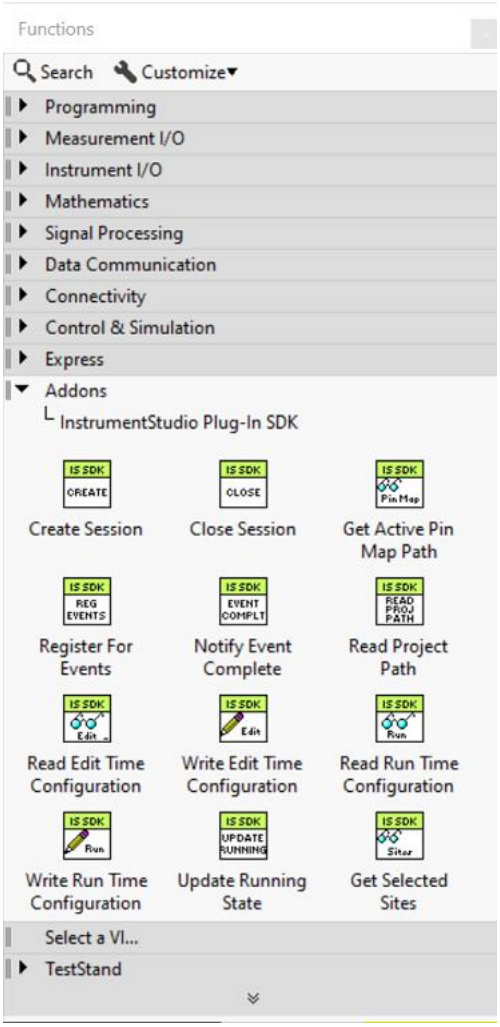
G Plug-In Development Guide for InstrumentStudio 2021

Creating Plug-In VIs

When creating a G plug-in for InstrumentStudio, you will first need to create a top level VI that you wish to use for the plug-in. For InstrumentStudio to establish communication with the plug-in, the VI must contain a U64 control named “Session Id” on the front panel. If the control is not present, InstrumentStudio will fail to load the plug-in. The control does not need to be part of the connector pane as there are no connector pane requirements for plug-in VIs. Since the Session Id control is not meaningful to end users of the plug-in panel, it is recommended to either place the control outside of the visible area of the front panel or use the Advanced -> Hide Control right-click menu command to make the control invisible.

After creating the front panel of the plug-in, you will need to use the APIs from the InstrumentStudio Plug-In SDK to establish a communication session with InstrumentStudio. The InstrumentStudio Plug-In SDK is available as a VI package (.vip) from the Releases in the instrumentstudio-plugins GitHub repo. Installing the SDK into LabVIEW will make the SDK VIs available from the Addons >> InstrumentStudio Plug-In SDK palette.

The InstrumentStudio Plug-In SDK palette API is shown below.



Once you have placed the first Plug-In SDK VI on the block diagram, you can obtain quick access to the palette through the right-click menu by clicking on a Plug-In Session terminal or wire. Properties that can be read or written through VIs in the API can also be accessed through a property node on the block diagram.

In order to be notified of events from InstrumentStudio, the plug-in must specifically register for each event using the Register For Events VI and create the appropriate cases within an event structure. Event notifications from InstrumentStudio are asynchronous and require an acknowledgement from the plug-in with the appropriate Completion Id once the plug-in has finished processing the event. Failure of a plug-in to acknowledge an event may result in InstrumentStudio becoming unresponsive with a busy cursor. To avoid this, be sure to always acknowledge all events using the Notify Event Complete VI and never discard or flush events from the queue of the event structure.

The plug-in will be notified to exit via the Shutdown event when the InstrumentStudio project or the soft front panel document containing the plug-in is closed. This event is different from other events in that it does not require an acknowledgement using the Notify Event Complete VI as the execution state of the VI going idle is the ultimate acknowledgement. If the plug-in fails to stop running in response to the Shutdown event, the thread running the plug-in VI will never exit. This will prevent projects from closing and the InstrumentStudio process from exiting on shutdown, requiring the user to manually kill the process using Task Manager. To avoid this, plug-ins should always register for the Shutdown event and stop execution within a reasonable amount of time upon receipt of the event. If a plug-in does not stop executing within a reasonable amount of time, users will start receiving prompts to abort the running plug-in instead of waiting for it to shut down gracefully.

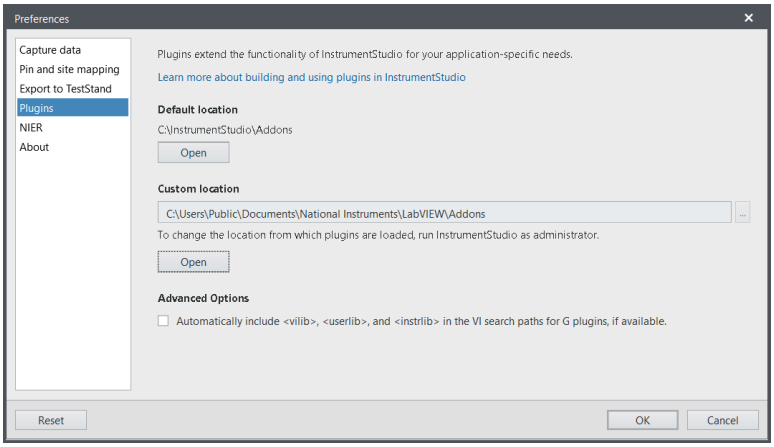
Adding Plug-Ins to InstrumentStudio

Once the plug-in VI has been created and you are ready to start testing it within InstrumentStudio, you will need to create a .gplugindata file. The .gplugindata file is what allows InstrumentStudio to discover plug-ins and contains information about how to display and execute plug-ins. Reference GPluginData-File-Format.md for more detailed information on the file format and what information is required.

