capability. For example, a mobile phone or a computer that does not LabVIEW installed.

When Not to Use Web Services

Web services are not a suitable solution for the following situations.

- Continuous streaming of data—streaming does not map easily to request/response.
- When you need to closely monitor a rapidly changing value, or your application requires fast response to events or value changes.

  Unless you are running an HTTP server on the client, the server cannot initiate responses. It must wait for a request.

  The only way to monitor for rapidly-changing values is to poll rapidly repeatedly. Periodic polling is okay. Rapid polling is not because it uses bandwidth and target resources.

Key HTTP Characteristics

Because Web services are based on HTTP, you need to understand the characteristics of HTTP.

HTTP uses standard resource addressing; that is, services are requested by URLs. It allows resources to be requested in a standard fashion, regardless of the platform the server is running on.

HTTP allows the use of hyperlinks, which are embedded references to other resources. It is client-driven. Servers do not initiate requests.

HTTP is synchronous. Only one request can be outstanding at a time, and the client must wait for the request to complete before the next can be started on that same connection.

It uses a request/response model. It performs one service in response to one request. Requests and responses consist of two parts: the header, which is always plain text, and the content body, which can be any sequence of bytes. Requests and responses are encoded using HTTP message formatting.

Query Variables

The (URL) specifies a resource on a server. The URL is used with many protocols in addition to HTTP. An example of an HTTP URL is `http://www.example.com/`

A resource is a file or the dynamic output of a Web method residing on the server. A resource is accessed on a Web server using an HTTP request method.

Request variables are parameters passed to a resource within a URL in name/value pairs. Maps to front panel controls of Web method VIs using the terminal output type. Resource variables can be used with any HTTP request method.

The following is an example of request variable syntax:

```
?variable1=value1&variable2=value2
```

- `?`—indicates the beginning of the query string
- `=`—separates a name from its value
- `&`—separates key value pairs within the query string

URL Character Sets

URLs use a standard set of characters that include upper and lower case English characters, numbers, and the marks shown in Table 6-1.

Table 6-1. Allowed URL Marks

URL Marks	
-	_
.	!
~	*
)	(
‘	

All other characters must be escaped. The characters shown in Table 6-2 are allowed within URLs, but may not be allowed within a particular component of URL syntax (they are used as delimiters of the URL components).

Table 6-2. Reserved Characters

Reserved Characters	
;	/
?	:
@	&
=	+
$	,

Escaped characters are encoded as triplets. Use the percent character (%) followed by the two hexadecimal digits representing the octet code. For example, `%20` is the escaped encoding for the US-ASCII space character.

Common Escape Codes

All URLs use ASCII characters to define the location of a resource on a server. But there are cases where a resource is encoded with non-ASCII characters or characters reserved to format the URL. Some examples of non-ASCII characters include **ñ** and **?**. Some examples of reserved characters include **/**, **?**, and **&**. URLs that contain non-ASCII or reserved characters use escape codes to represent these characters.

Table 6-3. Common Escape Codes

ASCII Character	URL Encoding
space	%20
!	%21
“	%22
#	%23
$	%24
%	%25
&	%26
?	%3F

Escaping URLs in LabVIEW

LabVIEW provides a simple way to escape or unescape a URL. Use the Escape URL VI or the Unescape URL VI found on the **Data Communication»Protocols»HTTP Client»Utilities** palette to escape or unescape URLs as shown in Figure 6-2.

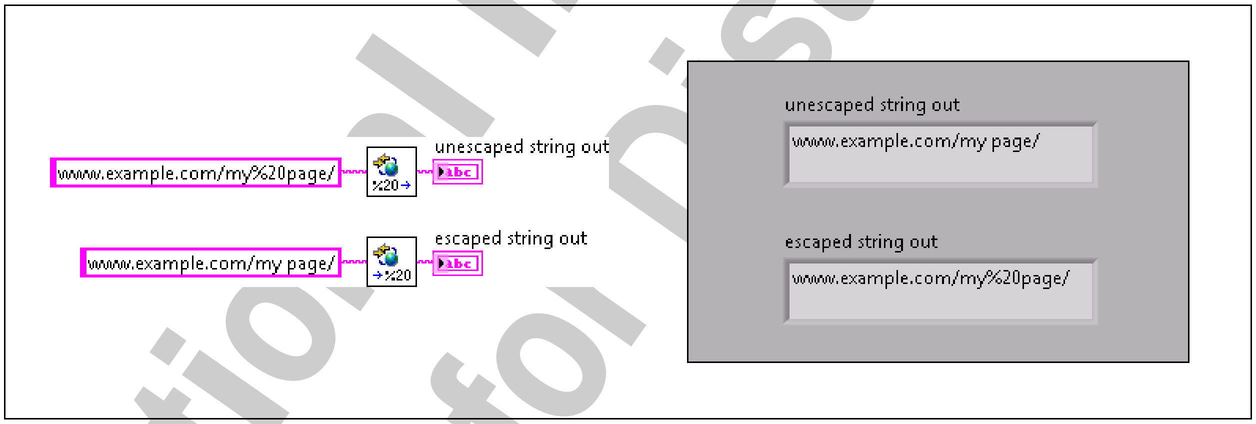


Figure 6-2. Escape or Unescape URLs in LabVIEW

B. LabVIEW Web Services Overview

Deploying a VI as a Web service in LabVIEW means that any Web-capable device can now communicate with your application. The Web interface does not require the LabVIEW run-time engine. Because this feature was designed from the ground-up to work on LabVIEW Real-Time targets, VIs not only can be deployed from your desktop, but they can also be deployed as Web services on hardware such as CompactRIO and PXI. Because it uses the standard HTTP Web protocol, users can use any client-side technology, such as HTML, JavaScript, and Flash, to develop user interfaces and interpret standard data formats such as XML.

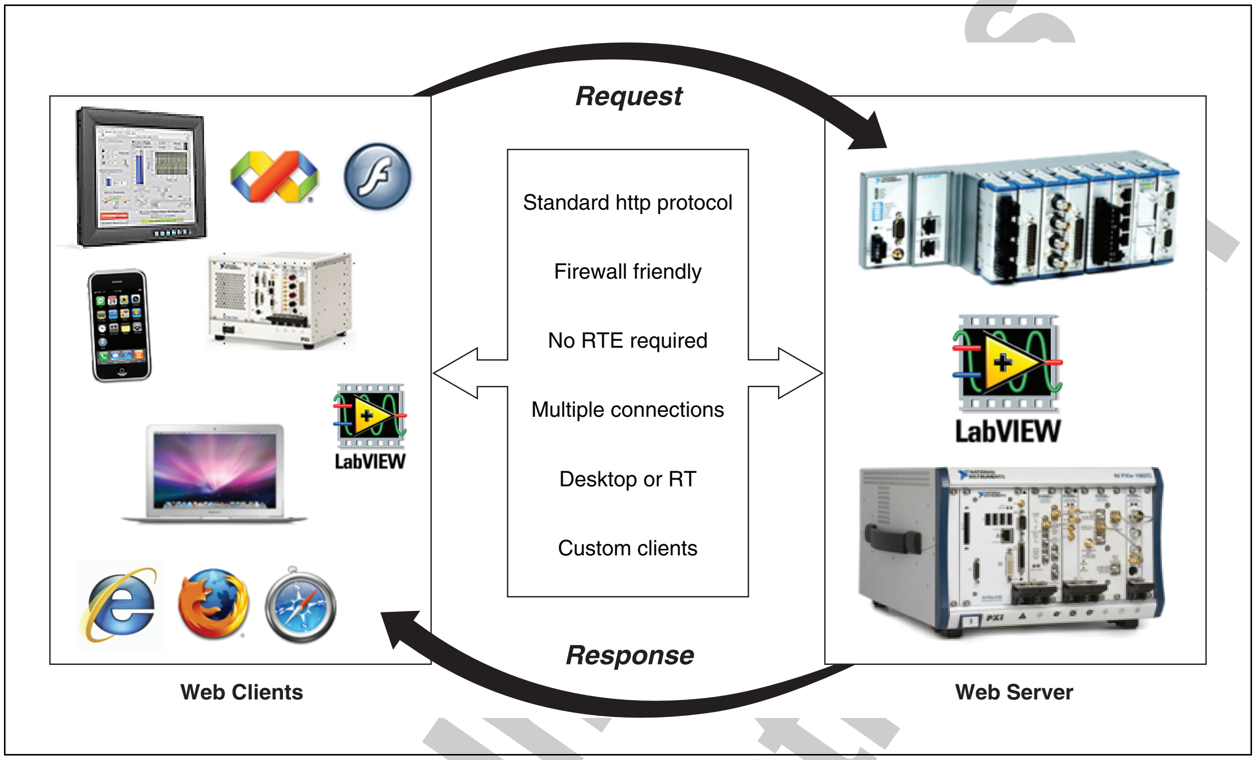


Figure 6-3. LabVIEW as a Web Server

LabVIEW Web Services Protocol Layers

LabVIEW Web services use RESTful Web service architecture, which is based upon the Representational State Transfer (REST) architecture. RESTful provides a lightweight approach accessible to a wide variety of HTTP-enabled clients and does not require complex message parsing.

