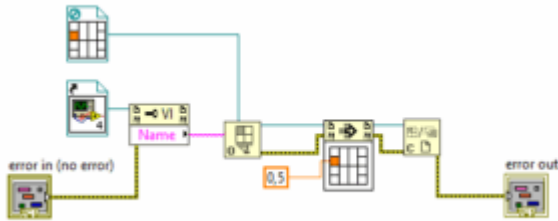


LabVIEW Dynamic VI calls

Category: Insight,News

written by Davide Oratelli | February 12, 2025



In LabVIEW, dynamic VI (Virtual Instruments) calls allow for flexible loading and execution of VIs during application runtime, offering significant advantages in terms of modularity and memory management.

This technique is particularly useful in complex applications that require conditional execution of modules or the extension of functionalities through plugins.

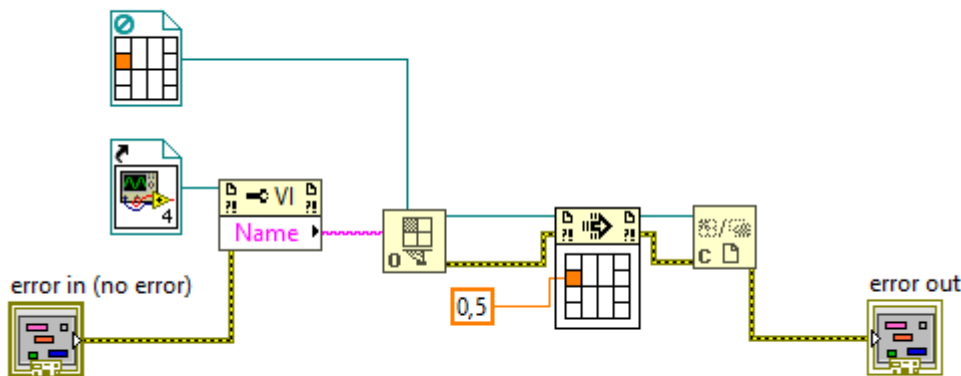
Modes of Dynamic VI Loading

Synchronous Call via “Call By Reference”

This mode uses the ‘Call By Reference’ node to execute a VI specified by a strictly typed reference. The called VI is executed synchronously, blocking the data flow until execution is complete.

To implement this technique:

- Provide the necessary input data and handle the outputs.
- Use the ‘Open VI Reference’ function to obtain a reference to the target VI.
- Connect the reference to the ‘Call By Reference’ node.



Asynchronous Call via “Start Asynchronous Call”

This technique allows the calling VI to continue execution without waiting for the called VI to complete (Call and Forget).

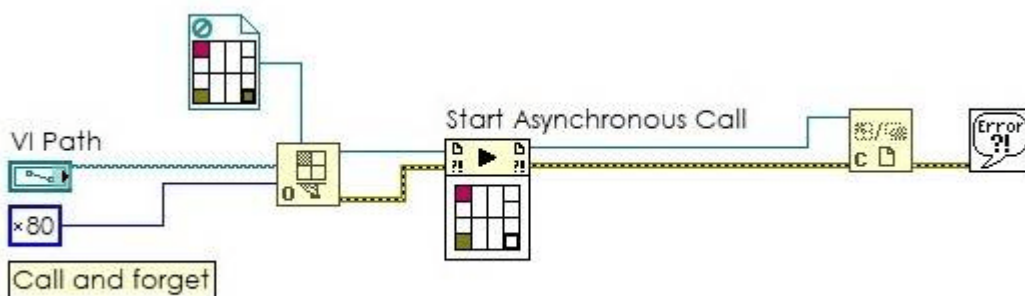
There are two main calling modes:

Call and Forget

The called VI is launched, and the caller neither waits for nor collects the execution results.

To implement this mode:

- Set the 'options' terminal of the 'Open VI Reference' function to the value 0x80.
- Use the 'Start Asynchronous Call' node to launch the VI.



Call and Collect

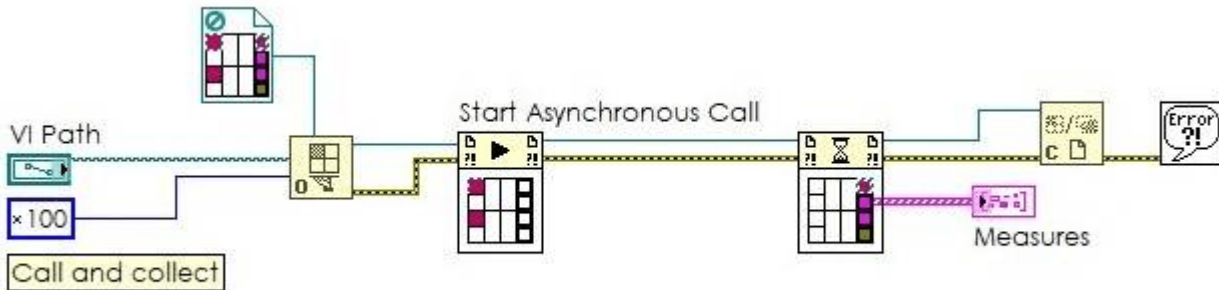
The called VI is launched, and the caller can subsequently collect the execution results.

To implement this mode:

Set the 'options' terminal of the 'Open VI Reference' function to the value 0x100.

Use the 'Start Asynchronous Call' node to launch the VI.

Then, use 'Wait on Asynchronous Call' to collect the results.

