**Actor Framework Project Providers**

**Design Reference**

**Overview**

This document describes the set of LabVIEW Project Providers that support the Actor Framework. It is intended to provide an overview of their structure, theory of operation, and implementation, so that future developers can effectively maintain and extend them.

**About Project Providers**

A project provider is a plug-in to the LabVIEW Project window that extends the window’s functionality. The new functionality can apply to all items in a project, or to specific item types. Primary providers add new item types to the project tree, while secondary providers add functionality to existing items.

The Actor Framework includes primary providers for creating actors and actor interfaces, and secondary providers for creating messages.

Provider development is outside the scope of this document. Users are referred to the *LabVIEW Project Provider Developer’s Guide*, which can be found at the LabVIEW Project Provider forum. See Reference Links, at the end of this document.

**About VI Scripting**

Project providers make extensive use of VI Scripting to do their work. A general discussion of VI Scripting is outside the scope of this document. Please see Reference Links, at the end of this document for links on how to get started.

You must enable VI scripting in your LabVIEW IDE to work effectively with these project providers. From any LabVIEW menu, select Tools » Options…, and then navigate to the VI Server page. Select “Show VI Scripting functions, properties and methods” as shown, and make sure “Display additional VI Scripting information in Context Help window” is selected.



We also recommend adding the following line to your LabVIEW INI file:

SuperSecretPrivateSpecialStuff=true

**Messages in Actor Framework**

A brief review of the types of message classes created by the providers is helpful in understanding the code artifacts the providers create.

In Actor Framework, a message is a class that inherits from **Message.lvclass**. It implements an override of **Do.vi**, a dynamic dispatch method of Message.lvclass. This message invokes a method of the message’s target actor. When created by the project provider, the message name is of the form <method name> Msg.lvclass. The message class itself includes as attributes all of the inputs to the method targeted by the message (except for the target actor and standard error input).

Messages created by the provider also include a method that writes data to an instance of the message class, and puts that object on the actor’s enqueuer. The name of this method is Send <method name>.vi.

The providers will also create abstract messages and their child concrete implementations. (See the note about abstract messages immediately following this section.)

Abstract messages do not target a specific actor’s method. They include a set of attributes defined by the developer and a Send VI. The Send VI writes attributes to a child of the abstract message, and puts that object on an actor’s enqueuer. The child class is specified by an input to the Send VI. Abstract messages do NOT include an override of Do.vi. Abstract messages are named by the developer.

Abstract messages also provide a read accessor that returns all of the message’s class data. This VI used in the Do.vi methods of children of the abstract message.

A concrete child message inherits from an abstract message. Like the standard message, a concrete child implements a Do.vi that invokes a method of a specific target actor. Concrete children are named in the same fashion as standard messages.

The following diagram shows the artifacts created for each message type.

****

**A Note on Abstract Messages**

Abstract messages and their concrete children are **deprecated** in Actor Framework applications created in LabVIEW 2020 SP1 or later. Developers are strongly urged to use actor interfaces instead.

However, we will continue to maintain the abstract message provider code in the Actor Framework Project Provider so that developers can continue to work efficiently with older code bases.

**The Project Providers**

The role of a project provider is to provide the user with right-click menu options for its project items, and then to invoke the VI Scripting code (or other operations) associated with the user’s menu selection. When a user right-clicks on a project item, the provider first validates the item as one on which it can act. If the item is valid, the provider adds choices to the right click menu. Should the user select one of those choices, the provider then queries the project item for any relevant attributes, and passes those attributes to the appropriate scripting code.

**A Note on Context and Application Instances**

LabVIEW project providers run in their own instance of LabVIEW, completely separate from the instance in which application code runs. These instances don’t share memory or other resources, which avoids the potential for unfortunate interactions between provider code and application code. Most scripting operations, however, *must* be performed on code in the target item’s application instance. Developers must diligently mange the provider and application instances to avoid bugs and performance issues.

The most common indication of a context problem is that code that works in test no longer works when executed in the provider. The original provider developer also experienced issues with copying certain data types (variants, in particular) across the context boundary; some of these issues may still exist.

For reference, provider codes runs in the NI.LV.MxLvProvider instance. Elements of the provider API that are intended to run in that instance have a “mxLv” prefix.