Note The alternative to REST is SOAP, which is far more complex and not appropriate for RT targets or hardware that has less horsepower.

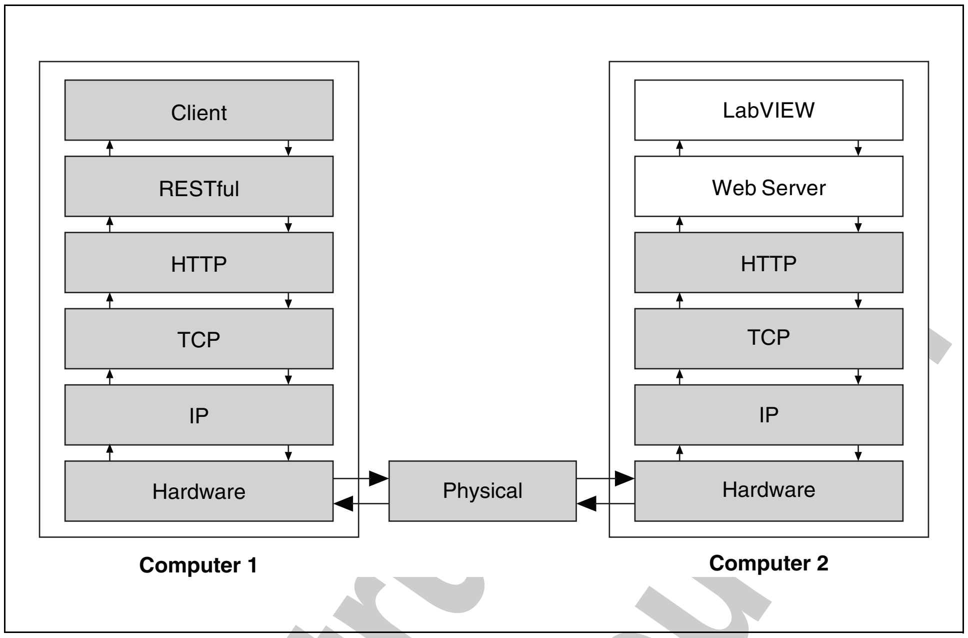


Figure 6-4. LabVIEW Web Services Protocol Layers

LabVIEW as a Web Server

LabVIEW Web services are special LabVIEW applications that can be loaded an run within the NI Application Web Server, which invokes Web service methods. Multiple Web services can exist on the system simultaneously and each Web service can contain multiple methods. Each Web method corresponds to a Web method VI or static document (such as an HTML file).

You use a LabVIEW project to build and deploy a Web service. No NI software is required on the HTTP client machine. LabVIEW can act as the client, but the client also can be any popular browser, such as Internet Explorer, Mozilla Firefox, or Google Chrome.

Use Application Builder to define and compile Web services into files with the `.lvws` extension.

Web Services File System and .LVWS Files

LabVIEW Web services use a `.lvws` file to deploy Web services to the Application Web server. After you build a Web service, LabVIEW packages all necessary files into a zip file with the `.lvws` extension. When you deploy that Web service, LabVIEW deploys the `.lvws` file to the target Web server, which can be located on the same system or another hardware target. LabVIEW then unzips the file and installs the Web service components into a unique directory on the target Web server.

The deployed Web service root location differs based on operating system.

(Windows 7/Vista) `C:\ProgramData\National Instruments\WebServices\NI`

(Windows XP) `C:\Documents and Settings\All Users\Application Data\National Instruments\WebServices\NI`

(ETS and VxWorks) `/c/tmp/webservices/`

What Makes a Web Service RESTful?

RESTful Web Services are based on the following four verbs:

- **GET**—Give me the resource named.
- **POST**—Take this from me and do something with it.
- **PUT**—I changed what you gave me earlier. Overwrite your version with mine.
- **DELETE**—Destroy this resource.

GET

Use GET to retrieve any data, such as a current variable or status in your application, the current value of a temperature sensor, or a copy of the most recently captured waveform. GET requests must not change the state of the program. They are safe to call over and over without operational side effects.

POST

Use POST when you want to send new data to the server and have it store a new copy of that data. For example, use POST to store a new measurement to a database or log file, submit a complex query to be processed, or upload a file or program to be instantiated. POST changes commonly alter the server state.

PUT

Use PUT to update a particular named resource with a new value. For example, PUT can flip a control from off to on or update a set point. It is common to read a value with GET, modify it, then write it back with PUT using the same resource name. PUT operations commonly change the state of the server or application.

DELETE

Use DELETE for anything that maps logically to DELETE, DESTROY, STOP, CANCEL, TERMINATE, and so on. DELETE usually alters the server or application state.

GET Request Using Terminals

To receive data in a Web method VI, you must assign controls to terminals on the connector pane of the Web method VI. Web clients can then send values to a Web method VI using the label of the control as the identifier. To return data from a Web method VI to a Web client, you must assign indicators to terminals on the connector pane of the Web method VI.

Labels used to send and receive values in Web services can contain only letters, numbers, hyphens, and underscores available in the US ASCII character set. Labels cannot contain spaces or special characters.

Supported Data Types for terminal output

Many people are familiar with XML, Text and HTML. JavaScript Object Notation (JSON) is another option. JSON is a lightweight, text-based open standard designed for human-readable data interchange. It is derived from the JavaScript programming language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for most programming languages.

When you have created a VI that you want to access using Web services, you must first build and deploy the Web service. Refer to the section for details about creating a build specification for your LabVIEW Web service.

To practice the concepts in this section, complete Exercise 6-1.

Accept Form Data Using POST

In this section, you will learn how you can use an HTML form to send and receive data from a VI using the POST method to pull data from a VI.

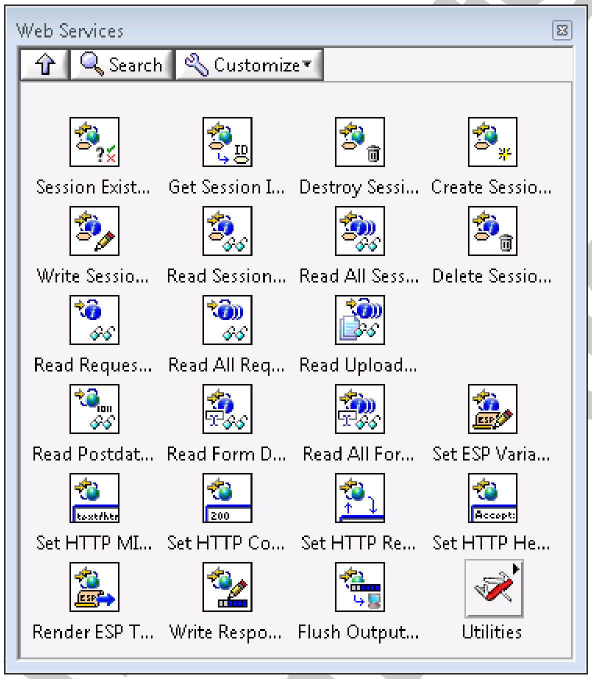


Figure 6-5. LabVIEW Web Services Palette

In Exercise 6-1, you used terminals to pass data to and from the Web method VI. The Web Services palette, shown in Figure 6-5, allows for additional functionality and alternative ways to communicate with Web Method VIs, including:

- Server side scripting—similar to PHP, but it is called ESP
- Sessions—data stored in cookies on the client can be accessed and set.
- Returning data from an HTML form can be parsed by VIs on this palette.
- Streaming any MIME data type, including images, videos, application and text.

In Exercise 6-2, you will use the Read Form Data VI to read data from a html form. This is an alternative to using the connector pane and terminals.

The Read Form Data VI reads a single form data value associated with the current HTTP request. Use this VI for requests that use the multipart/form-data encoding type.