The ApplicationContext setting within the .gplugindata file can have some subtle downstream consequences and warrants further discussion. The recommendation is that plug-ins should execute in unique application contexts whenever possible. A unique application context provides the most isolation for a plug-in and ensures other plug-ins cannot interfere with its execution due to VI name conflicts between plug-ins.

The only reason not to use a unique application context is if the plug-in needs to communicate with other plug-ins in InstrumentStudio, and it wants to do this using shared data like global variables or references to other data communication APIs like queues and events. In this scenario, both plug-ins must execute in the same application context, or they will not be able to share data. The alternative is to use IPC (inter-process communication) techniques like TCP/IP which already handle marshalling data across application context boundaries. However, if the additional complexity of implementing IPC is prohibitive, plug-ins can share a project context instead. Plug-Ins should rarely if ever use the global application context as it provides the least isolation. It should only be used when communicating with a plug-in from a third party without using IPC and use of a shared project is impractical.

Once the .gplugindata file has been created, it will need to be copied into one of the plug-in directories within InstrumentStudio along with the associated plug-in VIs and dependencies (see the Building and Deploying Plug-Ins section for additional details and recommendations on packaging of plug-ins). The default plug-in directory is the Addons folder underneath the root folder of InstrumentStudio. Plug-In authors are free to create any folder hierarchy desired underneath the plug-in directory as InstrumentStudio will recurse the entire directory hierarchy when looking for .gplugindata files. However, it is recommended that plug-ins install to a company specific subdirectory to avoid conflicts with other plug-ins that might be installed on the system. In addition to the default plug-in directory, users can also specify additional directories where plug-ins can be discovered. This is controlled by the preferences page for plug-ins as shown below.



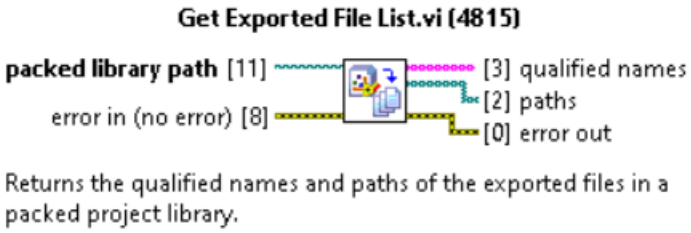
If there are errors detected in a .gplugindata file during discovery, those errors will be presented in a message box during launch of InstrumentStudio. Once you make edits to correct the errors and resave the .gplugindata file, you should get prompted to restart InstrumentStudio to pull in the new changes. If you choose not to restart or do not get prompted to restart, the changes will not take effect until the next time you launch InstrumentStudio. If there are no errors shown during launch, the plug-in should now be visible in the Edit Layout dialog of a soft front panel document.

Executing Plug-Ins within InstrumentStudio

Once the plug-in is visible in the InstrumentStudio Edit Layout dialog, you should now be able to instantiate and run the plug-in VI within InstrumentStudio. If the plug-in VI fails to load and run, the panel for the plug-in will display the LabVIEW error code and explanation for what caused the failure. Some error descriptions are more helpful than others, but the following are some common reasons a VI may fail to run. If you are unable to resolve the problem from the reported error code, you should verify the following.

1. An appropriate version of the LabVIEW Runtime Engine is installed. The minimum version required by InstrumentStudio is the 64-bit version of LabVIEW 2020 SP1. If you have LabVIEW 2020 installed instead of LabVIEW 2020 SP1, you may still be able to run the VI, but you will not be able to enable debugging of the plug-in. If the plug-in was built or saved in a newer version than LabVIEW 2020 SP1, then that version or newer of the runtime engine must be installed.
2. The VI or project file specified in .gplugindata file exists at the specified path. If you receive an error that the file can't be found, verify the relative path specified in the .gplugindata file

resolves to the expected absolute path on disk and that the VI exists on disk. If you are using a packed library to deploy your plug-in, a common mistake is to use the wrong relative path to the file within the packed library. To verify the path is correct, you can use the Get Exported File List.vi located in the Packed Library palette.

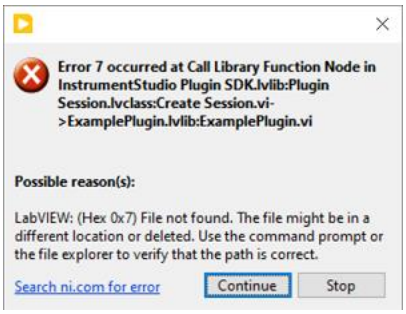


This VI will return the correct paths to use to access the files within the packed library.

