Best Practices for Sharing and Reusing LabVIEW Code

Publish Date: aug 19, 2009 | 13 Ratings | **3.54** out of 5 | Print | 4 Customer Reviews | Submit your review

Overview

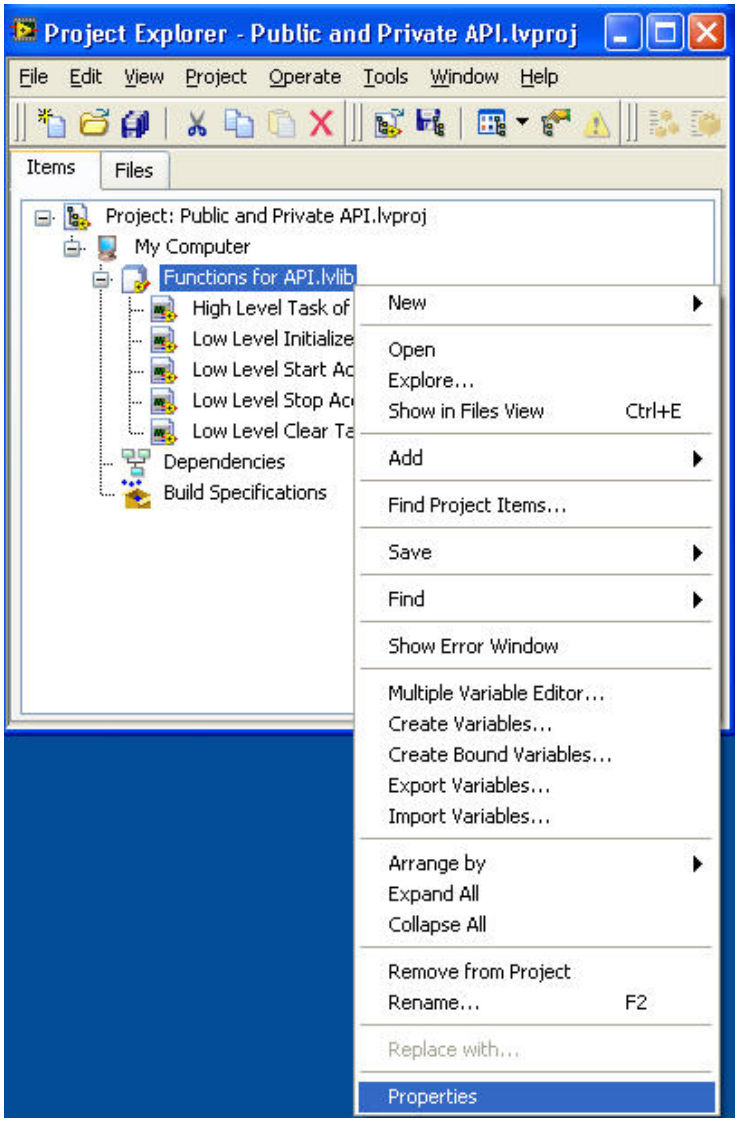
Sharing code is a common task for large applications. The use of proper techniques for sharing code can improve the efficiency of the development team by reducing the duplication of work and the risk of bugs in the code. In LabVIEW, two of the most common ways of sharing or re-using source code are with Project Libraries, or .lvlib files, and Source Distributions, or .llb files. Depending upon the type of application, each of these libraries has its own advantages over the other and makes the sharing of the source code easier. Here are some common applications and the types of library files that fit the best with them.

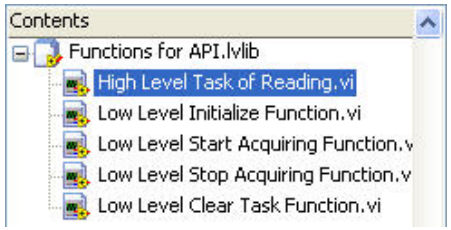
Table of Contents

Custom API

Creating a custom API is a good way of implementing a common interface for multiple people to achieve the same functionality. A good example of this is with the NI-DAQmx driver or with some of the toolkits that can work inside of LabVIEW. These APIs expose some VIs that can be selected by the user to perform the task at hand. These VIs will then call into some other lower-level VIs to help achieve the functionality. Since it may not be necessary that the user know and understand all of the different low-level VIs that are called, it makes the API more intuitive if a limited set of VIs are available to be placed on the block diagram. The developers can then pick the function block for the task they want to complete and save some time by not having to re-create the low-level functionality.

