Calling LabVIEW VIs from Other Programming Languages

Publish Date: Jun 06, 2018

Overview

You can call a LabVIEW VI from other programming languages such as LabWindows™/CVI™, C++, Visual Basic, C#/.NET, and JAVA using ActiveX or by building VIs into shared libraries.

Table of Contents

1. Calling a VI Using ActiveX
2. Calling a LabVIEW-Built Shared Library
3. Related Links
4. Sample Programs and Examples

1. Calling a VI Using ActiveX

ActiveX refers to Microsoft's ActiveX technology and OLE technology. ActiveX is a Microsoft standard for inter-application communication. The standard passes data and commands among applications on one computer in the same way it performs the functions over a network, making network communications transparent. Other Windows applications can access LabVIEW (http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/windows_connectivity/) using ActiveX technology. With ActiveX automation, a Windows application provides a public set of objects, commands, and functions that other Windows applications can access. ActiveX is available only on Windows. Refer to the Microsoft Developer's Network (MSDN) documentation for more information about ActiveX.

- *Inside OLE*, by Kraig Brockschmidt, second edition
- *Essential COM*, by Don Box

ActiveX Automation

LabVIEW interacts with other Windows applications using ActiveX technology. ActiveX automation regulates the communication of data and commands between LabVIEW and other applications. Using automation, you can send commands and data to different applications with a single format. ActiveX automation allows you to integrate several applications to create a unified data acquisition, analysis, and presentation system. An application invokes ActiveX methods on another application to control or perform an action in that application. For example, using ActiveX automation (http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/using_activex_with_labview/), a LabVIEW program can launch Microsoft Excel and open an existing spreadsheet, invoking methods that Excel, as an ActiveX server, exposes for automation. Usually, objects in an application have methods that other applications can access. For example, after LabVIEW opens the spreadsheet, it can create rows and columns of data in the 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.

To invoke a method, you must know the name of the method. In some development environments, you must enter the correct method name, which can be found with an object browser. In Visual Basic, as long as you know the beginning of the method name, the environment attempts to complete your line of code for you. In LabVIEW, you wire the name of your server application to an invoke node and a shortcut menu containing the appropriate methods appears automatically. You then select the method or methods you need from the menu.

In addition to invoking methods, applications can get and set properties in other applications or in other application objects. Properties include attributes of the application, such as setup options, colors, and titles. For example, Visual Basic (http://www.ni.com/white-paper/document/epd-1182) can control the setup options of a LabVIEW VI. You can get and set properties of other applications in the same way you invoke methods. In other programming languages, you type the property and the value you need to get or set. Methods and properties keep ActiveX automation simple by limiting the number of commands to learn. They also keep ActiveX automation powerful by ensuring the flexibility of those commands through the variety of options methods and properties offer.

Servers and Clients

ActiveX automation defines the relationship between communicating applications. When two applications communicate, one application acts as a server and the other acts as a client. 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, allowing other applications to access LabVIEW objects, properties, and methods. If you want to call a stand-alone application from an ActiveX automation client, you must enable the ActiveX server for the application when you build the application. To do so, place a checkmark in the **Enable ActiveX server** checkbox on the **Advanced page** of the **Application Properties** dialog box. The client application initiates the connection to a waiting server application. The client requests a connection and sends commands to the server to transfer data, change parameters, run tests, and so on. The ActiveX automation server exposes methods and properties for the client to control and read. ActiveX defines all commands in terms of methods or properties. You can apply the commands to any combination of applications, methods, and properties using the ActiveX automation standard.

Transferring Data

ActiveX automation makes the transfer of data between applications easier. Usually the most basic way of transferring data from one application to another is by writing and reading ASCII text files. Most applications can read text data and convert it to the appropriate format for computation or manipulation. However, this delays the transfer of data by requiring additional time, effort, and disk space to reformat data to fit the needs of the application. ActiveX automation avoids delays by transferring data without the intermediate state. Using ActiveX automation, a client program is able to launch a server application and send data, for example the values for the rows and columns of a spreadsheet, to the appropriate object in the server. A client application can also read data from the correct object in the server.

ActiveX VIs, Functions, Controls, and Indicators

Refer to Using ActiveX with LabVIEW (http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/using_activex_with_labview/#ActiveX_VIs__Functions__Controls__and_Indicators) for information about LabVIEW VIs, functions, controls, and indicators you can use to access the objects, properties, methods, and events associated with other ActiveX-enabled applications.