3. The reentrancy setting of the plug-in VI doesn't match the reentrancy settings specified in the .gplugindata file. The reentrancy settings for the VI can be found in the Execution category of the VI Properties dialog. If the reentrancy setting on the VI is configured for reentrant execution, then reentrancy can be disabled programmatically based on the ReentrantExecutionEnabled setting in the .gplugindata file. However, if the reentrancy setting on the VI is configured for non-reentrant execution, attempting to execute the plug-in with ReentrantExecutionEnabled="true" will result in a runtime error.
4. You are attempting to run the plug-in from source without building a source distribution, and one or more sub VIs is a source only VI that does not contain compiled code to execute. Using a source distribution resolves this problem since the build specification will embed the compiled code with the source by default. Without the use of a source distribution, some other action on the machine will need to be taken to ensure the compiled object cache of LabVIEW has been populated with the code you are trying to execute. Generally, running the plug-in VI within the LabVIEW IDE of the same version is sufficient for populating the cache. However, take caution that the cache can be cleared at any time using commands in the Tools -> Advanced menu of LabVIEW. If this occurs, you will need to recreate the cache or use a source distribution.
5. The VI is not executable. If the reason is not obvious from the error code, the failure is often due to a missing dependency that cannot be found within the runtime engine. This can occur when trying to run VIs from source rather than a built source distribution or packed library or if the build specification omitted required dependencies as part of the build. Trying to run from source is problematic because the LabVIEW Runtime Engine is not able to resolve dependencies from vi.lib the same way the LabVIEW IDE can. To help diagnose these problems, LabVIEW creates a log file in the temp directory (e.g., C:\users\<username>\AppData\Local\Temp) using the naming convention *<process name>_BrokenVILog_<date and time>.txt* (e.g. InstrumentStudio_BrokenVILog_2021-04-19T175816.txt). While the information from this log file is not always straight forward, it is often helpful in identifying which dependencies might be preventing the plug-in from running. The log file accumulates error messages for each load failure throughout the process' lifetime so it can be helpful at times to close and relaunch InstrumentStudio to ensure the log file contains the minimum information pertinent to your plug-in's load failure.

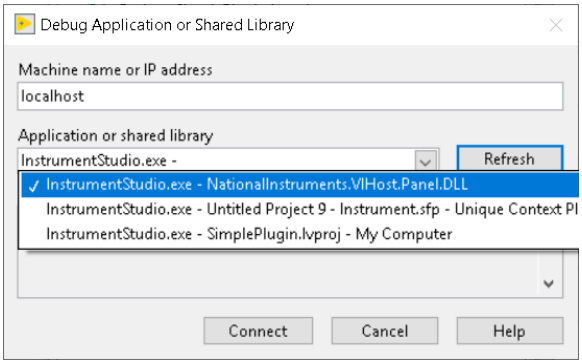
Debugging Plug-Ins within InstrumentStudio

While developing a plug-in, it is expected that most testing and debugging of the core plug-in functionality (for example, measurement and other non plug-in related code) will occur while executing the plug-in within the LabVIEW IDE. NOTE: You cannot run the top-level plug-in VI in the LabVIEW IDE independently of InstrumentStudio. The InstrumentStudio Plug-In SDK Vis that are being called cannot find the dlls they need in this context and you will get error dialogs such as this.



To test code specific to communication between the plug-in and InstrumentStudio, you will need to execute the plug-in within InstrumentStudio. If testing exposes problems in this area of the code, you can debug the problem using LabVIEW's remote debugging feature to attach to the code running in InstrumentStudio. Note, remotely debugging the plug-in is only possible from within the same version of the LabVIEW IDE as the version of the LabVIEW Runtime Engine in which the plug-in is executing.

To debug the plug-in running in InstrumentStudio, first enable debugging for the plug-in within the .gplugindata file by setting DebuggingEnabled="true". Once debugging is enabled, instantiate an instance of the plug-in panel within InstrumentStudio. The plug-in should now be visible within the Debug Application or Shared Library dialog shown below. This dialog is launched from LabVIEW's Operate menu.



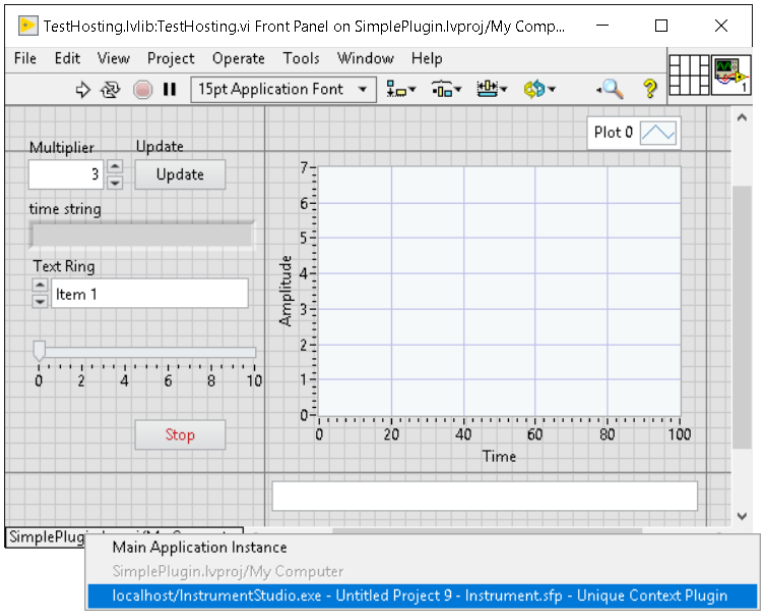
How the plug-in is displayed in the dialog and which context you need to connect to depends on the ApplicationContext the plug-in specifies in the .gplugindata file.