This idea of having some high-level functions exposed to the user for selecting makes the .lvlib library a perfect fit. When using an .lvlib with a number of VIs inside of it, the developer can select which of the VIs will be Public and which ones will be Private. This creates an ideal situation for an API that is being developed because all of the low-level VIs can be included with the Project Library when it is distributed, but they do not have to be exposed to the end user, which could cause the API to be less intuitive. For example, if a custom API is being built to perform a high level task of reading from a particular instrument, some of the lower level functions may be to initialize the task, begin acquiring data, stop acquiring data, and clear the task. By right-clicking on the name of the .lvlib, selecting the Properties option, and then going to the Item Settings tab, all of the VIs in the .lvlib will be shown like in the figures below.





While in this dialog box, the Access Scope of each of the VIs can be set to Public or Private. The high level functions that the user would put on their block diagrams would typically be set to Public while the lower level functions will usually be set to Private. This will make the API more user-friendly and intuitive since the high-level tasks will be what the user focuses on.

When comparing this use of the .lvlib to the possibility of using an .llb, the inability of the .llb to define certain files as Public or Private makes the .lvlib the better candidate for distributing and sharing source code for an API.

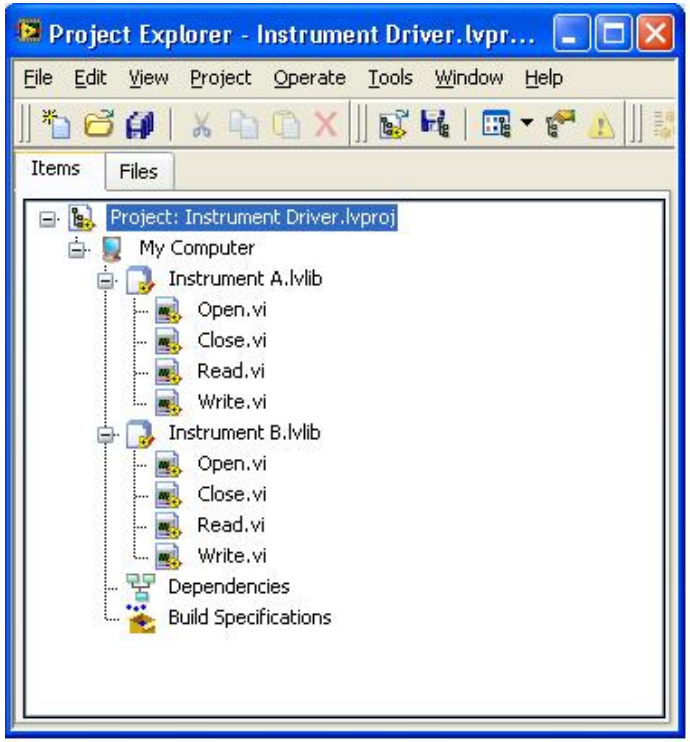
Type of project recommendation for a custom API: .lvlib

- Ability to change the access scope of all VIs in the library
- VIs with 'Public' scope create a more intuitive and user-friendly library

Instrument Drivers

LabVIEW is a programming language that is often used to communicate with other instruments. When attempting to do this, an instrument driver is often created so that people will be able to talk with the other hardware. When creating the driver, specific VIs can be created to perform functions that the instrument will perform. Some common examples of these functions can be Open a Reference, Close a Reference, Read, Write, etc. These functions are generic and can be applied to a number of different instruments. However, since different instruments will be anticipating slightly different commands, each function will have to be tailored to the instrument it is designed for. This can result in multiple Open, Close, Read, or Write VIs, which could result in a naming conflict.