Refer to the Downloads section for examples of calling LabVIEW from other programming languages using ActiveX.

2. Calling a LabVIEW-Built Shared Library

ActiveX objects can be built into shared libraries, which provide a way for programming languages other than LabVIEW to access code developed with LabVIEW. Refer to Using Build Specifications (http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/building_standaloneapps/) for information on shared libraries in LabVIEW.

Building a VI Into a Shared Library

Use the Application Builder to build a C-style shared library that you can call from other programming languages to pass data (http://www.ni.com/white-paper/document/epd-1288) from one application to another. Refer to Developing and Distributing an Application (http://zone.ni.com/reference/en-XX/help/371361R-01/lvhowto/develop_distribute_applications/) for information on using Application Builder to build a shared library.

Note Calling a LabVIEW-built shared library requires the LabVIEW Run-Time Engine (http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/using_the_lv_run_time_eng/).

If you want to dynamically link a LabVIEW-built shared library into an application, known as run-time linking, you must write code to load the library and retrieve the exported functions. You can later free the library programmatically on Windows in C++ with the functions LoadLibrary, GetProcAddress, and FreeLibrary. When you statically link to a shared library, known as load-time linking, you do not need to write code to load the library and the functions. Instead, you can link to `labviewv.lib` or `labview.lib` from `labview \cintools`. When you launch the application, the library loads automatically. Similarly, you may need to call functions LabVIEW exports, such as memory management functions like DSNewHandle and DSDisposePtr. To do this, you should link your application against `\ cintools \labviewv.lib`, or `\ cintools \labview.lib`, or you need to write code to load the functions. To learn about individual functions, refer to the LabVIEW Help.

When your application statically links to a LabVIEW-built shared library, the shared library loads when your application launches. The LabVIEW-built shared library loads the appropriate LabVIEW Run-Time Engine at that time. When your application completes, the shared library unloads which automatically unloads the Run-Time engine. When your application dynamically links to a shared library, you can decide when to load the shared library and retrieve any exported functions, for example LoadLibrary and GetProcAddress. When you retrieve the functions, the appropriate Run-Time Engine will load automatically. When your work is complete, you can unload the shared library using FreeLibrary which will unload the Run-Time Engine.

You also can use multiple shared libraries from different versions of LabVIEW. In this case, multiple versions of the LabVIEW Run-Time Engine can load at the same time. Each version of the Run-Time Engine has its own reference count to know when to load. If multiple shared libraries from the same version of LabVIEW are loaded and used at the same time, the Run-Time Engine is shared between them and a reference count is kept. If you unload one shared library, the Run-Time Engine reference count will decrease but it will not unload. When the last shared library unloads, the Run-Time Engine will be free at that time.

Use the Call Library Function Node (http://zone.ni.com/reference/en-XX/help/371361R-01/glang/call_library_function/) to call shared libraries from inside LabVIEW. The shared library also can link back to LabVIEW and call exported functions such as DSNewHandle. When you want to build a shared library that is called from LabVIEW and also calls back to LabVIEW, statically link your shared library with `labviewv.lib`. You also can load C-built shared libraries from different versions of LabVIEW at the same time. The library, `labviewv.lib` addresses multiple LabVIEW Run-Time engines in memory at the same time. The library allows a shared library to track the appropriate Run-Time Engine to call for the given thread even if multiple versions of LabVIEW are in memory at the same time. All C-built shared libraries should link with `labviewv.lib` and the different versions of LabVIEW resolve automatically.

Data Types and Memory Allocation

When using a shared library to pass data from another programming language to LabVIEW, you must interact with LabVIEW data types. Refer to the Call DLL VI in the `labview\examples\dll\data passing\Call Native Code.llb` for an example of LabVIEW data types. For more complex data types, you can use the LabVIEW memory manager (http://zone.ni.com/reference/en-XX/help/371361R-01/lvexcodeconcepts/memory_manager/). LabVIEW exports several functions that you can use to interact with LabVIEW memory (http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/how_labview_stores_data_in_memory/). In general, National Instruments recommends using simple data structures and keeping data flat when passing it from an external programming language into a LabVIEW-built shared library. In other words, use numbers, strings, 1D arrays, and other data structures that do not require the use of handles to refer to specific blocks of memory. Passing complex data types between applications often is difficult. The memory manager functions take care of allocating memory, creating LabVIEW data types and handling the data space sharing to pass complex data types into LabVIEW. Byte alignment is also related to the way data is stored in memory. LabVIEW uses one-byte alignment for its structures. When interacting with LabVIEW, make sure to use data that is one-byte aligned.