- **Global** – If the plug-in executes in the Global application context, you will need to connect to the *InstrumentStudio.exe – NationalInstruments.VIHost.Panel.dll* context that is highlighted in the first entry in the example above.
- **Unique** – If the plug-in executes in a unique application context, the context name will have the form *InstrumentStudio.exe - <InstrumentStudio project name> - <InstrumentStudio document*

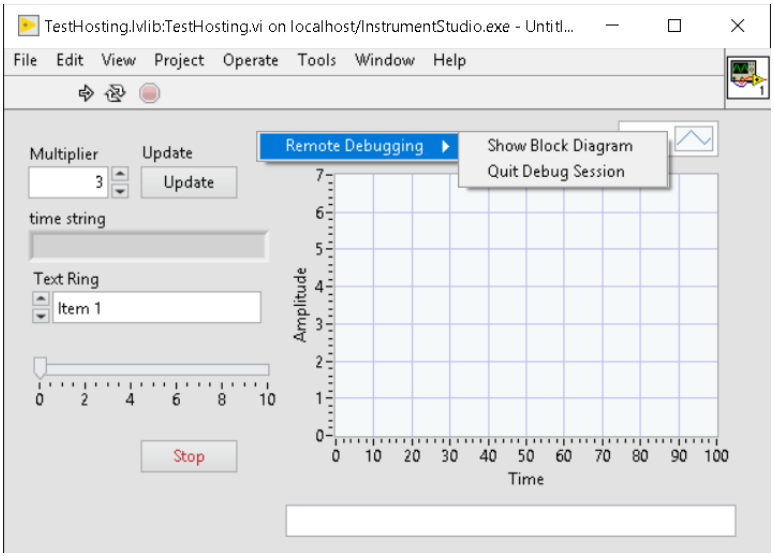
name> - <DisplayName property defined in .gplugindata file> as shown in the second entry in the example above.

- **Project** – If the plug-in executes in a Project context, the context name will have the form *InstrumentStudio.exe - <LabVIEW project name> - <target name in project>* as shown in the third entry in the example above.

After you have connected LabVIEW's remote debugger to the application context running the plug-in, you will need to open the plug-in VI in the same application context. The application context in which a VI is loaded is displayed in a control in the bottom left corner of the VI. To change the application context, right-click the control and select the appropriate context from the pop up as shown below.



Once loaded in the proper application context, the VI will update to reflect a running state. You can then right-click on the VI to quit the debug session or open the block diagram as shown below.



From the block diagram, you can set breakpoints and probes and debug as you typically would when running a VI locally within the LabVIEW IDE.

To support instantiating multiple instances of the plug-in within InstrumentStudio, it is recommended that your top-level plug-in VI be configured for reentrant execution. Unfortunately, bugs present in LabVIEW 2020 make it difficult to impossible to debug reentrant VIs as breakpoints often fail to break as expected and pause execution in the debugger. While these bugs are expected to be fixed in LabVIEW 2021, you may need to disable reentrant execution of the plug-in while debugging with LabVIEW 2020 as a work around. To disable reentrant execution, set ReentrantExecutionEnabled="false" in your .gplugindata file. However, be aware that once reentrant execution has been disabled, the plug-in panel can only be loaded once successfully within InstrumentStudio. Loading of subsequent panel instances will generate a runtime error until the initial panel instance is closed. Because of this, it is recommended that you create debug-only PluginData entries within a debug only .gplugindata file that is only used for internal testing and debugging. This will allow you to easily switch between debugging and testing of the plug-in in its desired release configuration. Having a separate .gplugindata file for debugging avoids the need to modify the original .gplugindata file you intend to release and minimizes the risk of forgetting to reset debug only settings back to their intended release configuration.

When remotely debugging the plug-in, it is generally better to interact with the front panel in LabVIEW rather than the panel in InstrumentStudio. This is because any changes made to control values from the LabVIEW front panel will be reflected in both panels while the reverse is not always true. However, to debug events sent from InstrumentStudio, you will first need to perform the appropriate action within InstrumentStudio to trigger the event. For example, clicking the Stop all outputs button in the toolbar of the soft front panel document will trigger the Stop All Outputs event for the plug-in and can only be done from within Instrument Studio. In these cases, you need to be mindful to manually switch back to LabVIEW if you have breakpoints enabled that you expect to break when the event fires. Since LabVIEW does not provide any indication or alert that execution of the plug-in VI has paused at a breakpoint, it can be confusing as to why the panel in InstrumentStudio appears to no longer be working. If the plug-in does not respond to interaction from within InstrumentStudio, double check that execution is not currently paused at a breakpoint in LabVIEW.

