How to Call Win32 Dynamic Link Libraries (DLLs) from LabVIEW

Jim Baker

Introduction

LabVIEW is a graphical programming language rich in data acquisition, data analysis, and data presentation capabilities. You assemble software components using the LabVIEW innovative graphical programming environment to create virtual instruments to meet your application needs. LabVIEW includes VIs to acquire data from plug-in data acquisition boards, programmable instruments, and other applications. It also includes VIs that analyze data and present results through graphical user interfaces. In most cases, the VIs included in the LabVIEW Development System meet the needs of users.

However, LabVIEW programmers can access Dynamic Link Libraries (DLLs) through the Call Library Function. DLLs are extremely powerful tools, because you can use them to share code among many applications. By using LabVIEW to access DLLs, you gain access to functions available in third-party libraries, including existing DLLs you or your colleagues may have written.

This application note discusses how to use the Call Library Function in LabVIEW 4.0 to access Win 32 DLLs, giving you access to numerous functions available in the Win32 Applications Programming Interface (API) for increasing the functionality of LabVIEW applications. Win32 is a 32-bit API provided in Windows 95 and Windows NT. This API has numerous changes from the Win16 (16-bit API in Windows 3.1 and Windows for Workgroups). Most functions contained in the Win16 API have equivalent functions in the Win32 API with their parameters changed from 16 to 32-bits.

The Call Library Function in LabVIEW 4.0 can also be used for direct access to shared libraries on Unix operating systems and on Mac OS.

For information on writing your own Win32 DLLs, see Application Note 087, Writing Win32 Dynamic Link Libraries (DLLs) and Calling Them from LabVIEW.

For information on using 16-bit DLLs with LabVIEW, refer to Application Notes 057 and 072, How to Call Windows 3.X 16-Bit Dynamic Link Libraries (DLLs) from LabVIEW, and Writing Windows 3.X 16-Bit Dynamic Link Libraries (DLLs) and Calling Them from LabVIEW.
The Call Library Function

LabVIEW 4.0 features the Call Library Function node to offer easy access to your dynamic link libraries.

Some of the important features of the Call Library Function in LabVIEW for Windows 95/NT are:

- You can call DLLs that use either the C or the Default(stdcall) calling convention.
- You can pass integer and floating point arrays of arbitrary dimensions.
- You do not have to be concerned about HUGE, NEAR, or FAR pointer types.
- LabVIEW strings can be passed as C or Pascal string pointers, or as a LabVIEW string handle, depending on the DLL being called.
- You can use void, integer, and floating point return types.

The Call Library Function icon is located in the Advanced subpalette of the Functions palette.
To configure the Call Library Function to call a specific function within a DLL, pop up on the icon and select the Configure... option, as shown.

You use the following configuration window to specify the DLL, the specific DLL function to call, and the function parameters.
Example – Calling a Function in User32.DLL

What Information do You Need?

This example of using LabVIEW 4.0 to call a DLL will involve making a function call to one of the standard DLLs that is part of the Windows 95/NT operating system. You will learn how to configure the Call Library Function to call USER32.DLL, which resides in the \WINDOWS\SYSTEM directory. In USER32.DLL, you will call the MessageBoxA function to create a three-button dialog box with “Yes”, “No”, and “Cancel” buttons. Note that you can easily create this type of dialog box in LabVIEW without the use of a DLL, but the DLL is already written for you and using it reduces the development time. This example is used to introduce you to the concepts of the Call Library Function, and to demonstrate the use of the Win32 API.

When you call a function in a DLL, you need to know the following information, almost all of which can be obtained from the appropriate Win32 include file (windows.h, winuser.h, and so on):

- The data type returned by the function; you can use LabVIEW to call functions that return void, integer, or floating point data types (signed or unsigned 8, 16, and 32-bit integers, or 32 and 64-bit floating point data types).
- The calling convention used; both C and Default(stdcall) conventions are available. The Win32 API uses the Default(stdcall) convention whereas most user written DLLs use the C convention.
- The parameters to be sent to the function, their types, and the order in which they must be passed.
- The location of the DLL on your computer.