**Available Providers**

The Actor Framework Project Provider suite consists of five individual providers. Two primary providers create actors or actor interfaces, while three secondary providers create or refactor messages. The secondary providers each provide options for a single project item: an actor, an actor’s methods, or an existing message.

In theory, a single secondary provider can act on several types of project items, but this was shown during development to be difficult, so it was decided to create a different provider for each type.

Each provider is described individually, below.

Add Actor

****

Add Actor acts on Targets and Libraries, and creates a new actor under that target/library. When placing a new actor under a target, it creates a library for the actor, and includes a virtual method for messages created for that actor.

Add Actor does not create any messages, and does not invoke any messages from the core Message Maker library (see next section).

Add Actor Interface

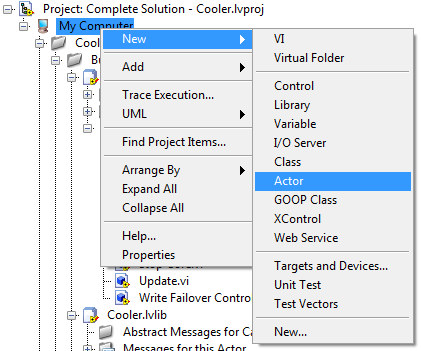
****

Add Actor Interface acts on Targets and Libraries, and creates a new interface under that target/library. When placing a new interface under a target, it creates a library for the interface, and includes a virtual method for messages created for that actor.

The option to create a new actor interface is not available in projects that contain no actors.

Add Actor Interface does not create any messages, and does not invoke any messages from the core Message Maker library (see next section).

Message Maker Provider



Message Maker Provider creates messages for one or more actor methods. Several methods may be selected, and the methods may span multiple actors.

If a single method is selected, the developer also has the option to create a child of an abstract message.

Actor Message Maker

Actor Message Maker Provider creates a message for each method in one or more actors. Several actors may be selected.

If a single actor is selected, the developer also has the option to create an abstract message.

Message Rescripter



Message Rescripter acts on an existing Actor Framework message. It rebuilds the existing message in place.

**Architecture**

The following table associates each project provider with its corresponding LabVIEW library file and configuration file.

|  |  |  |
| --- | --- | --- |
| Provider | Library | INI File |
| Add Actor | Add Actor.lvlib | AddActor.ini |
| Add Actor Interface | Add Actor Interface.lvlib | AddActorInterface.ini |
| Messager Maker Provider | Message Maker Provider.lvlib | MessageMaker.ini |
| Actor Message Maker Provider | Actor Message Maker Provider.lvlib | ActorMessageMaker.ini |
| Message Rescripter | Message Rescripter.lvlib | MessageRescripter.in |

Two additional libraries complete the provider suite. **Message Maker.lvlib** contains most of the VI Scripting code that actually creates messages. **AFPP Shared.lvlib** contains several small scripting VIs shared over several packages. Most of the few password-protected VIs in the suite are contained in this library.

The dependency relationships between the libraries are shown in this diagram.

****

(**Note:** This diagram shows package relationships as they will exist following resolution of Issue #NN, which has not been resolved as of this writing. Please remove this note as part of resolving the issue.)

As described in the *LabVIEW Project Provider Developer’s Guide*, providers consist of a set of VIs that plug in to the provider framework. These VIs perform the various services included in the provider. If a provider does not perform one of those services, it is theoretically not required to provide a VI for the service. This has not proven to be strictly true, and the Actor Framework Project Providers contain a few VIs that do nothing other than fill a slot the framework seems to require.

This suite of providers is focused solely on providing right click options to construct actors or messages, and then invoking scripting code to carry out that construction.

**Primary Providers**

The suite’s primary providers include Add Actor and Add Actor Interface. The function nearly identically.

Populating Project Right-Click Menus

The provider framework invokes **CreateNewWizard\_IncludeItem.vi** when the user right-clicks on an item. Framework API calls provide a reference to the project and the unique ID for the selected item; scripting code then determines the target type string. If the type string is found on a white list of approved targets, the VI returns TRUE.

Add Actor Interface’s version of **CreateNewWizard\_IncludeItem.vi** also checks to determine if the project contains any child of Actor.lvclass. The VI returns TRUE only if a child of Actor.lvclass is present and the target is on the white list.