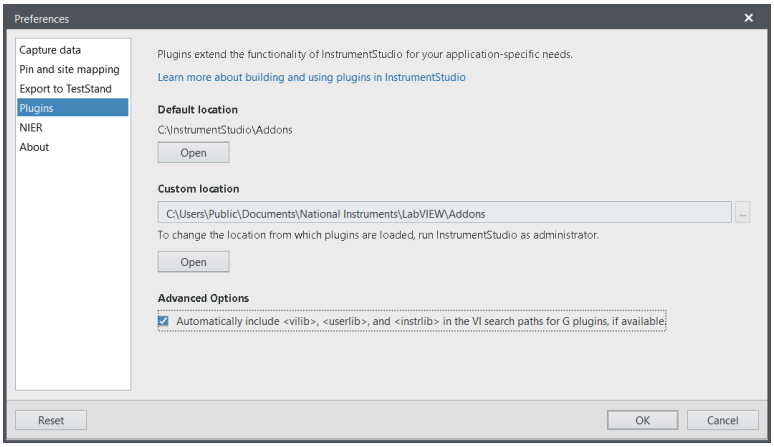
As you debug the plug-in and need to make changes to the source code, you will need to quit the remote debug session, make the appropriate edits, save, rebuild, and redeploy the plug-in to InstrumentStudio. From InstrumentStudio, you will then need to close all documents containing an instance of the plug-in panel to ensure all previous versions have been unloaded from memory. At this point, any new instances of the plug-in panel that you create should include the latest changes made to the source code.

To shorten the test/debug/fix cycle, there are a couple of techniques plug-in authors can utilize. The first and simplest thing is to skip redeploying the plug-in to the plug-in directory of InstrumentStudio after each build of the plug-in in LabVIEW. This can be accomplished by creating a debug .gplugindata file with a debug PluginData entry where VIPath contains an absolute path to the output of the build directory rather than a relative path from the .gplugindata file. As this technique does not provide portability between machines, it should only be used during development and never for a release plug-in.

If the time it takes to build a plug-in is longer than desired, you can also attempt to run the plug-in directly from source rather than performing a full rebuild with each source change. However, the effort involved to get this working successfully will vary with the complexity and number of dependencies used by the plug-in. Because of this, the effort to get this working may not be worth it unless there is a significant time savings between debug iterations. To run from source, simply use the same technique described previously and update the VIPath to contain an absolute path to the VI in the source directory rather than the build output directory.

The biggest stumbling blocks when trying to run from source are caused by source only VIs and missing dependencies. If the plug-in or any of its dependencies use source only VIs, those VIs must have been compiled once on the machine so that the compiled object cache has been populated with code that can be executed by the runtime engine. If a matching entry in the compile cache cannot be found, then the plug-in will not be executable. This issue is becoming more common as libraries from vi.lib are increasingly being installed as source only.

Missing dependency failures when running from source is often due to differences between VI search path resolution when running in the LabVIEW IDE vs. the LabVIEW Runtime Engine. For instance, when using dependencies from vi.lib, the path to that dependency is stored relative to a symbolic root for vi.lib. When running in the LabVIEW IDE, this symbolic path can be resolved successfully to an absolute path on disk. However, when running in the LabVIEW Runtime Engine, there is no way to resolve this symbolic path, and the dependency will not be found. To resolve this, you either need to copy the dependency into the build output as part of your build specification or update the LabVIEW Runtime Engine's VI search paths. To update the default search paths of the LabVIEW Runtime Engine used by InstrumentStudio, you can enable the checkbox under the Advanced Options of the Plugins preferences page as shown below.



Enabling this option will only work if the same version of the LabVIEW IDE is installed as the LabVIEW Runtime Engine which is being used by Instrument Studio. This option is only intended to be used by plug-in authors during plug-in development and should not be relied upon for released plug-ins. Use this option with care as it can mask real issues in your build specification / installer due to missing dependencies. You should always disable this option when performing final testing of your build specification / installer or perform final validation on a machine that only contains the LabVIEW Runtime Engine and not the full LabVIEW IDE.

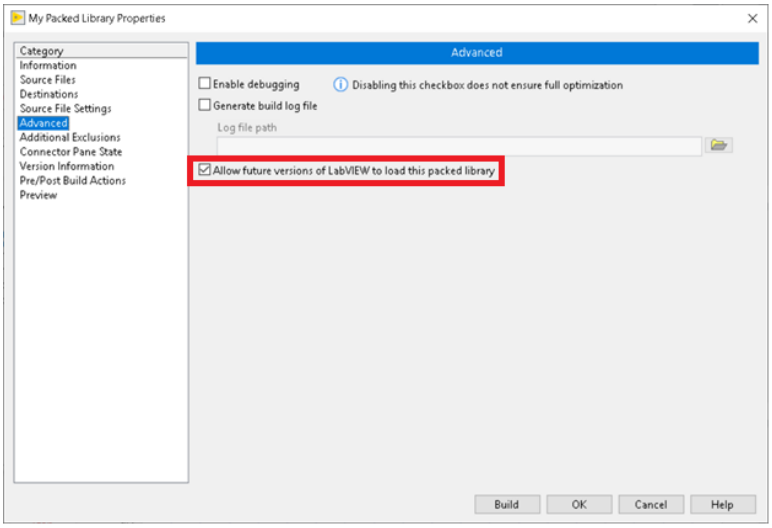
If you resolve all issues caused by missing VI dependencies, you may still encounter missing dependency errors due to shared libraries (i.e., dlls). The search algorithm used to find the target dll of a Call Library Node depends on whether the dll is specified by path or name. If the dll is specified by name, then the search algorithm used by the operating system comes into play in addition to the search algorithm used by LabVIEW. Because of this, when running in the LabVIEW IDE vs. the LabVIEW Runtime Engine, it is possible the dll will be found in one case but not the other. To resolve missing library errors, you will likely need to either create multiple copies of the dll on disk for the various deployment scenarios, update the VI search paths, or update the Windows Path variable.