The Read Form Data VI includes the following inputs and outputs.

- **httpRequestID**—is an identifier that VIs on the Web Services palette use to reference a specific HTTP request. The httpRequestID works similarly to a refnum in LabVIEW, allowing you to wire together VIs from the Web Services palette within your application. You must create a new httpRequestID control or wire an existing value from the httpRequestID out terminal of another VI. You also must assign the original httpRequestID control to the connector pane of the top-level VI that contains the VI(s) from the Web Services palette.
- **key**—specifies the name of the form data element for which to return a value. For example, if an HTML form submits the post data, x=2&y=3, you can enter either x or y as the key to retrieve the respective value.
- **error in**—describes error conditions that occur before this node runs. This input provides standard error in functionality.
- **httpRequestID out**—is an identifier that VIs on the Web Services palette use to reference a specific HTTP request. Use httpRequestID out to wire together VIs from the Web Services palette within your application. You also must assign the original httpRequestID control to the connector pane of the top-level VI that contains VI(s) from the Web Services palette.
- **value**—returns the value of the specified form data element.
- **error out**—contains error information. This output provides standard error out functionality.

Security and Encryption

Security

- LabVIEW Web services support user authentication with role-based authorization.
- Individual LabVIEW Web methods can be protected with a list of permissions.
- The security extension in LabVIEW's Web Based System Configuration and Monitoring allows permissions to be assigned to roles, and roles assigned to users.
- Only authenticated, authorized users with required permissions can invoke a protected Web method.

Encryption

- LabVIEW's Application Web Server supports SSL, the industry standard encryption technology used to protect shopping and banking transactions on the Internet.
- SSL support is standard on desktop systems, but it is an optional installation for RT targets

Note SSL is available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

To practice the concepts in this section, complete Exercise 6-2.

Modify Response Content Type

A common question is "how would I grab the image of a front panel object and stream it to the browser?" To accomplish this, you modify the response content type to be an image. In this section you learn how you can change the response type to an image to send back snapshots of VI front panels or complex controls.

MIME, or content-type, is an identifier for file formats on the Internet. It comes after the name of the HTTP header that specifies the type of the HTTP Message Body.

MIME types include a two-part identifier, type and subtype, and one or more optional parameters. The following are common MIME types and subtypes:

- Text types
  - text/html—HTML documents
  - text/css—cascading Style Sheets
  - text/csv—comma-separated values
  - text/plain—textual data
  - text/xml—extensible Markup Language
- Image types
  - image/gif—GIF image
  - image/jpeg—JPEG JFIF image
  - image/png—Portable Network Graphics
  - image/svg+xml—SVG vector image
  - image/tiff—Tag Image File Format

Customizing Response Header and Body

When stream mode is selected, the Web services runtime does not generate a response for you. It allows you to manipulate the response body and the response headers.

Stream mode allows you to implement your own customized presentation logic. It lets you choose how you want to format the information you send back to the client. Use this method if you are using the Write Response VI or the Render ESP Template VI

To practice the concepts in this section, complete Exercise 6-3.

Build and Deploy a Web Service

Use a LabVIEW project to organize the VIs and other files for your Web service. To use LabVIEW Web services, you must build and deploy a Web service using the Build Specification in the LabVIEW Project Explorer.

Web Services Properties

From the Project Explorer window, right-click **Build Specifications** and select **New»Web Service (RESTful)** from the shortcut menu to display this dialog box. You also can right-click a Web service specification name under Build Specifications and select **Properties** from the shortcut menu, or you can double-click the Web service specification name. If you rebuild a given specification, LabVIEW overwrites the existing files from the previous build that are part of the current build.

Use the Web Services Properties dialog box to access and configure settings for LabVIEW Web services. This topic assumes familiarity with the Web services introduction and the Building a LabVIEW Web Service Application tutorial in the *LabVIEW Help*.

The Web Service Properties dialog box includes the following pages, which you use to configure the settings for the build:

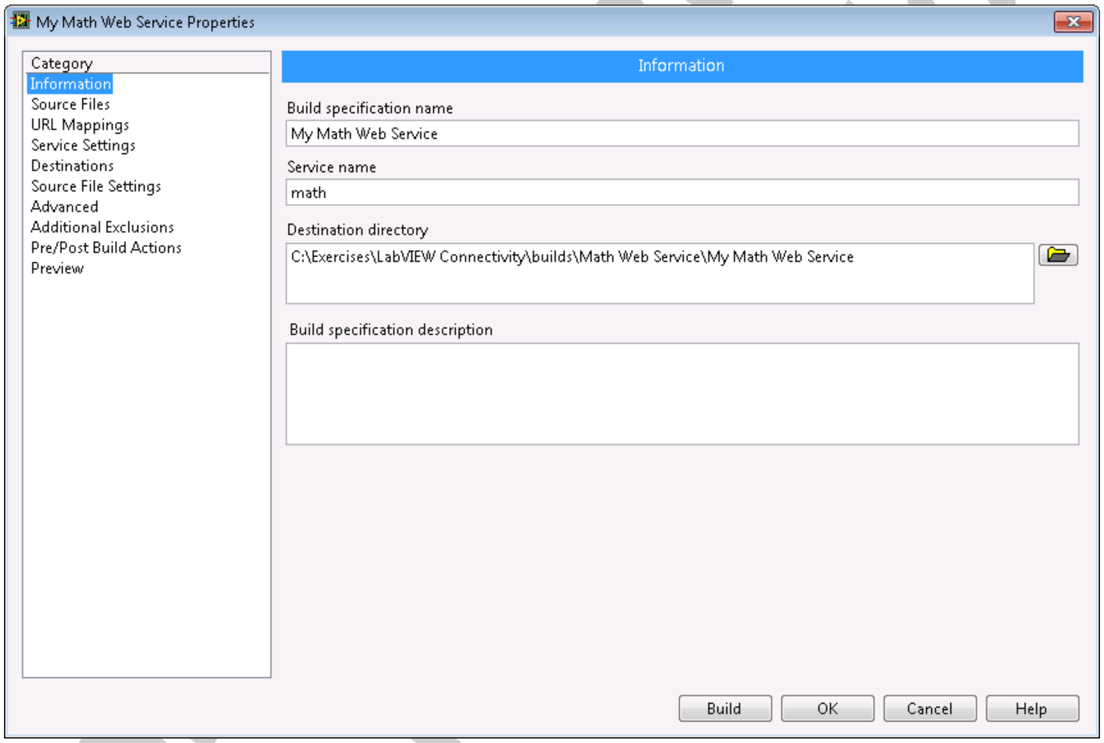
- Information
- Source Files
- URL Mappings
- Service Settings
- Destinations
- Source File Settings
- Advanced
- Additional Exclusions

- Pre/Post Build Actions
- Preview

For additional information about all the pages in the Web Service Properties dialog box, refer to the *Web Service Properties Dialog Box* topic of the *LabVIEW Help*.

Information

Use this page of the Web Service Properties dialog box to name the Web service and select the location to build the Web service.



Build specification name: My Math Web Service

Service name: math

Destination directory: C:\Exercises\LabVIEW Connectivity\builds\Math Web Service\My Math Web Service

Build specification description:

Figure 6-6. Web Service Properties—Information Page

This page includes the following components:

- **Build specification name**—Specifies a unique name for the build specification. The name appears under Build Specifications in the Project Explorer window.
- **Service name**—Specifies the name for the Web service. The service name must conform to standard URL syntax because the service name is part of the URL that HTTP clients use to access the Web service. The Application Builder also uses the service name and automatically appends a `.lvws` extension when creating the Web service file.

- **Destination directory**—Specifies the location to save the build on the local computer. You can enter a path or use the Browse button to navigate to and select the location.
- **Build specification description**—Displays information about the build specification. You can view and edit the description on this page only.

Source Files

Use this page of the Web Service Properties dialog box to add and remove files to build into the Web service.

