Using External Code in LabVIEW

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

- Using platform-specific protocols.
- Using the <u>Call Library Function Node</u> to call the following types of shared libraries:
 - Dynamic Link Libraries (DLL) on Windows
 - Frameworks on Mac OS
 - Shared Libraries on Linux
- Creating a <u>Code Interface Node</u> (CIN) to call code written specifically to link to VIs. LabVIEW must be installed on the computer you use to create a CIN.



Note To convert an instrument driver written in LabWindows[™]/CVI[™], download the LabVIEW Interface Generator for LabWindows/CVI Instrument Drivers from <u>ni.com/idnet</u> to complete the conversion of the front panel file.

Refer to cvilvsb.h in the labview\cintools directory for information about creating a LabVIEW CIN in LabWindows/CVI.



Note There are <u>platform-specific considerations</u> when calling external code in LabVIEW.

(Windows) To view related topics, click the Locate button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the Contents tab so you can navigate the related topics.

Configuring the Call Library Function Node

Use the <u>Call Library Function Node</u> to directly call a 32-bit 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 the Call Library Function Node and select **Configure** from the shortcut menu to display the <u>Call Library Function</u> 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** 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 <u>must have a version of the LabVIEW Run-Time Engine</u> that is compatible with each version of LabVIEW used to create the applications or shared libraries.

Configuring for Multiple Thread Operation

In a <u>multithreaded</u> operating system, you can make multiple calls to a DLL or shared library simultaneously. 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 the 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 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 protect access to global resources.
- It is called by only one non-reentrant VI.

Note All calls to LabVIEW-built shared libraries should specify **Run in any thread**. If you configure the Call Library Function Node using LabVIEW-built shared libraries and specify **Run in UI thread**, LabVIEW might hang and require you to restart.

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

This section discusses the return value and how to add parameters to the Call Library Function Node.

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.



Configuring Return Type

For return type, you can set **Type** to **Void**, <u>Numeric</u>, or <u>String</u>. **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 <u>C string pointer</u>. 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.

■ Open example ■ Browse related examples

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 specific operating system 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 platforms, translates the numeric data types to 32-bit integer types.



Note You can pass <u>extended-precision numbers and complex</u> <u>numbers</u> by selecting **Adapt to Type** from the **Type** pull-down menu. However, standard libraries generally do not use extendedprecision 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 that points to a fourbyte 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 array data pointers.

Note If you pass in an array that is smaller than the **Minimum** N 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.

Caution Do not attempt to resize an array with system functions, such as realloc. Doing so might crash your system. Instead, use one of the Code Interface Node (CIN) manager functions, such as NumericArrayResize .

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.
- **Caution** Do *not* attempt to resize an array with system functions, such as realloc. Doing so might crash your system. Instead, use one of the Code Interface Node (CIN) <u>manager functions</u>, such as <u>NumericArrayResize</u>.

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 greater than one-dimensional.



Note Pointer-sized signed and unsigned integers are not available in the **Data Type** pull-down menu for waveforms.

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 greater than one-dimensional.

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 greater than one-dimensional.



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 **Data Type** pull-down menu:

- ActiveX Variant Pointer—passes a pointer to <u>ActiveX</u> 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 data types to DLLs in the same way they are passed to a CIN. 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 to the 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. <u>Create a skeleton .c</u> 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.

Configuring Callbacks

Use callbacks to specify functions to call at predefined times. The functions allow you to initialize, update, and/or cleanup data for the DLL or shared library based on the particular instance of the Call Library Function Node you are configuring. Each call passes in the **Instance Data Pointer** for the Call Library Function Node the function is configured for. Use the Reserve callback to specify the function to call at reserve time and initialize receiving information for the instance. Use the Unreserve callback to call a function at unreserve time to save or analyze information and carry out clean-up operations. Use the Abort callback to specify a function to call if you abort a VI while a call to a DLL is in progress.

Calling Functions That Expect Other 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 <u>Flatten To String</u> 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 **byte order** 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.

Configuring Error Checking

Use error checking to ensure no errors occur if you call a DLL or shared library with the Call Library Function Node.

The **Maximum** and **Default** controls on the **Error Checking** tab of the <u>Call Library Function</u> dialog box allow LabVIEW to recover from unhandled exceptions that occur in the configuration of the Call Library Function Node or during a call to a shared library or DLL. The **Disabled** control on the **Error Checking** tab disables error checking but improves the execution speed of the Call Library Function Node.

CINs

The LabVIEW <u>Code Interface Node</u> (CIN) is a block diagram node that links C/C++ source code to LabVIEW. Use the CIN as an alternative to using the <u>Call Library Function Node</u> for calling source code written in C.

- Note It is technically possible to write CINs in a language other than C or C++ if the CIN entry points, such as CINRun, CINLoad, and so on, are declared as extern "C". However, National Instruments recommends using a shared library rather than a CIN if you want to use a language other than C or C++. LabVIEW does not support the creation of external subroutines. If you want to share code among multiple CINs, use DLLs.
- (Windows) To view related topics, click the Locate button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the Contents tab so you can navigate the related topics.

Programming Issues for CINs

This book describes the data structures LabVIEW uses when passing data to a <u>CIN</u> and describes the <u>function libraries</u>, or managers, that you can use in external code modules. The function libraries include the <u>memory manager</u>, the <u>file manager</u>, and the <u>support manager</u>.

(Windows) To view related topics, click the Locate button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the Contents tab so you can navigate the related topics.

Calling Shared Libraries

You can call shared libraries from LabVIEW. If you have existing applications written in other languages, such as C, Visual Basic, and Visual C++, you can use the <u>Call Library Function Node</u> for incorporating code directly into a LabVIEW block diagram. The Call Library Function Node is also useful for applications that use mathematical analysis routines or custom-designed data acquisition hardware.

(Windows) A shared library is called a DLL.

(Mac OS) A shared library is called a Framework.

(Linux) A shared library is called a Shared Library function.

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 <u>configure the</u> <u>Call Library Function Node</u> in LabVIEW. The general methods described here for DLLs also apply to other types of shared libraries.

LabVIEW loads shared libraries in a unique <u>application instance</u>. 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.

(Windows) You also can use functions that belong to the MathScript libraries class to call shared libraries from the LabVIEW MathScript Window or the MathScript Node. Refer to the MathScript Shared Libraries.lvproj in the labview\examples\MathScript\MathScript Shared Libraries directory for examples of calling shared libraries from MathScript.

Browse related examples

(Windows) To view related topics, click the Locate button, shown at left, in the toolbar at the top of this window. The LabVIEW Helphighlights this topic in the Contents tab so you can navigate the related topics.