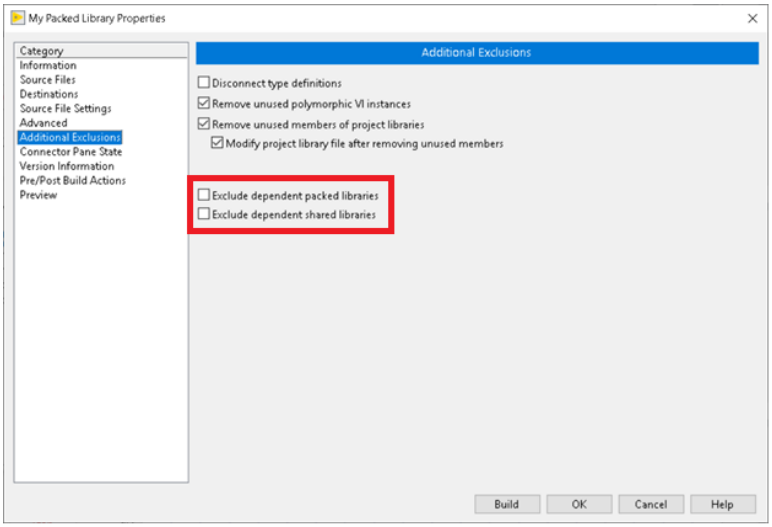
Building and Deploying Release Plug-Ins

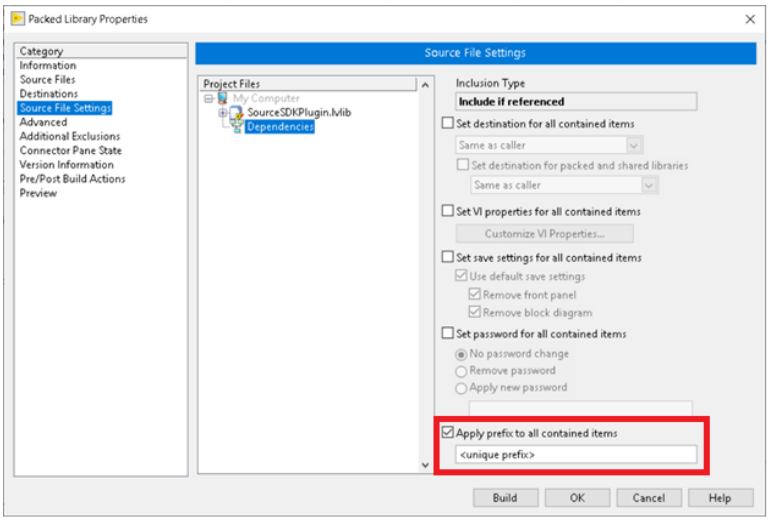
Release plug-ins deployed to InstrumentStudio should always be produced from either a Source Distribution or Packed Library Build Specification in LabVIEW.

For most plug-ins, it is preferable to deploy using a packed library as all source files will get packaged into a single binary file for distribution. Packed libraries are also beneficial in that any source dependencies built into the library are also namespaced by the library name. This provides further isolation and makes it easier to have multiple plug-ins running in the same application context that were built against different versions of the same source. When setting up a packed library build specification, pay particular attention to the following settings.

- Allow future versions of LabVIEW to load this packed library – This option should always be enabled. InstrumentStudio only supports loading of plug-ins using a single version of the LabVIEW Runtime Engine. If this option is not checked, the plug-in will fail to load if InstrumentStudio uses a different version of the runtime engine than the version the plug-in was built against. If multiple versions of the LabVIEW Runtime Engine are installed on the machine, InstrumentStudio will use the newest version.
- Exclude dependent packed libraries – This option should typically be disabled. If this option is enabled, then any packed libraries used by the plug-in will not be copied into the build output and will need to be distributed through some other mechanism.
- Exclude dependent shared libraries – This setting will be plug-in dependent. If the plug-in calls into any shared library other than NationalInstruments.VIHost.Interop.dll, then this option is typically disabled unless those libraries are system libraries that will be distributed through some other mechanism such as the installation of a driver.
    - Apply prefix to all contained items – If including dependent shared libraries and the shared library is private to the plug-in and not part of a system path, you may also want to add a prefix to the library name as part of the build if it can be installed by multiple plug-ins to different locations on disk. Adding a unique prefix to the library prevents name collisions when loaded into memory and ensures the plug-in uses the exact version it was built against rather than another version installed by a different plug-in.