To find this information for the MessageBoxA function, consult a Windows programming manual that covers the Win32 API. If you have installed a 32-bit Windows compiler such as Borland C++ or Microsoft Visual C++ then you will also have access to the Windows “include” files such as windows.h, windowsx.h, and the winuser.h. You will find your compiler documentation and the .h include files to be invaluable resources in locating information about the Win32 DLL functions (other useful tools for viewing export functions in a DLL are QUICKVIEW, provided with Windows 95, and DUMPBIN, provided with Visual C++). The description of the MessageBoxA function from winuser.h supplies us with the information we need to call the function:

```
int MessageBoxA (hWnd, lpText, lpCaption, uType)
```

The return type for the function is defined as a 32-bit signed integer:

```
int 32 bit signed integer
```

The Win32 API lists the names of the constants for the possible return values for the MessageBoxA function. The actual values of these constants are stored in the winuser.h file. In this example, the possible return values are IDYES, IDNO, and IDCANCEL, which have the decimal values 6, 7, and 2, respectively. If the message box cannot be created due to a lack of memory, zero will be returned.
parameters

The *Microsoft Win32 Programmer’s Reference* lists the data types of each of the parameters to the `MessageBoxA` function; the actual type definitions are all found in the `winuser.h` file.

**HWND hWnd**  Identifies the owner or parent window of the message box to be created. If this parameter is `NULL`, the message box has no owner window. The `HWND` data type is a 32-bit unsigned integer as defined in `winuser.h` and `windows.h`. Essentially, we can identify which window the message box “belongs to” by passing a valid value for `hwnd`. However, it is not necessary to define a parent for this window, so we will assign “no parent”, or `NULL`. The constant `NULL` is defined to be zero.

**LPCTSTR lpText**  The `LPCTSTR` type is a 32-bit pointer to a constant character string and is defined as a C-style (`NULL` terminated) string. This string contains the text we wish to display in the window.

**LPCTSTR lpCaption**  This parameter is a C-style constant character string containing the desired name to appear in the title bar of the window.

**UINT uType**  The `UINT` data type is defined as an unsigned 32-bit integer value. It determines which type of message box is displayed. The Windows API lists the names of valid constants that can be passed to this function, and `winuser.h` will contain the actual decimal values. In this example, we will create a dialog box with “Yes”, “No”, and “Cancel” buttons. The name of the constant is `MB_YESNCANCEL`, which is defined to have the value 3 in `winuser.h`. We will pass this value for the `uType` parameter. The other types of message boxes and their corresponding `uType` are:

<table>
<thead>
<tr>
<th>message box button type</th>
<th>uType</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>0</td>
</tr>
<tr>
<td>OK CANCEL</td>
<td>1</td>
</tr>
<tr>
<td>ABORT RETRY IGNORE</td>
<td>2</td>
</tr>
<tr>
<td>YES NO CANCEL</td>
<td>3</td>
</tr>
<tr>
<td>YES NO</td>
<td>4</td>
</tr>
<tr>
<td>RETRY CANCEL</td>
<td>5</td>
</tr>
</tbody>
</table>

*Warning:*  Do not use different values for `uType` than those listed in the Win32 API or `winuser.h`. You could cause errors in Windows 95/NT which may result in a crash or incorrect behavior!

calling convention

The calling convention for the `MessageBoxA` function can be found in the `winuser.h` file. Searching in `winuser.h` for `MessageBoxA`, we find that the function is preceded by the word `WINAPI`. This is defined as `__stdcall` in `windef.h`.