This creates an ideal situation for using the .lvlib file format. By using the name-spacing functionality of the .lvlib, there can be multiple VIs with the same name in the LabVIEW Project. As long as the VIs are associated with a different .lvlib library, then each Project Library will be able to have its own set of functions that include the same names like Open, Close, Read, or Write. Using a different .lvlib for each instrument driver that is being developed could be an effective way of developing source code to be distributed for multiple instruments. Having a project file organized like the one below will allow for this type of driver development.



The name-spacing can also be easily demonstrated by simply opening a VI that is part of an .lvlib. Looking at the title bar of the front panel of the VI, the name of the .lvlib should be appended to the front of the name of the VI. This visually represents how the VI appears in memory. For example, here is how the title bar appears for Open.vi inside of Instrument A.lvlib.

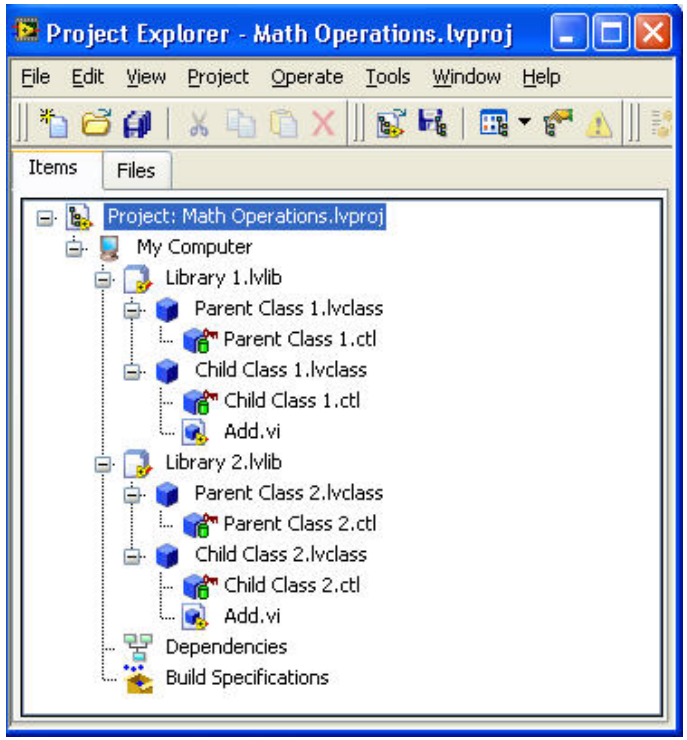
Type of project recommendation for instrument drivers: .lvlib

- Name-spacing functionality avoids naming conflicts
- Ability to have VIs with the same name for different instruments

Object Oriented Programming

When using Object Oriented Programming inside of LabVIEW, there is often the use of parent and child classes. The parent class will allow the child class to inherit data and VIs from it in order to use. With Object Oriented Programming, there are specific functions that must be performed on some data, but depends on where the data is coming from. The VIs in a child class will be able to perform operations on data knowing that the data is being inherited from a parent. There are often a number of child classes with the same VIs inside of them to perform the functions. Since the VIs inside of the different child classes have the same name, there is a potential for a naming conflict.

Similar to the instrument driver application, Object Oriented Programming makes the Project Library, or .lvlib, a great candidate for effectively sharing and using the source code. A defining characteristic of the Project Library is that it can be used to provide unique name-spacing. When any type of file is located inside of an .lvlib, that file is associated with that library and will have the Project Library name appended to the beginning of the file name when loaded into memory. As a result, having multiple VIs with the same name in different classes makes Object Oriented Programming a great candidate for using an .lvlib to avoid any naming conflicts in memory. The figure below demonstrates how two different child classes contain VIs that have the same name, but there are no naming conflicts because the VIs are associated with different .lvlib files.