When using a build specification for a source distribution, the same settings described for a packed library apply. In addition, you also need to worry about name collisions of all source VIs used to

implement the plug-in as they are loaded into memory. If the plug-in executes in a unique application context as specified by its .gplugindata file, then there is no possibility for name conflicts. Therefore, it is recommended that plug-ins execute in unique application contexts whenever possible. Otherwise, the plug-in should organize its source VIs into libraries (.lvlib files) to help ensure there are no name conflicts within the application context in which it is loaded. Alternatively, the build specification can be used to generate a unique name by applying a prefix to each source VI as part of the build.

In addition to the code needed to run the plug-in, you will also need to include the .gplugindata file as part of the build. The .gplugindata file should be copied to the build output so that the relative path to the plug-in specified in the .gplugindata matches the directory structure produced by the build. When using a packed library, the path specified for VIPath in the .gplugindata file should be of the form *<directory of packed library>\<packed library name>\<relative path to VI from project file in which it was built>*. Absolute paths should never be used for release plug-ins as they are not portable from one machine to another.

Once the source distribution or packed library has been built, you will need to create an installer that will take the output of the build specification and copy it to a plug-in directory within InstrumentStudio. To accomplish this, you can use LabVIEW to create a build specification for an installer or package depending on your preferred distribution model. You can also use third party tools to create an installer. As part of the installer, you should include installation of the version of the LabVIEW Runtime Engine that is required to run your plug-in. InstrumentStudio will not always install the LabVIEW Runtime Engine depending on the installation options chosen at install time. Even if InstrumentStudio installs a LabVIEW Runtime Engine, there is no guarantee it will meet the minimum version required for the plug-in. Because of this, it is recommended the installer for the plug-in always includes the LabVIEW Runtime Engine.

Plug-In Panel Limitations and Layout Considerations

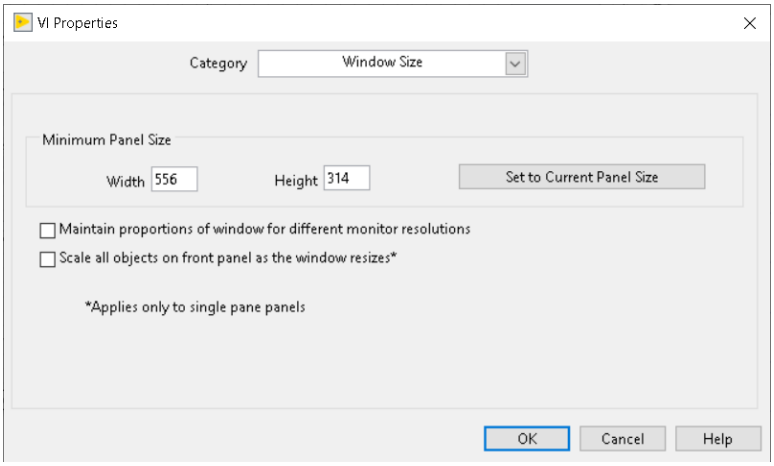
Due to the implementation and design of how LabVIEW front panels are hosted inside InstrumentStudio, the following limitations and behavioral differences exist for plug-in panels.

- No vectorized scaling of the plug-in panel is supported. For example, when changing the DPI of the system, the panel may appear stretched and blurry in areas. This is a limitation of LabVIEW front panels.
- ActiveX and .Net container controls on the LabVIEW front panel are not supported.
- Tabbing between controls on the plug-in panel using the tab key only works if one of the controls is active for keyboard input. For example, if you click a numeric control and activate it for keyboard entry, you can then use the tab key to navigate between controls. However, if you just click the front panel of the plug-in and use the tab key, no navigation between controls will occur.
- If the plug-in panel has focus within a soft front panel document, and the document has visible scrollbars, using the arrow keys to scroll the document will not work. Similarly, if the plug-in panel itself has visible scrollbars, using the arrow keys to scroll the panel will not work.
- Using the mouse scroll wheel while hovering over the plug-in panel will only scroll the contents of the plug-in vertically within the panel. If the panel does not have a visible vertical scrollbar,

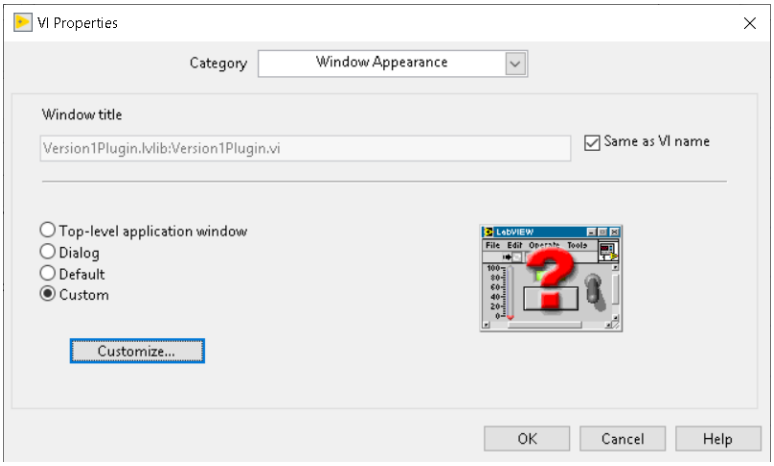
scrolling the mouse wheel will have no impact and cannot be used to scroll the soft front panel vertically.