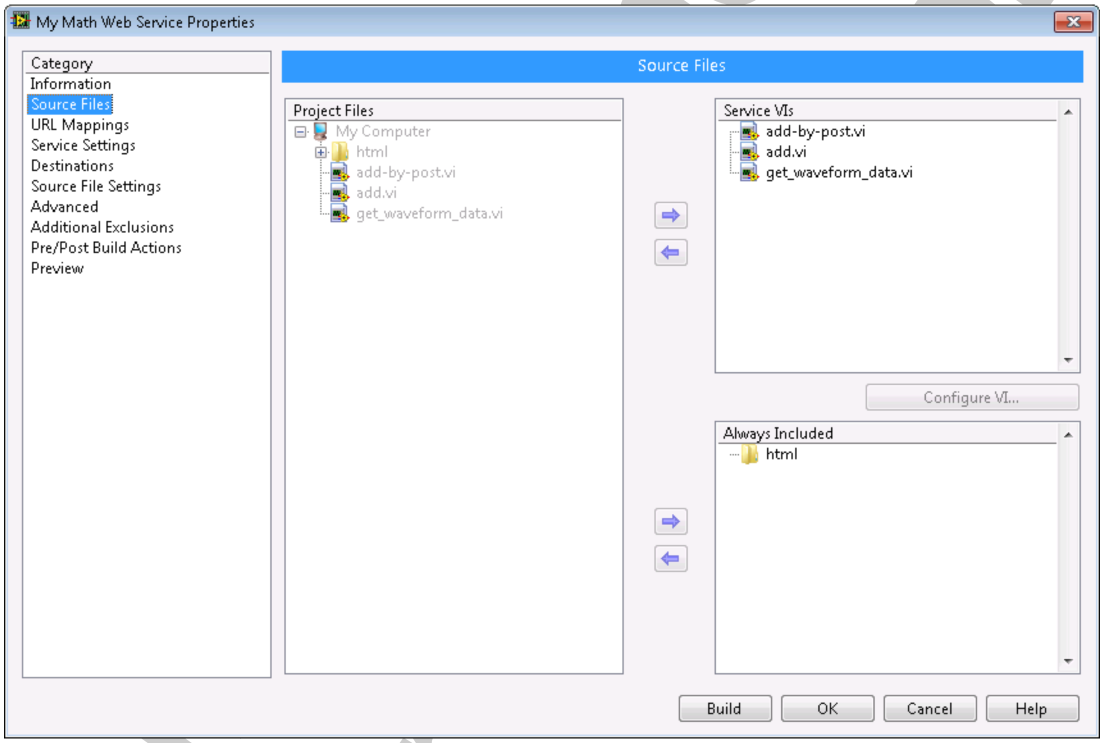


Figure 6-7. Web Service Properties—Source Files

This page includes the following components:

- **Project Files**—Displays the tree view of items under My Computer in the Project Explorer window. Click the arrow buttons next to the Service VIs and Always Included listboxes to add selected files from Project Files to those lists or to remove selected files from the listboxes. The Configure RESTful VI dialog box only appears when you move an individual VI to the Service VIs listbox. Otherwise, if you move multiple VIs to the Service VIs listbox, each VI uses default configuration. You must select the individual VI and click the Configure VI button to modify the configuration.

- **Service VIs—**Specifies the service VIs, also known as Web method VIs, which users can access as functions of the Web service. You must include at least one Web method VI. Click the arrow buttons next to the Service VIs listbox to add selected Web method VIs from the Project Files tree or to remove selected Web method VIs from the Service VIs listbox.

  When you add a Web method VI to the Service VIs listbox, LabVIEW creates a default URL mapping on the URL Mappings page of the Web Service Properties dialog box.
- **Configure VI**—Activates when you select a Web method VI in the Service VIs listbox. Click the button to open the Configure RESTful VI dialog box, which you can use to define the parameters of a Web method VI.
- **Always Included**—Specifies the dynamic VIs, support files, and static content always to include in the Web service. Click the arrow buttons next to the Always Included listbox to add selected files from the Project Files tree or to remove selected files from the Always Included listbox. When you add a folder to the listbox, you add all items in the folder and cannot remove individual items.

  Dynamic VIs are VIs that LabVIEW dynamically calls through the VI Server. Support files are non-VI files, such as drivers, text files, help files, and .NET assemblies that the Web service uses.

You cannot move the following files to the Service VIs or Always Included listbox:

Service VIs

- Polymorphic VIs
- Controls
- Private data controls
- Files that are not VIs, such as text, image, or `.mnu` files
- Library files, such as LabVIEW classes, or XControls
- Global variables
- VIs with Dynamic Dispatch Terminals

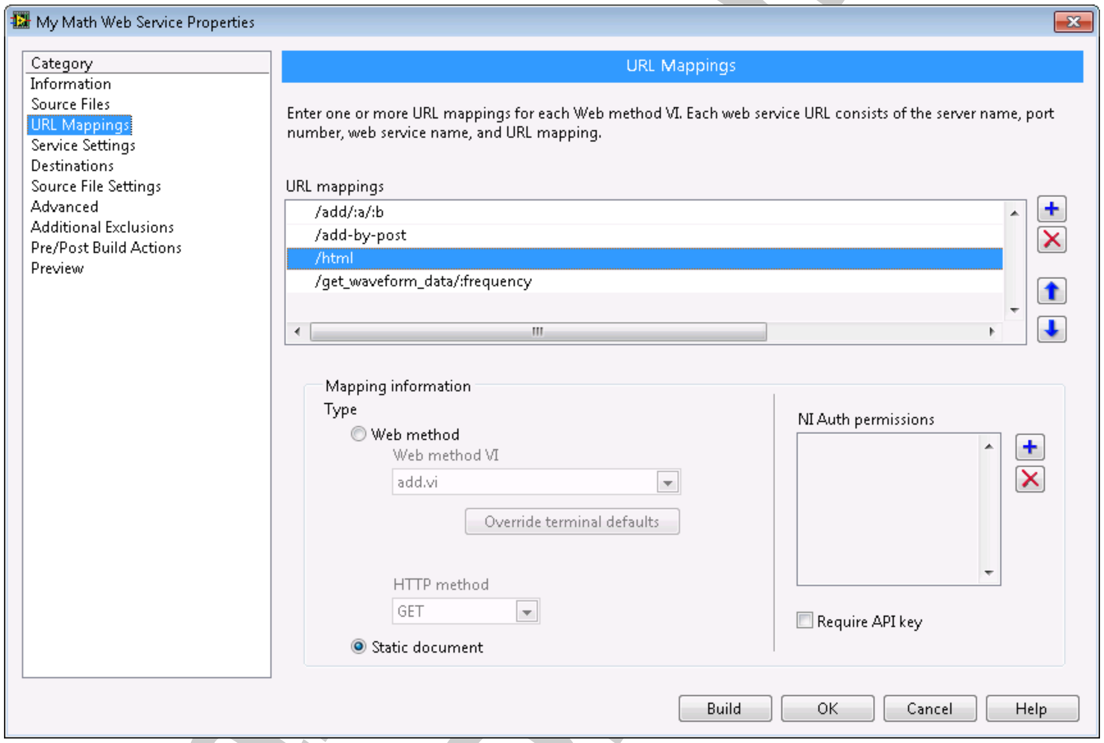
Always Included

- Private data controls

If you move a library or LabVIEW class to the Always Included listbox, LabVIEW labels all items in the library or class referenced. You can still designate any of the individual items as exported VIs. However, if you move any part of an XControl to the Always Included listbox, LabVIEW dims the related XControl files and includes the related files in the build. You cannot designate any of the related items as service VIs.

URL Mappings

Use this page of the Web Service Properties dialog box to configure URL mappings for a LabVIEW Web service.



Enter one or more URL mappings for each Web method VI. Each web service URL consists of the server name, port number, web service name, and URL mapping.

URL mappings
/add/:a/:b
/add-by-post
/html
/get_waveform_data/:frequency

Figure 6-8. Web Service Properties—URL Mappings Page

URL mappings allow a Web client to specify the Web method VI and input parameters to send data as an HTTP request. When you create a URL mapping, you associate a string value with a unique Web method VI, HTTP method, and optional input parameters. Web clients then use the URL mapping string as part of the URL that the client uses to exchange data with the Web service.

Note LabVIEW generates a default URL mapping string when you add a Web method VI to the Web service on the Source Files page of the Web Service Properties dialog box.