The Default(“standard C” or “__stdcall”) calling convention is used to call Win32 API functions. Parameters are passed by a function onto the stack from right to left, and are passed by value unless a pointer or reference type is passed. Function arguments are fixed, and a function prototype is required. Functions using this calling convention return values the same way as functions using the C calling convention. The C calling convention is the default calling convention for C and C++ programs. Arguments are passed from right to left; however, a called function pops its own arguments from the stack. Because the stack is cleaned up by the caller, it can have variable argument functions.
Creating the VI that Calls the MessageBoxA Function in the USER32.DLL

First, create the following front panel. Note that the comments on the front panel, which are there to explain the controls and indicators, are not necessary.

On the block diagram, create the Call Library Function. To view the on-line help for this function, select Show Help from the Help menu and move the cursor over the function icon. You can also obtain more information about this function either by selecting Online Help from the pop-up menu, or by reading Chapter 24 of the LabVIEW User Manual for Windows.

Note that at this point, only one set of terminals appears on the function icon, and they are grayed out. After you configure the Call Library Function for the DLL function you wish to use, the appropriate terminals will be available on the icon. Pop up on the Call Library Function icon, and select Configure... from the pop-up menu.

Complete instructions are listed below:

1. Type USER32.DLL in the Library Name or Path box. You will not need to type in the entire path to the DLL unless the DLL is stored in a location that does not appear in the PATH statement in your AUTOEXEC.BAT file or the LabVIEW VI Search Path.
   If you press <enter> on the keyboard, the configuration window will close. You can re-open it by selecting Configure... from the pop-up menu of the Call Library Function icon or by double-clicking on it.

2. Next, click in the Function Name field, and type the name of the function: MessageBoxA. Function names in general are case sensitive.

3. You do not need to change the value in the Calling Conventions box.

4. At this point, you need to indicate what kind of data the MessageBoxA function will return to LabVIEW when it has finished. We know the return value of the function is a 32-bit integer indicating which button was pressed in the dialog box. To set this, we use the Parameter and Type fields. Observe that the Parameter field contains the text “return type”, and below this, you see that the Type is set to Void.
The MessageBoxA function returns a signed 32-bit integer value, so select this data type for the Return value. To do this, click on the selection box next to the Type field and select Numeric from the pop-up list. You may also change the name of the return type from “return type” to something more descriptive, for example “button pressed”.

5. After setting the return type to Numeric, you will see a new field appear, called Data Type. The default is Signed 32-bit Integer.

6. In addition to defining the return type of the function, we must also define the four arguments to be passed to the function. The first argument of the MessageBoxA function is the hWnd parameter, which we know to be an unsigned 32-bit integer. Click on the Add Parameter After button to add the first parameter. Then, select Unsigned 32-bit Integer from the Data Type menu. Because the function expects the value, and not a pointer to the value, leave the Pass setting unchanged. If you like, you can change the name of the parameter from arg1 to something more descriptive, such as hWnd.

7. From the definition of the MessageBoxA function, the second and third arguments of the function are pointers to C-style strings. To add a string to the parameter list as the second argument, first make sure that the first argument appears in the Parameter box. You can select an argument by using the selector to the right of the box containing the parameter name. Click the Add a Parameter After button. To set the Type of the data to a pointer to a string, select String from the Type menu.

When you send a string to a function, you can select whether the pointer to the string points to a C-style (string followed by a NULL character), Pascal-style (string preceded by a length byte) string, or as a LabVIEW string handle (four bytes of length information followed by the string data). In this example, the default setting (C String Pointer) is correct. For more information about pointers and strings, see the Array and String Options section of this document.

8. The third argument passed to this function is another string, which contains the title of the message box window. Setting up this argument is the same as the previous argument.

9. Finally, you must add the uType parameter, which is an Unsigned 32-bit Integer. This is the value that determines which type of message box is displayed.

When you have finished configuring the Call Library Function, you can double-check if your configuration is correct by comparing the Function Prototype displayed in the configuration window to that obtained from the documentation of the function. This will help you to be certain that you are passing the correct data types to the function. Note that LabVIEW uses descriptive names for data types. For example, the int 32 data type describes a 32-bit signed integer in LabVIEW. In most compilers, this data type is described as int.
Check to see that you have completed the dialog box correctly by studying the figure above. Click the OK button to close the configuration window. Notice how terminals have been added to the icon, and the parameters of the function listed from left to right in the function prototype match the data types appearing on the terminals of the icon from top to bottom. The upper left input terminal is disabled because the top output terminal is the return value of the function, not an argument to the function.