If **CreateNewWizard\_IncludeItem.vi** returns TRUE, the provider next invokes **CreateNewWizard\_Init.vi**. This VI simply provides the name of the artifact to add to the “New” menu (either “Actor” or “Interface for Actor”). Note that this string, like many other strings in the suite, is stored in a global variable, to facilitate localization. For Add Actor and Add Actor Interface, this global variable is **Add Actor.lvlib:Localized Strings.vi**.

Executing Scripting Code

Both primary providers have a version of **CreateNewWizard\_Invoke.vi** to create their artifacts. Add Actor’s version delegates that responsibility to **CreateNewWizard\_Invoke Core.vi**; this VI can create both Actors and Actor Interfaces.

These VIs invoke either **Add Actor.vi** or **Add Interface Dialog.vi** to gather user input and perform actual scripting operations. See Creating Actors and Actor Interfaces, below.

**(Note:** Add Actor Interface.lvlib relies on scripting code contained in Add Actor.lvlib, but Add Actor.lvlib:CreateNewWizard\_Invoke Core.vi invokes a VI contained in Add Actor Interface.lvlib. This creates a circular dependency that should be broken at some point in the future. Since the two providers are so similar, it may make the most sense to create a separate library that contains all the scripting code used by both providers, in the way that Message Maker.lvlib is used by all of the secondary providers.)

**Secondary Providers**

Message Maker Provider, Actor Message Maker Provider, and Message Rescripter are the suite’s secondary providers. They are responsible for creating and updating messages for actors and actor interfaces.

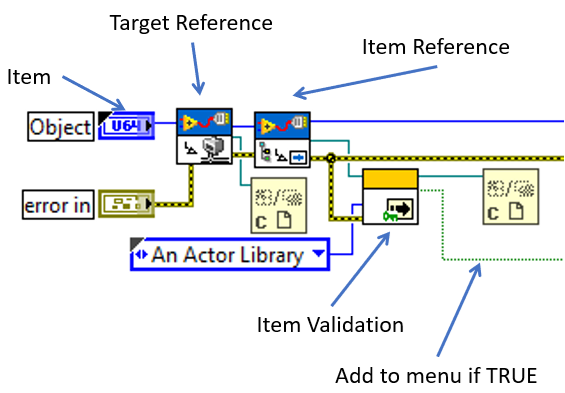
Populating Project Right-Click Menus

When you right click on a project item, LabVIEW finds all the providers that support the item’s type. For each such provider, LabVIEW invokes the VI **Item\_OnPopupMenu.vi**, which adds items to the popup menu.

Since Message Maker Provider and Actor Message Maker Provider can act on several project items when activated, they also provide versions of **Provider\_OnPopupMenu.vi**. These VIs work in much the same way as their respective Item\_OnPopupMenu VIs, but they provide slightly different menu options.

Since actors are just LabVIEW classes, and actor methods are just VIs, actor framework providers must perform additional validation on the selected project item, to ensure they are actor framework artifacts.

This diagram, taken from **Message Maker Provider.lvlib:Item\_OnPopupMenu.vi**, is typical:



OnPopupMenu.vi uses the provider framework API to obtain a reference to the project item. It passes this reference to an item validation VI, which indicates whether the provider should add choices to the right-click menu. Since each provider targets a different project item type, they each use a different validation VI. As of this writing, each validation VI is found in its respective provider’s library.

**Note:** OnPopupMenu.vi is invoked when the user right-clicks on a project item, and must complete before the right-click menu is displayed. Furthermore, LabVIEW invokes OnPopupMenu.vi for *every* provider associated with that project item type. For this reason, it is *imperative* that any operation performed in an OnPopupMenu.vi execute quickly. The total execution time for all OnPopupMenu VIs for the target item should be less than 100 ms.



Message Maker Provider invokes **Is Actor Public Method.vi**, which, as its name implies, returns true if the selected project item is a public method of a class that inherits from Actor.lvclass

This VI contains scripting nodes that are private to NI, so it is password-protected.



Since it acts on actors and actor interfaces, Actor Message Maker Provider invokes **Is Actor.vi**. This VI will determine if the project item is an actor or an actor interface, and if it has public methods.