This page includes the following components:

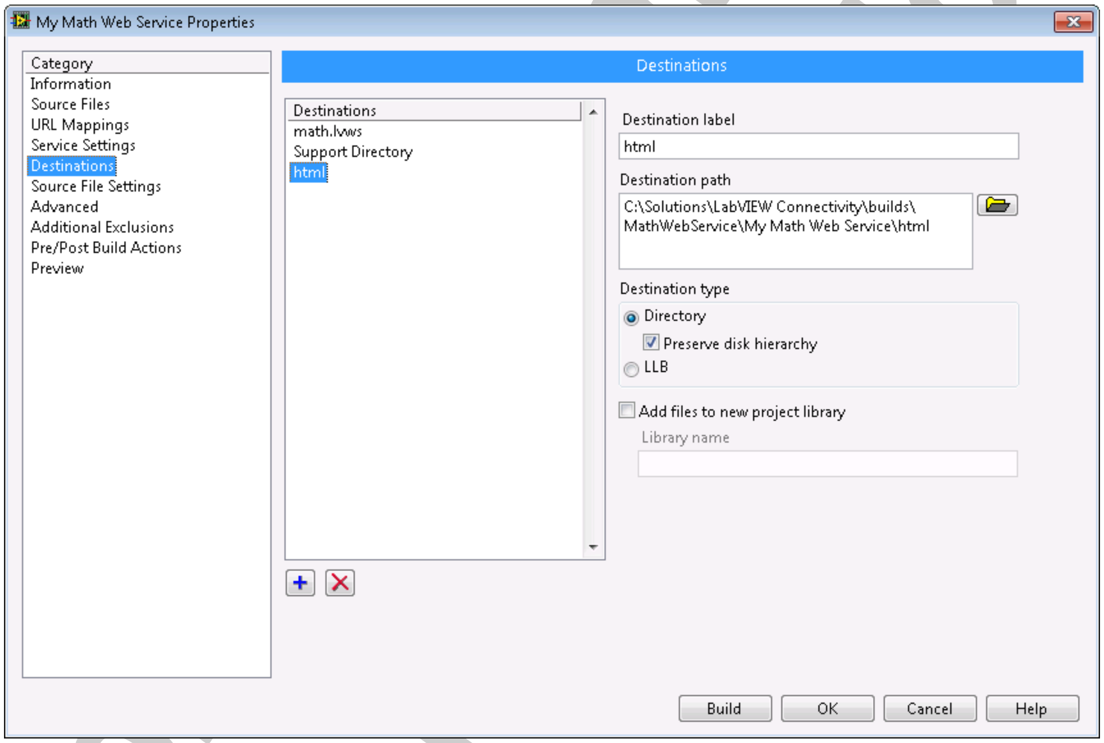
- **URL mappings**—Lists any configured URL mappings.
  - **Add URL Mapping**—Adds a new URL mapping to the Web service.
  - **Remove URL Mapping-**—Removes the selected URL mapping from the Web service.
  - **Move URL Mapping up**—Moves the selected item up on the list of URL mappings to establish URL mappings precedence.
  - **Move URL Mapping down**—Moves the selected item down on the list of URL mappings to establish URL mappings precedence.
- **Mapping information**—Displays the configuration options for the selected URL mapping.
  - **Type**—Specifies whether the URL mapping activates a Web method VI or displays static content.
    - **Web method**—Specifies the selected URL mapping to call a specific Web method VI.
      - **Web method VI**—Specifies the Web method VI to associate with the selected URL mapping. All available Web method VIs specified in Service VIs on the Source Files page appear in the pull-down menu.
      - **Override terminal defaults**—Displays the Override terminal defaults dialog box for the selected URL map. You must configure input parameters for the selected Web method VI to view this dialog box.
      - **HTTP method**—Lists HTTP methods available to access the selected Web method VI.
    - **Static document**—Specifies the selected URL mapping to display static content.
  - **NI Auth permissions**—Specifies a list of permissions that a user or group must have to exchange data using the selected URL mapping. You can establish permissions and assign those permissions to users and groups using the Security Configuration page when you configure a target using a Web browser.
  - **Add permission**—Adds a new permission.
  - **Remove permission**—Removes the selected permission.
  - **Require API key**—Specifies whether the selected URL mapping uses API key security.

Destinations

Use this page of the Web Services Properties dialog box to configure the destination settings and add destination directories to a Web service.

Note Destinations do not appear in the output directory. The Application Builder creates the destinations on the specified target when it deploys the Web service.

Caution The file path of a deployed Web service, including the directory hierarchy that results from configurations on this page, cannot exceed 256 characters.



Destination label: html

Destination path: C:\Solutions\LabVIEW Connectivity\builds\MathWebService\My Math Web Service\html

Destination type: [✓ Directory] LLB

[✓] Preserve disk hierarchy

[ ] Add files to new project library

Figure 6-9. Web Service Properties—Destinations Page

This page includes the following components:

- **Destinations**—Specifies the destination directories in which you want to include the files that the build generates.

Note To avoid receiving an error during the build process, ensure that file paths for the destination directory, including the filename, are less than 255 characters.

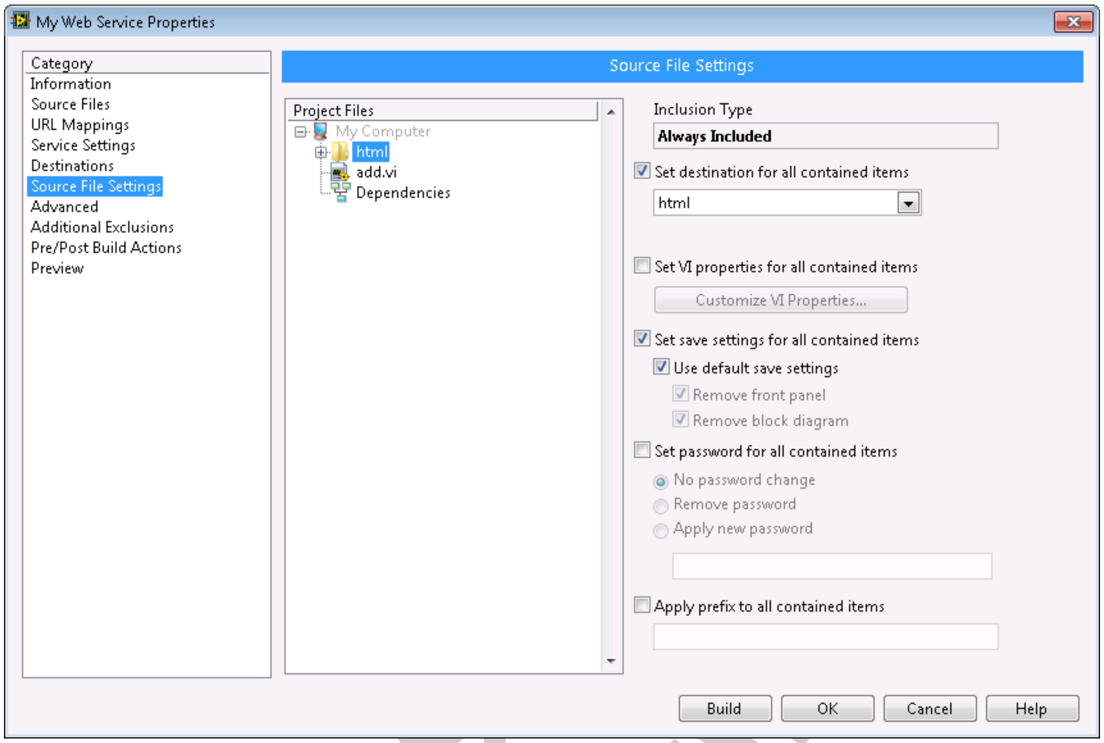
- Click the Add Destination and Remove Destination buttons to add and delete directories from the list.
- **Add Destination**—Adds a custom destination directory to the Destinations listbox.

- **Remove Destination—**Removes the selected destination directory from the Destinations listbox. You cannot remove any default destination directory that is already present in the Destinations list.
- **Destination label—**Specifies the name that the Application Builder uses for the directory selected in the Destinations listbox. You can select these names as destinations on the Source File Settings page in the pull-down menu. You cannot change the Destination label setting for the destination or support directories. If you do not change the text in the Destination path text box, editing the destination label updates the text in the Destination path text box.
- **Destination path—**Specifies the path to the directory or LLB you select in the Destinations listbox. Click the Browse button to navigate to and select a path. If you change the path of the destination directory, any destinations that are subdirectories automatically update to reflect the new path.
- **Destination type—**Specifies the destination type of the item you select in the Destinations listbox. You cannot change this setting for applications or support directories.
  - **Directory—**Specifies that the destination is a directory.
- **Preserve disk hierarchy—**Preserves the disk hierarchy of the files targeted to this destination directory.
  - **LLB—**Specifies that the destination is an LLB.
- **Add files to new project library—**Specifies that you want to add files you move to the selected destination to a new project library.
  - **Library name—**Name of the new project library to which LabVIEW adds the files.