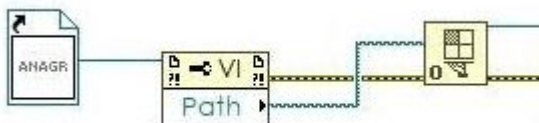


Call via Static Reference

An alternative to dynamic loading is to use a static reference to the VI. This approach does not require specifying the VI's path, as it directly uses a static reference.

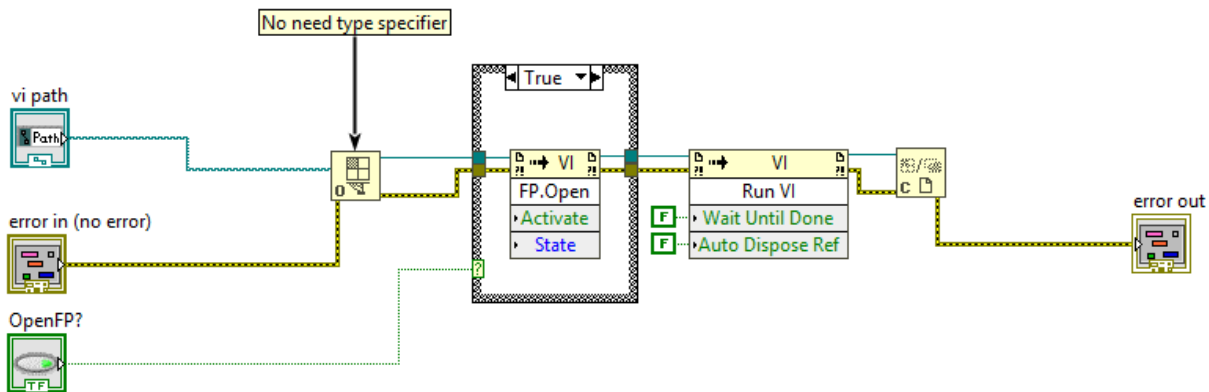
Steps to implement it:

- Place a 'Static VI Reference' constant on the block diagram.
- Use the 'VI Path' property to obtain the VI's path.
- Open the reference to the VI using 'Open VI Reference' based on the obtained path.
- Invoke the method associated with the opened VI called "Run VI."



Call with Run VI Method

The Run VI method allows starting the execution of a VI similarly to the "RUN" button, using the current values of all front panel controls instead of data passed through parameters.



This approach, which ignores the Execution:Show Front Panel On Call and Execution:Close After Call properties, offers greater flexibility: a strictly typed reference is not required, allowing the execution of any VI, even those that do not share the same connector pane.

Additionally, the Run VI method enables interaction with the VI while it is running, for example, to monitor or repeatedly read outputs.

However, if you want to pass values to the VI, you need to know the names of the controls and their data types, which requires additional programming.

PRO

Does not require a strictly typed reference, allowing any VI reference to be passed.

Starts the VI using the current values of the front panel controls.

Enables real-time interaction during execution, similar to the “Run” button.

CONS

To pass values, you need to know the names and data types of the controls, requiring additional programming.

If using the Call by Reference node, values are emitted only at the end of execution, limiting interaction.

Comparison table

Characteristics	Run VI	Start Asynchronous VI	Wait on Asynchronous Call	Call via SubVI
Type of execution	Starts the VI immediately, similar to the “Run” button.	Starts the VI asynchronously, allowing it to run in the background without blocking the caller’s flow.	Waits for the completion of the VI started asynchronously, blocking the caller’s flow until the end of execution.	Synchronous execution: The VI is called directly as part of the code flow, with inline execution.

Parameter and data management	Uses the current values of the front panel; to pass values you need to know the names and types of the controls.	It can be configured to use parameters or the current values of the front panel, based on the logic implemented in the VI.	Retrieves output from the VI after asynchronous execution completes, and then processes the resulting data.	Parameters are passed via wiring in a strictly typed manner, ensuring compile-time checking.
Interaction with the VI	Allows direct interaction with the front panel during execution, allowing real-time updates.	Execution occurs in the background, providing no direct interaction while the VI runs.	It does not allow interactions until the asynchronous VI finishes, as the flow waits to synchronize.	The VI's front panel is not displayed during execution; interaction occurs exclusively through input and output data.
Typical use	Ideal for dynamically starting a VI when you want to interact in real time with the front panel and use the current values.	Useful in applications that require concurrent or non-blocking execution, where the VI needs to run in the background.	Used to synchronize the flow of execution, waiting for an asynchronously started VI to complete before proceeding with the data.	Suitable for static VI calls, when you need strict control of the interface and data types thanks to the strictly typed connection.
Additional Notes	Ignore the proprietary <i>Execution:Show Front Panel On</i> and <i>Execution:Close After Call</i> .	It allows you to manage VIs in parallel thanks to the asynchronous token, but requires specific synchronization management.	It is closely tied to the token returned by the Start Asynchronous VI and ensures that the outputs are captured correctly when finished.	It provides the benefit of compile-time verification and built-in interface management, although it does not offer the dynamic flexibility of the Run VI methods.

Advantages of Dynamic VI Loading

- **Modularity:** facilitates the integration of independent components.
- **Memory management:** Load VIs only when needed.
- **Flexibility:** Allows you to implement plugin architectures.

Considerations and Limitations

- Reference Management: Avoid memory leaks by properly managing references.
- Synchronization: Required to avoid conflicts in accessing shared resources.
- Debugging: More complex than static VIs.

Application Examples

Parallel Process Management: Improving Performance.

Conditional Loading: Run VIs only when needed.

Plugin Architectures: To expand the functionality of an application.