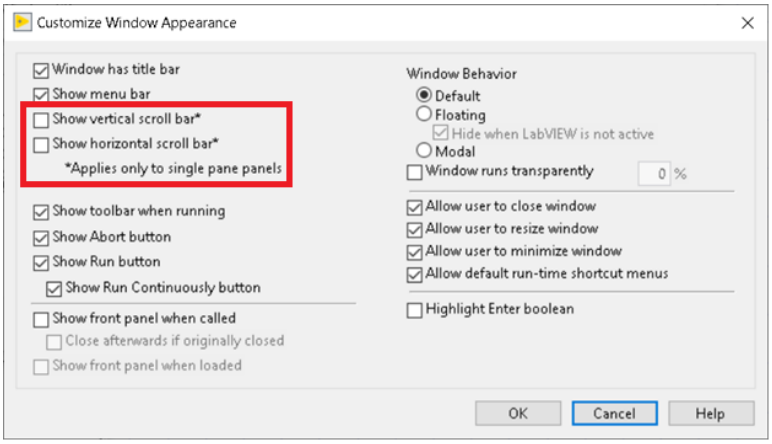
- Only the keyboard commands for cut (Ctrl + X), copy (Ctrl + C), and paste (Ctrl + V) will be recognized by the LabVIEW front panel. All other keyboard commands will be routed to InstrumentStudio for handling.
- While launching modal dialogs from the plug-in is supported, dynamically opening VIs and showing their front panel should be avoided. The window management for the floating plug-in front panels and InstrumentStudio will not work well together, and it generally breaks the intended user experience for InstrumentStudio.

When displaying a plug-in panel within InstrumentStudio, vertical and horizontal scrollbars will automatically become visible as needed based on the current layout of the soft front panel and the minimum size specified for the front panel of the plug-in VI. If the space allocated to the plug-in panel is smaller than the minimum size specified by the plug-in VI, then scrollbars will become visible. If the space allocated to the plug-in panel is larger than the minimum size specified by the plug-in VI, then no scrollbars will be visible and any controls on the front panel that scale with size will scale to fill the space allocated to the panel. Because of this, it is recommended that plug-ins always assign a minimum size for the plug-in VI and test with a variety of layouts to ensure the appearance of the panel meets expectations. Setting the minimum size can be done from within the Window Size category of the VI Properties dialog as shown below.



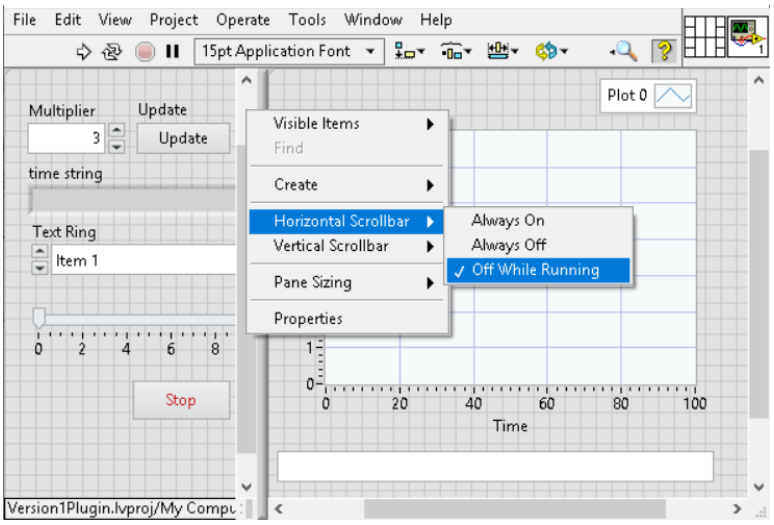
Unless the front panel of the plug-in is specifically designed to be scrolled, plug-in VIs should also specifically disable run time scrollbars on the front panel. This can also be done from the VI properties dialog as shown below.





If you do not disable scrollbars on the front panel, LabVIEW always displays the scrollbar even if the front panel is large enough such that no scrolling is required.

If the front panel of the plug-in utilizes splitters, you can specify minimum sizes for each split panel in addition to the entire front panel. However, InstrumentStudio will only honor the minimum size specified for the entire front panel, even if that is smaller than the combined minimum sizes of each split panel. To prevent display issues due to this, ensure the minimum size of the front panel is always at least as big as the combined minimum sizes of the individual panels that comprise it. Also, when splitters are used, you cannot disable scrollbars from the VI Properties dialog. Instead, you must disable it for each split panel as shown below.


File Edit View Project Operate Tools Window Help
15pt Application Font
Multiplier
3
Update
Update
time string
Text Ring
Item 1
0 2 4 6 8
Stop
Plot 0
Visible Items
Find
Create
Horizontal Scrollbar
Vertical Scrollbar
Pane Sizing
Properties
Always On
Always Off
Off While Running
1
0
0 20 40 60 80 100
Time
Version1Plugin.lvproj/My Compu