Source File Settings

Use this page of the Web Service Properties dialog box to edit destinations and properties for the files included in a Web service. LabVIEW enables the options in this page only if the item you select in the Project Explorer tree supports the option. You can apply settings to all the items under Dependencies, but not to individual items under Dependencies.

Note If you plan to distribute a Web service that uses shared variables, do not include the `.lvlib` file in the Web service. Change the Destination of the `.lvlib` file to a destination outside the Web service.



Inclusion Type: Always Included

[✓] Set destination for all contained items: html

[ ] Set VI properties for all contained items — Customize VI Properties...

[✓] Set save settings for all contained items
- [✓] Use default save settings
  - [✓] Remove front panel
  - [✓] Remove block diagram

[ ] Set password for all contained items
- (●) No password change
- ( ) Remove password
- ( ) Apply new password

[ ] Apply prefix to all contained items

Figure 6-10. Web Service Properties—Source File Settings

This page includes the following components:

Note The following options appear only when you select a folder from the Project Files tree. The settings apply to all items in the selected folder. You cannot specify settings for individual items in the folder. The Inclusion Type indicator displays the inclusion type for the folder.

- Set destination for all contained items
- Set VI properties for all contained items
- Set save settings for all contained items
- Set password for all contained items
- Apply prefix to all contained items

Caution The file path of a deployed Web service, including the directory hierarchy that results from configurations on this page, cannot exceed 256 characters.

- **Project Files**—Displays the tree view of items under My Computer in the Project Explorer window.
- **Inclusion Type**—Displays how LabVIEW includes the item in the build. This option corresponds to the inclusion type you selected in the

Source Files page. For example, a VI in the Always Included list appears as Always Included.

If you select a folder, the inclusion type corresponds to the inclusion type of the items inside the folder.

- **Service VI**—Includes the item in the build as a service VI.
- **Always Included**—Includes the item in the build.
- **Include if referenced**—Includes the item in the build if another item references it.

- **Destination**—Sets the destination for the selected item. LabVIEW enables this option if you have not designated the item as a startup VI. The names in the Destination pull-down menu correspond to the options in the Destination label text box on the Destinations page. The default destination is Same as caller and LabVIEW places the item in the same destination as the caller.
  - **Make top level in LLB**—Appears when you select a destination that is an LLB.

    Place a checkmark in the Make top level in LLB checkbox if you want the selected VI to be the top level item in the LLB.
- **Customize VI Properties**—Displays the VI Properties dialog box. Use the dialog box to specify the properties for the selected VI. By default, LabVIEW uses the property settings configured in the VI. Any settings you configure in the VI Properties dialog box override any settings you configured in the Customize Window Appearance dialog box. LabVIEW dims this option for items other than VIs.
- **Use default save settings**—Saves the VIs using default save settings. The default save setting for the VIs you add to the Service VIs and Always Included listboxes on the Source Files page is to remove the block diagram. The default for all other VIs is to remove the block diagram and the front panel. Remove the checkmark from this checkbox to change the default settings for each item you select in the Project Files tree.
  - **Remove front panel**—Removes the front panel from a VI in the build. Removing the front panel reduces the size of the application or shared library. If you select yes, LabVIEW removes the front panel, but Property Nodes or Invoke Nodes that refer to the front panel might return errors that affect the behavior of the source distribution. LabVIEW enables this option if you remove the checkmark from the Use default save settings checkbox.
  - **Remove block diagram**—Removes the block diagram from a VI in the build. LabVIEW enables this option if you remove the checkmark from the Remove front panel checkbox. If you remove the front panel, you also remove the block diagram. As a result, if

you place a checkmark in the Remove front panel checkbox, a checkmark automatically appears in the Remove block diagram checkbox.

- **No password change—**Specifies for LabVIEW to not prompt for a password if you use the VI in a build. LabVIEW also does not modify or apply a password you previously applied to the VI.
- **Remove password**—Removes the password you previously applied to a VI or library. Prior to removing the password, LabVIEW prompts you to enter the current password during the build.
- **Apply new password—**Applies the password you supply in the text box to a VI or library. Prior to applying the new password, LabVIEW prompts you to enter the current password during the build. Enter the password in the text box below the Apply new password option.
- **Rename this file in the build**—Appears when you select a file in the Project Files list. Renames the selected file. Enter the new name of the file in the text box.
- **Set destination for all contained items—**Appears when you select a folder in the Project Files tree. Place a checkmark in the checkbox if you want to set the destination directory for the items in the selected folder.

Note LabVIEW places an item set to Same as caller in the directory of the caller. If you set an item to Same as caller and two different callers are in different directories, LabVIEW places the item in the same directory as the build.

- **Set VI properties for all contained items**—Appears when you select a folder in the Project Files tree. Place a checkmark in the checkbox if you want to set the VI properties for the items in the selected folder. When you place a checkmark in the checkbox, LabVIEW enables the Customize VI Properties button.
- **Set save settings for all contained items—**Appears when you select a folder in the Project Files tree. Place a checkmark in the checkbox if you want to set the save settings for the items in the selected folder.
- **Set password for all contained items**—Appears when you select a folder in the Project Files tree. Place a checkmark in the checkbox if you want to configure the password for the items in the selected folder.
- **Apply prefix to all contained items—**Appears when you select a folder in the Project Files tree. Place a checkmark in the checkbox to enter a prefix in the text box and rename all items in the folder by applying the prefix to the existing names.

C. LabVIEW as an HTTP Client

So far, you have learned to use LabVIEW as a server. The client was a Web browser. A Web browser is a HTTP client that also has an engine for rendering text and graphics. The HTTP client uses a HTTP request method to receive or submit data to a Web server. The Web server returns data to the Web browser. The Web browser then renders this data on screen for the user.

In this section, you will learn how to use LabVIEW as a HTTP client.

WSDL: Web Services Description Language

Use the Import Web Service wizard to transform a Web service into a library of VIs that you then can use to access the Web service methods and properties. To use the Import Web Service wizard, you must provide a valid URL to a Web Service Description Language (WSDL), and the WSDL must validate correctly with ASP.NET.

WSDL is an XML-formatted language used to describe a Web service and its functionality.

In this lesson, you will learn about the HTTP Client VIs which are used to build a Web client that interacts with servers, Web pages, and Web services. The HTTP Client VIs use the RESTful architecture. These VIs also can interact with LabVIEW Web services.

HTTP Client VIs

An HTTP Client is software that implements the standardized HTTP request methods developed by the Internet Engineering Task Force and the World Wide Web Consortium. All Web browsers are composed of two basic components: an HTTP client and an engine for rendering text and graphics. The HTTP client requests resources from a Web server, and this data is rendered on screen for the user.

The LabVIEW HTTP Client API allows you to create LabVIEW applications to interact with Web servers and Web services. The LabVIEW HTTP Client API includes VIs for executing all common HTTP methods including GET, HEAD, POST, PUT, and DELETE. The HTTP Client also allows for Secure Socket Layer (SSL) communication, access to authenticated resources, and custom headers.

Note SSL support for the HTTP Client requires a license for the Internet Toolkit.

The HTTP Client palette is found in **Functions»Data Communication» Protocols»HTTP Client**.

The following definitions are important for understanding and using HTTP client VIs in LabVIEW.

URL—specifies a resource on a server. The URL is used with many protocols in addition to HTTP. An example of an HTTP URL is `http://www.example.com/`.

Resource—a file or the dynamic output of an Web method residing on the server. A resource is accessed on a Web server using an HTTP request method.