When compared to the option of using an .llb for Object Oriented Programming, the .lvlib's name-spacing functionality makes it the superior choice. All of the VIs that are inside of an .llb do not have the Source Distribution's name associated with them. Therefore, if

multiple VIs with the same name in different .llbs are being called, then only one will be loaded into memory at one time because they will appear as though they have the exact same name. This will create a naming conflict with LabVIEW.

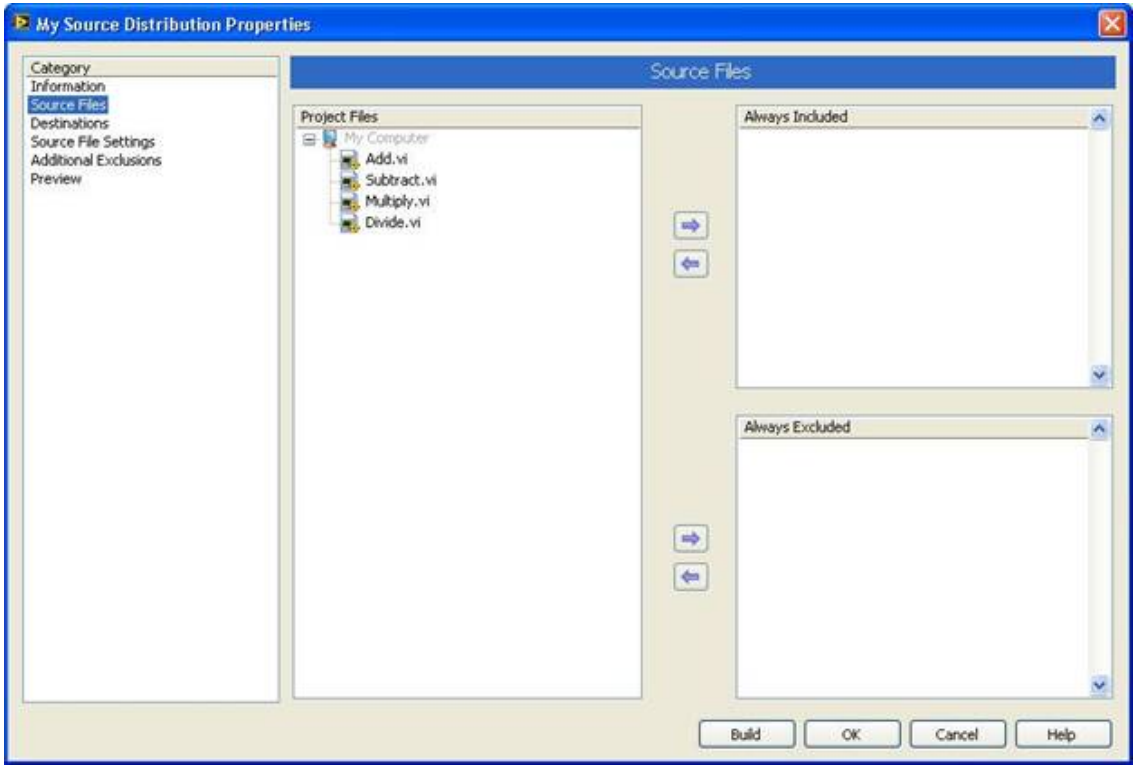
Type of project recommendation for Object Oriented Programming: .lvlib

- Name-spacing functionality avoids naming conflicts
- Ability to have multiple child classes with VIs that share the same title