Note The use of memory manager functions can lead to problems if more than one instance of the LabVIEW Run-Time Engine is installed on your computer. Incompatible calls might result because the DLL uses the first instance of the Run-Time Engine that it finds, which might not be the correct version needed to run the DLL.

Calling a LabVIEW-Built Shared Library In C to Acquire, Analyze, and Present Data

This example illustrates the concept of sharing LabVIEW code by building a shared library. This example includes C and CVI applications that acquire, analyze, and present data by calling LabVIEW-built shared libraries.

Refer to the Downloads section below for the example file `call_lv_dll_in_c.zip`.

Requirements:
To view and run the LabVIEW source code behind the DLLs, you need LabVIEW 7.0 or later.

Project Files are included for Microsoft Visual C++ and LabWindows™/CVI™ so you can easily open and run the C code. Download LabWindows™/CVI™ 7.0 or later to run the C code and to try the premium C environment for measurement and automation applications. The LabWindows™/CVI™ Run-Time Engine is required to run the CVI executable.

To run the C code, open `call_LV_DLL.c` in any C compiler and include the `LV_AAP.h` and `LV_AAP.lib` files with your project. Place the `LV_AAP.dll` file in the same directory as the other files in your project. To run the CVI executable, make sure to have the `LV_AAP.dll`, `LV_AAP.h`, `LV_AAP.lib` and the `CallLVDLLLAAP.uir` files in the same directory as the `CallLVAAP.exe` executable. The example also includes the source code behind the LabVIEW-built DLLs and the buildscript used to create the DLLs.

Note When running the application from Microsoft Visual C++ 6.0, you must include the directory containing the LabVIEW support files in the "directories" section of the environment. From the VC++ environment, select **Tools»Options»Directories** and add the `C:\Program Files\National Instruments\LabVIEW x.x\cintools` directory where `x.x` is LabVIEW 7.0 or later. Adding the `cintools` directory prevents you from receiving the Cannot find `extcode.h` error.

Refer to the Downloads section for examples of calling a shared library from C#/.NET, C++, and Visual Basic.

3. Related Links

ActiveX and LabVIEW (http://www.ni.com/white-paper/2983/en/)
Calling LabVIEW DLL in Visual C++ that Passes Array Handles by Reference (http://www.ni.com/example/26518/en/)
Can LabVIEW C? (http://www.ni.com/white-paper/2718/en/)
Can LabVIEW C? — Example 3: Using the Right Tools with LabVIEW (http://www.ni.com/example/2719/en/)

4. Sample Programs and Examples

C/C++

- Calling LabVIEW DLL in Visual C++ That Passes Array Handles by Reference (http://www.ni.com/example/26518/en/)
- Refer to the Calling LabVIEW from C++ Using ActiveX (cpp) download (http://ftp.ni.com/pub/devzone/tut/callinglvfromcppusingax.zip)
- Refer to the Calling a LabVIEW built DLL from C++ (cpp) download (http://ftp.ni.com/pub/devzone/tut/calllvbuiltdllfromcpp.zip)
- Refer to the Calling a LabVIEW built DLL with Visual Basic and C++ download (http://ftp.ni.com/pub/devzone/tut/callinglvdllinvisuallcpp.zip)

.NET

- Calling LabVIEW DLL from C# App (http://forums.ni.com/ni/board/message?board.id=170&message.id=70372&requireLogin=False)
- Refer to the Calling a LabVIEW built DLL from C# (cs) download (http://ftp.ni.com/pub/devzone/tut/callinglvbuiltdllfromcsharp.zip)
- Refer to the Calling LabVIEW from C# Using ActiveX (cs) download (http://ftp.ni.com/pub/devzone/tut/callinglvcsharpusingactivex.zip)
- Refer to the Calling a LabVIEW built DLL from a C# Application download (http://ftp.ni.com/pub/devzone/tut/callinglvdllfromcsharpapp.zip)

Visual Basic

- Calling a LabVIEW VI from Visual Basic through ActiveX (http://www.ni.com/example/26182/en/)
- Refer to the Calling LabVIEW from Visual Basic with ActiveX download (http://ftp.ni.com/pub/devzone/tut/callinglvfromvbwithax.zip)