Request Variable—parameters passed to resource within a URL in name/value pairs. Maps to front panel controls of Web method VIs using the terminal output type. Can be used with any HTTP request method

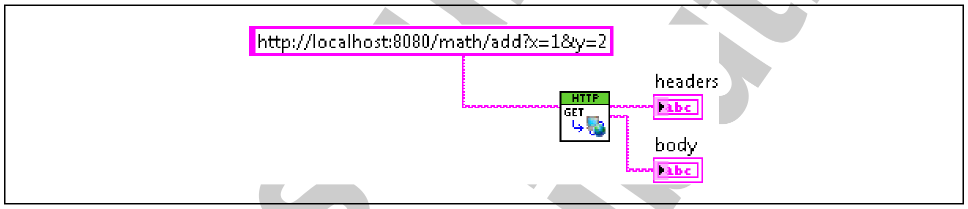
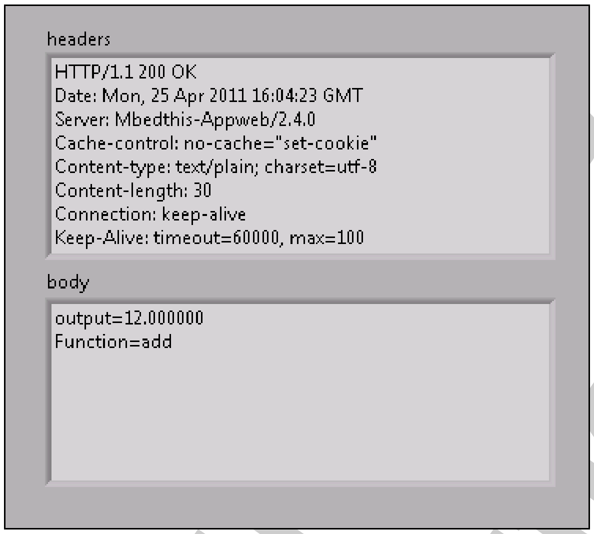


Figure 6-11. URL Passing Request Variables with the GET Request Method.

Header—contains a series of strings that define the request or response parameters of the current HTTP session. Figure 6-12 shows the response header sent by the Web server to an HTTP client VI in LabVIEW. A request header (not shown) is sent to the Web server by the HTTP client. The HTTP client API in LabVIEW is designed such that for most standard requests the user does not need to define the request header. If the user does need to use a custom header, this can be added with the AddHeader VI.

Body—contains the HTML, JavaScript, XML, and so on provided by the currently accessed resource. This data can be almost anything, but the most common types of data are either plain text or a variant of plain text including HTML, XML, JavaScript, and CSS. The type of data that is returned is implemented by the designer of the Web server or Web service.



```
HTTP/1.1 200 OK
Date: Mon, 25 Apr 2011 16:04:23 GMT
Server: Mbedthis-Appweb/2.4.0
Cache-control: no-cache="set-cookie"
Content-type: text/plain; charset=utf-8
Content-length: 30
Connection: keep-alive
Keep-Alive: timeout=60000, max=100
```

```
output=12.000000
Function=add
```

Figure 6-12. The Response Header and Body Returned by a Request Method in LabVIEW

HTTP Request Methods

The HTTP Client API supports all common HTTP request methods. Request methods fall into two categories: safe, and not-safe. Safe request methods should only return information from the server, and should not result in any change of state or the addition or removal of a resource on the server. Not-safe methods are the opposite and can, but not always, result in a change of state on the server or the addition or removal of a resource. It is the responsibility of the Web service developer to design Web methods that follow these design practices.

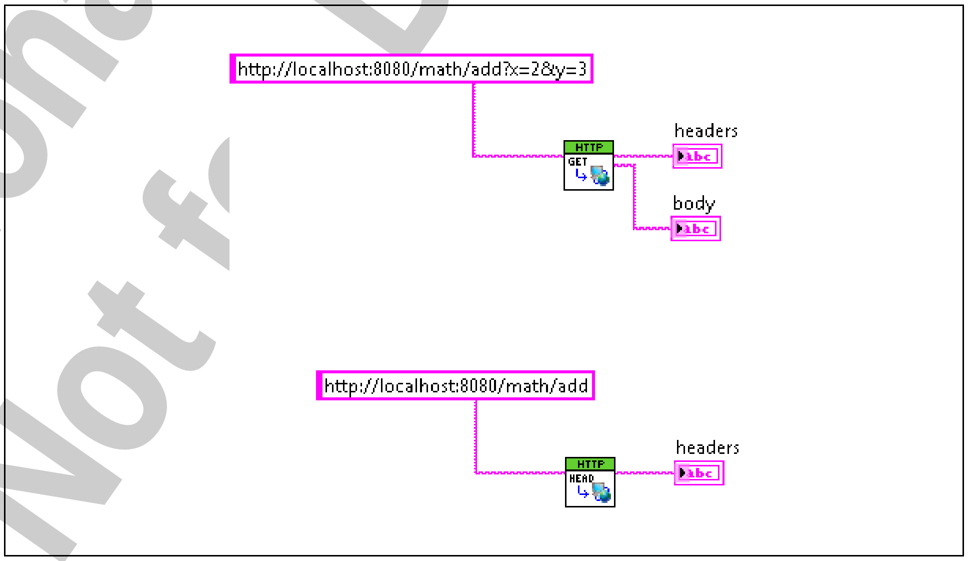

http://localhost:8080/math/add?x=2&y=3
HTTP GET
headers
body
http://localhost:8080/math/add
HTTP HEAD
headers

Figure 6-13. Safe HTTP Request Method VIs

The GET and HEAD VIs execute safe request methods. GET returns both the header and body data from a resource. HEAD only requests the header information of a resource. HEAD is often used to determine if a resource exists. Because only the header is returned by the server the request/response time is shorter than GET. Request variables can be used with these two methods to submit data to the server.

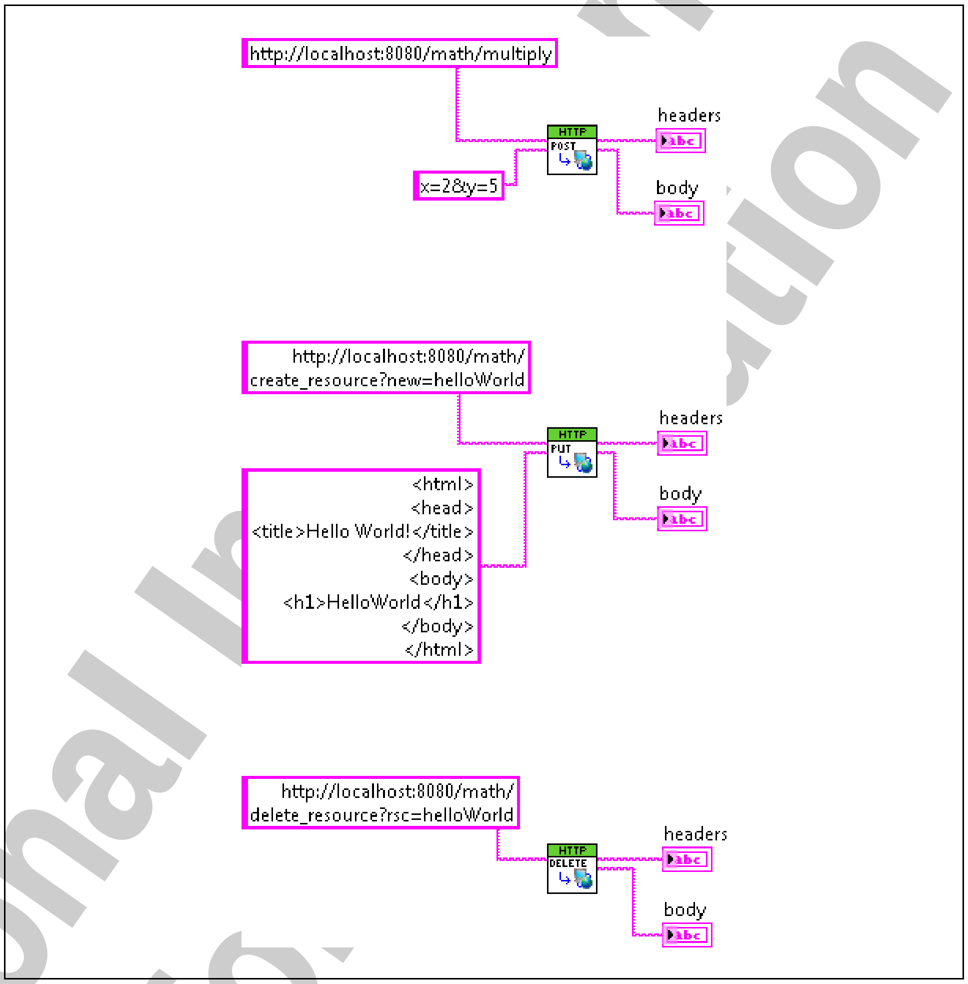

http://localhost:8080/math/multiply
x=2&y=5
HTTP
POST
headers
body
http://localhost:8080/math/create_resource?new=helloWorld
<html>
<head>
<title>Hello World!</title>
</head>
<body>
<h1>HelloWorld</h1>
</body>
</html>
HTTP
PUT
headers
body
http://localhost:8080/math/delete_resource?rsc=helloWorld
HTTP
DELETE
headers
body

Figure 6-14. Not-safe HTTP Request Method VIs

POST, PUT, and DELETE are all request methods that submit data to a server, result in a change of state on the server, or add or remove a resource on a server. POST is the most common of these methods and is often used to submit form data to a server. Form data uses name/value pairs with syntax similar to request variables. POST data can be encrypted for secure communication between a client and a server. PUT is intended to be used to create a resource on a server by submitting a file or data to the server. In Figure 6-14, the PUT VI is submitting the HTML contents of a resource that will be created on the server. DELETE is intended to be used when requesting to remove a resource on the server. Request variables can also be used with any of these methods.