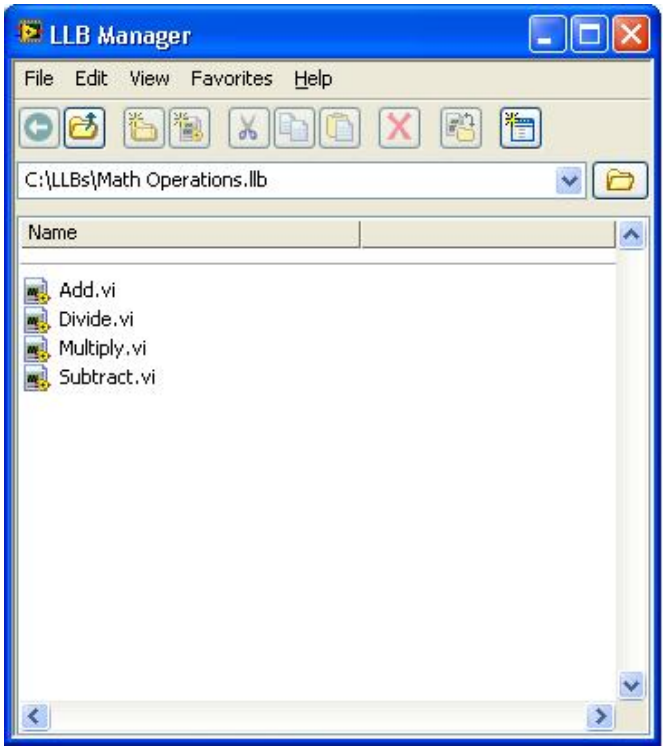
Commonly-Used Low-Level Functions

When developing a number of applications for different projects, there are times where some of the same functionality must be used several times. In these situations, it is helpful to be able to port these functions to other development machines easily and to be able to access them when desired.

Since these VIs will be moved from project to project and are not dependent upon the specific application, an .llb makes a good candidate for housing the VIs because of a number of reasons. In these types of applications, making it as easy as possible to move the VIs between developers is important. When using an .llb, all of the VIs can be put into the same file and moved as a single file, or Source Distribution. In addition, low-level functions can typically be called a number of times in a project, which can put more emphasis on the load time of a VI in order to avoid a slow application. Once a VI that is inside of an .llb is called, the entire .llb is loaded into memory. This makes successive calls to the same VI or other VIs inside of the .llb quick because they are already loaded into memory. When building an .llb, the files that should be included or excluded from it can be selected in the Source Files section of the dialog box like shown below.



Once the .llb library is created, all of the VIs that are part of it can be transported at the same time and can be viewed at once like shown below.



While considering what type of library to use, it is important to note why an .llb would be a better candidate in this situation than an .lvlib. When transferring an .lvlib with all of its contents to another developer, the files will be moved as the .lvlib file along with separate files for each of the VIs inside of it. This makes it more difficult to move the library and its contents than with an .llb because all of the files must be accounted for instead of one single file. In addition, loading VIs into memory is slower with an .lvlib than with an .llb. Since and .lvlib file does not encompass all of the VIs associated with it into the same file, the VIs are loose on disk as separate entities. This means that when one VI is called and loaded into memory, the other VIs are not in put into memory at the same time. For the VI calls that follow, each VI must be found on disk separately, which could slowly increase the time the application takes to run.

Type of project recommendation for commonly-used low-level functions: .llb

- Ability to transfer all functions in a single .llb file
- Ability to select which files to include or exclude
- Functions can be used for many different projects, since there is no name-spacing

Back to Top

Customer Reviews

Customer Reviews

4 Reviews | Submit your review

Needs updated, OOP section and PPLs - 20-mei-2015
In the OOP section, I believe that the class provides the namespace, so is there really any benefit to using a lvlib? Or do you just distribute the lvclass as a source distribution? Also, I don't think PPLs were out when this was written, it would be good if this article was revisited now that there is another option out there.

what about packed libraries - 31-jan-2013
This article should also talk about using packed libraries.

lvlib does not avoid naming conflicts with LVOOP - 5-jan-2012
"Name-spacing functionality avoids naming conflicts" is not an issue with LVOOP. The LabVIEW class name is prefixed to the member VI name. The LabVIEW class (lvclass) avoids name conflicts, not the lvlib. The example project provided above does not have a conflict when no lvlib is used. The following names would exist: Child Class 1.lvclass:Add.vi Child Class 2.lvclass:Add.vi

Thanks for a useful article - 2-jun-2010
This article is particularly concise, easy to read, and useful.

View more reviews