To complete the VI, build the following diagram. Remember to make sure to set the representation of the numeric constants you connect to the Call Library Function icon to the correct type.

**Note:** All input terminals to the Call Library Function must receive data!

Once you have finished constructing the diagram, save your program and run it.
Additional Examples

A. If you have a sound card with Windows sound drivers installed on your system, you can also investigate Play Sound.vi found in the LabVIEW Examples directory:

    \LABVIEW\EXAMPLES\DLL\SOUND\PLAYSND.LLB\Play Sound.vi

You can use this VI to play Windows .WAV sound files on your computer from LabVIEW.

B. If you do not have a sound card you can “beep” the PC by calling the MessageBeep function in User32.DLL. The function prototype is:

    VOID MessageBeep(UINT uType);

This function will generate a short tone through your PC speaker.

C. A good example for using LabVIEW string handles can also be found in the LabVIEW Examples directory:

    \LABVIEW\EXAMPLES\DLL\HOSTNAME\hostname.vi

This example will return the host name of your computer.
You can programmatically position your cursor anywhere on your monitor using the SetCursorPos function in User32.DLL. The function prototype is:

```c
BOOL SetCursorPos( INT x, INT y);
```

x and y are the desired coordinates referenced from the upper left corner of the screen. The return value is TRUE if the function was successful and FALSE if it was not. (The value returned is type BOOL, which is defined in the Win32 API as a 32-bit signed integer with values 0=FALSE and 1=TRUE.)

## Array and String Options

This section briefly reviews some important concepts you should be familiar with when using the Call Library Function to work with array and string data.

### Arrays of Numeric Data

Arrays of numeric data can be any type of integers, or floating point numbers with single (4-byte) or double (8-byte) precision. When you pass an array data pointer, you can also set the number of dimensions in the array, but you do not include information about the size of the array dimension(s). You will have to pass this information to your DLL in a separate variable unless you are using LabVIEW array handles. Because the Win32 API does not use LabVIEW array handles, the function definition of the specific API function you are calling will specify which array parameters are required.

### String Data

The Call Library Function passes C or Pascal-style string pointers, or LabVIEW string handles. You must select the same type of string pointer as that used in your function, or errors will occur. The C-style string consists of the string followed by a NULL character. The Pascal-style string consists of the string preceded by a length byte. The LabVIEW string handle consists of 4-bytes of length information followed by the string data. Most Win32 API functions use the C-style string pointer.

## Important Reminders and Quick Reference

- Make sure that the path to the DLL file is correct.
- If you get the error message Function not found in library, check the spelling, syntax, and case sensitivity of the function name you wish to call.
- Make sure that all the parameters passed to a DLL function have data wired to all of the input terminals of the Call Library Function icon. Be sure to configure the function properly for all input parameters.
- Know the return types and data types of arguments for your functions and remember to configure the Call Library Function to exactly match the data types your function uses. Failure to do so may result in crashes.
- Make sure you use the proper calling convention (C or Default (stdcall)). The Win32 API uses the Default (stdcall) convention.
- Know the correct order of the arguments passed to the function.
- When passing strings to a function, remember to select the correct type of string to pass – C or Pascal string pointers, or LabVIEW string handle. The Win32 API uses the C-style string pointer.
- If you are working with arrays or strings of data, **ALWAYS** pass a buffer or array that is large enough to hold any results placed in the buffer by the function.
References

For material about writing DLLs and calling them from LabVIEW, consult:

*Writing Win32 Dynamic Link Libraries (DLLs) and Calling Them from LabVIEW*, Application Note 087, National Instruments

Charles Petzold, *Programming Windows 3.1.*