Adding Web Services to an Existing Application

As the Web becomes more prevalent as the mechanism of choice for presenting data over the network, more applications will have to be modified so they can be accessed using Web services. All of these methods *except VI server* require you to make code changes to the original application.

There are two primary ways you can extend an existing application to use Web servers. We can either design our application such that it remains running in LabVIEW or an EXE. It runs outside of the Web service run-time, or convert the application to an auxiliary VI that runs in the Web service run-time even though it is not Web extensible. The first scenario requires us to transfer data between the original application and the Web service through the TCP stack even if both the application and Web services are local to the same machine. This is required because the application and the Web service do not share the same memory space.

If the original application is configured to run as an auxiliary VI we can use queues, global variables, RT FIFOs, etc to transfer data. Because the auxiliary VI and Web methods all exist within the same memory space we do not need to communicate through the TCP stack. This allows higher throughput, lower latency communication to be possible.

Network Published Shared Variable Technique

Shared Variables can be easily added to the application and Web methods to transfer data between the two. Be aware that it is possible to access network published Shared Variables from locations outside your LabVIEW VIs. It can be difficult to secure the data moving between the original application and Web service.

Auxiliary VI Technique

If you convert your original application to an auxiliary VI, you can take advantage of data transfer mechanisms that to not have to go through the TCP stack. Some examples of these are queues, RT FIFOs, and single process Shared Variables.

Using Auxiliary VIs to Keep References in Memory

References used in Web methods are discarded when the Web method finishes executing. This presents a problem whenever we need to keep a reference in memory between calls of the Web methods. This is commonly necessary when we want to avoid taking the time to open a reference to a resource for each call of the Web method. This can be avoided by using an auxiliary VI, that can run continuously to maintain the reference in memory.

This reference can be then passed to a Web method VI using queues or another data transfer mechanism.

To practice the concepts in this section, complete Exercise 6-4.

Self-Review: Quiz

1. Which of the following are client requirements to interface with LabVIEW Web services?
   a. Dependency on the LabVIEW Runtime Engine (RTE)
   b. Use the RESTful architecture
   c. Use HTTP protocol
   d. Use a Web browser

2. Which of the following are valid terminal labels for a web method VI that sends and receives data using the connector pane terminals?
   a. Input_1
   b. Input:1
   c. Input&1
   d. Input-1

3. Which of the following are server requirements, when using the HTTP Client VIs?
   a. Dependency on the LabVIEW Runtime Engine (RTE)
   b. Use the RESTful architecture
   c. Use HTTP protocol
   d. Use a web browser

Self-Review: Quiz Answers

1. Which of the following are client requirements to interface with LabVIEW Web Services?
   a. Dependency on the LabVIEW Runtime Engine (RTE)
   b. Use the RESTful architecture
   c. Use HTTP protocol
   d. Use a Web browser

2. Which of the following are valid terminal labels for a web method VI that sends and receives data using the connector pane terminals?
   a. Input_1
   b. Input:1
   c. Input&1
   d. Input-1

3. Which of the following are server requirements, when using the HTTP Client VIs?
   a. Dependency on the LabVIEW Runtime Engine (RTE)
   b. Use the RESTful architecture
   c. Use HTTP protocol
   d. Use a web browser

Notes

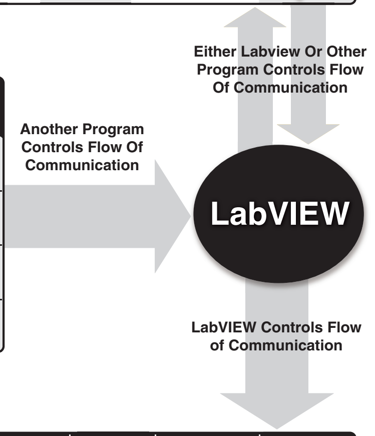
A

LabVIEW Connectivity Options

This appendix contains a table showing how communication flows between LabVIEW technologies and other technologies.

Technology	Examples	NI Technology	Local Machine Only	Covered in this Course	Covered in other Courses
Shared Variables	*LabVIEW on Real-Time Target*	✓			✓
TCP Protocol	*Other LabVIEW or Non-LabVIEW Applications*			✓	
UDP Protocol	*Other LabVIEW or Non-LabVIEW Applications*			✓	
Network Streams	*LabVIEW on Desktop or Real-Time Target*	✓			✓
IrDA (Infrared) protocol	*Nearby IrDA Device*				
Bluetooth	*Bluetooth-Enabled Device*				
VI Server	*Modify Front Panel Objects, Run Local or Remote VIs*	✓		✓	
VI Scripting	*Edit Traverse VIs*	✓			✓

Technology	Examples	NI Technology	Local Machine Only	Covered in this Course	Covered in other Courses
ActiveX Automation	*Microsoft Excel, Micosoft Word*		✓	✓	
LV-built shared libraries	*LabWindows/CVI, Microsoft VisualC++*		✓		✓
.NET Assembly Generator	*Microsoft C#*		✓		✓
LabVIEW Web Services	*Web Browser*			✓	


Either Labview Or Other Program Controls Flow Of Communication
Another Program Controls Flow Of Communication
LabVIEW
LabVIEW Controls Flow of Communication

Technology	Examples	NI Technology	Local Machine Only	Covered in this Course	Covered in other Courses
Windows Registry Access VIs	*Registry Keys, Registry Values*		✓		
Source Code Control tools	*Subversion, Perforce, ClearCase*		✓		✓
NI LabVIEW Database Connectivity Toolkit	*Oracle, Microsoft Access, SQL Server*			✓	
Event Structure	*Keyboard, Mouse Movements*		✓		✓
Input Device Control VIs	*Joystick Movements, Keyboard Movements, Mouse Movements*		✓		
Call Library Function Node	*Call Shared Libraries (e.g., DLLS)*		✓	✓	
System Exec VI	*Invoke Executables, Call System-Level Commands*		✓		
.NET Functions	*Microsoft System Components*		✓	✓	
ActiveX Container, ActiveX Functions	*Microsoft Excel, Microsoft Word*		✓	✓	
Instrument Drivers, VISA, or other network/bus communication	*DMMs, Oscilloscopes, Power Supplies*				✓
DataSocket VIs and Functions	*OPC Devices*	✓			
SMTP Protocol	*E-mail Communication*				
SOAP Web Services	*Web Services (e.g., Weather Information, Stock Information)*				
File I/O	*XML File, Text File, TDMS File, TDMS File, Binary File, INI File*		✓		✓
HTTP Client	*Web Pages, Web Services*			✓	

B

Additional Information and Resources

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

National Instruments Technical Support Options

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

- **Support**—Technical support at `ni.com/support` includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit `ni.com/support` for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at `ni.com/forums`. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

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

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. areas. Visit `ni.com/training` for more information about the NI certification program.

LabVIEW Resources

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

LabVIEW Publications

Many books have been written about LabVIEW programming and applications. The National Instruments Web site contains a list of all the LabVIEW books and links to places to purchase these books. Publisher information is also included so you can directly contact the publisher for more information on the contents and ordering information for LabVIEW and related computer-based measurement and automation books.

info-labview Listserve

`info-labview` is an email group of users from around the world who discuss LabVIEW issues. The list members can answer questions about building LabVIEW systems for particular applications, where to get instrument drivers or help with a device, and problems that appear.

To subscribe to `info-labview`, send email to:

`info-labview-on@labview.nhmfl.gov`

To subscribe to the digest version of `info-labview`, send email to:

`info-labview-digest@labview.nhmfl.gov`

To unsubscribe to `info-labview`, send email to:

`info-labview-off@labview.nhmfl.gov`

To post a message to subscribers, send email to:

`info-labview@labview.nhmfl.gov`

To send other administrative messages to the `info-labview` list manager, send email to:

`info-labview-owner@nhmfl.gov`

You also may want to search previous email messages at:

`www.searchVIEW.net`

The `info-labview` web page is available at `www.info-labview.org`