Message Rescripter operates on messages, so it validates project items by invoking **Is Message.vi**, which returns true if the project item inherits from Message.lvclass.

These VIs are invoked again in message creation (see below).

Is Actor Public Method and Is Actor can validate by checking to see if the item or its owner is an actor library, or if its owner has been tagged with the path to an Actor Framework PPL. (Is Message ignores this input.) Tags are metadata that can be associated with a project item through scripting; they can be of any data type. In the provider suite, when an actor or message inherits from a class in a PPL, the actor gets tagged with the path to the PPL. Accessing a tag is a single scripting call, which is much faster than searching for a PPL. The tags are used during message creation, and offer a small performance improvement.

Shown here are the nodes to get and set a target item’s tag. The set example is the code used by the provider.





**AFPP Shared.lvlib:Update Target Tag – PPL Path.vi** sets and clears the PPL path tag on actors, and is invoked in several different entry points to the message scripting process.

Executing Scripting Code

When the user selects an option to create a message, the provider framework invokes the provider’s Item\_OnCommand or Provider\_OnCommand (depending on the number of project items selected).

**Preparing for Message Construction**

#TODO how do I want to bridge the space between OnCommand and actual message scripting? Come back to this after completing the Creating Messages section.

**Message Maker.lvlib:Message Scripter.lvclass:Create Message.vi** is thetemplate method that governs creating a message. This VI invoked at the bottom of a call chain of several VIs whose purpose is to select the construction classes (scripters, prototypers, and builders) required to construct the desired message, and to provide those classes with data about the target actor and method and message attributes. Create Message.vi, and the construction classes it uses, are described in detail in Creating Messages, below. This section concerns itself with the call chains that invoke Create Message.vi.

The call chains are different for each message class. They are listed and described in the following tables. The first row of the table is the VI invoked by an OnCommand VI; the last row is the VI that invokes Create Message.vi. Unless otherwise noted, these VIs are part of the provider library that invokes them.

**Error Handling**

Project providers must provide their own error handling and reporting, as the provider framework has no default behaviors. The primary and secondary providers in this suite handle errors somewhat differently.

Primary Providers

The scripting code for Add Actor and Add Actor Interface is relatively simple and straightforward, and most errors involve user choices incompatible with the state of the users’ project file. The scripting functions are called directly from a user interface (see below), so many errors are handled in the moment; users are notified of the error with One Button Dialog boxes, allowing them to cure the error and retry the operation. In the rare instance of a scripting error, users are notified via the Simple Error handler, invoked at the end of CreateNewWizard\_Invoke.vi or CreateNewWizard\_Invoke Core.vi.

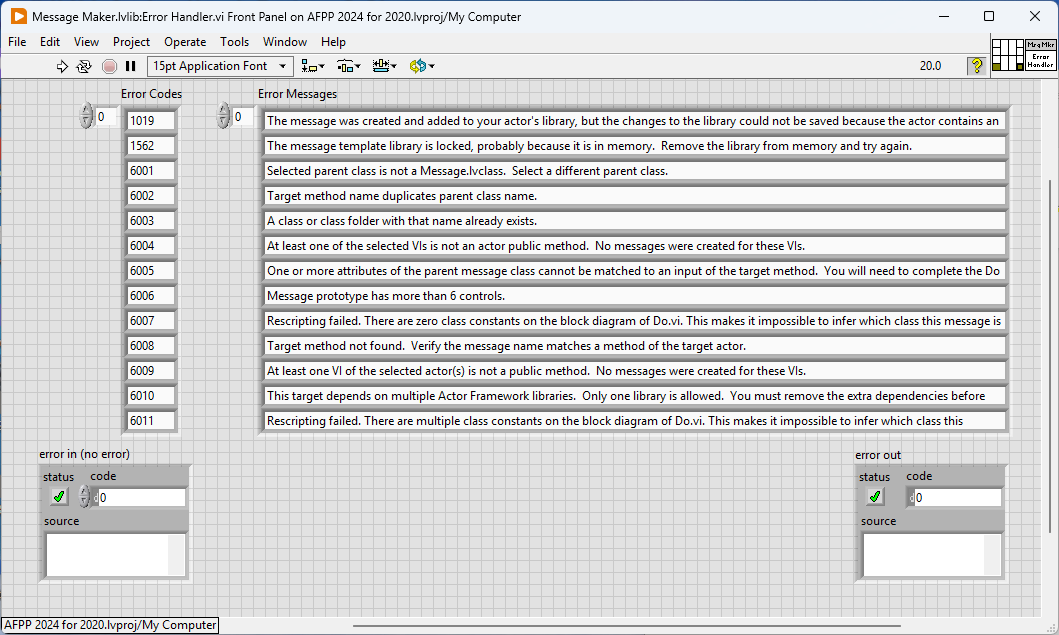
Secondary Providers

Creating a message is much more complicated than creating an actor or actor interface, and message creation is invoked directly from the provider framework. Errors generated by the scripting code are thus forwarded to an error handler, rather than handled in the moment.

Some standard LabVIEW error codes have specific meaning in the context of message creation. Furthermore, a number of the scripting code can calculate one of several error conditions when it encounters a conflict. Calculated errors are assigned an error code in the 6000 – 6100 range. (As all error values are consumed within the providers themselves, there is no risk of conflict with error codes users may assign in their applications.)

Scripting code is invoked by Provider\_OnCommand and Item\_OnCommand VIs; each such VI passes any error output to **Message Maker.lvlib:Error Handler.vi**. This VI specifies error codes with special meaning in this context; where such a code is detected, the handler presents the user with a One Button Dialog box with the appropriate error message. If an error code is not on the list, the error is passed to Simple Error Handler.vi

Here is the front panel of Error Handler.vi, showing the currently supported error codes. As of this writing, the calculated error codes range from 6000 to 6011.



**Creating Actors and Interfaces for Actors**

**Add Actor.lvlib:Add Actor.vi** and **Add Interface Dialog.vi** are the VIs invoked to create actors and actor interfaces. Each is a user interface that collects information about and then creates the new actor/actor interface. Most of the code on the block diagrams of these VIs is related to managing the user interface.

Once the user clicks OK, these VIs invoke **Add Actor.lvlib:Add Actor Library to Project.vi**, followed by either **Add Actor.lvlib:Create Child Actor.vi** or **Add Actor Interface.lvlib:Create Interface.vi**.

The Add Actor provider supports creating actors that inherit from other children of Actor.lvclass. These other actors can be source code or included in a PPL. Add Actor.vi uses an **Actor Store.lvclass** to maintain a list of actors previously selected for inheritance. **INI Store.lvclass** maintains the store as entries in the LabVIEW INI file. **Target Store.lvlcass** maintains the store as tags in the project file. Add Actor.lvlib selects which store to use when it is invoked. Target Store.lvlcass is selected if an Actor Framework PPL is detected in the project; otherwise, INI Store.lvlcass is selected.

**Creating Messages**

**Message Templates**

The providers use template classes to build specific messages of a given type. Templates are included for standard, abstract, and concrete messages, as shown.

Note that these class templates are not part of a library, nor do they inherit from Message.lvclass. It is much easier to work with an uncoupled class than one that is linked to other code artifacts.

To create a message, the providers copy the appropriate template to the specified location, modify its VIs as necessary, change it to inherit from Message.lvclass, and then add it to the actor’s library.

The template to be used is selected based on the provider action invoked by the user.



**Required Data**

To create a message, the developer right-clicks on an actor or message and selects an option. The selected option determines the message type (and thus the starting template). The provider then then obtains additional information about the selected actor or method that is used to complete message construction:

1. Target Actor (reference or path): the actor for which the new message is being created.
2. Method Path: the path to the method to be invoked by the new message. In the case of abstract messages, the method path is the path to a prototype method (see Prototyping, below).
3. Class Name: the name of the new message class. For an abstract message, this is specified by the developer; otherwise, it is derived from the target method.
4. Parent Class Path: the path to the abstract parent class for this message (Concrete child only).

**Scripters, Builders, and Prototypers**

As mentioned above, the provider suite supports the creation of three types of messages: standard messages, abstract messages, and concrete messages (which are children of abstract messages). A standard message includes a Send VI and a Do VI. Abstracts include a variant of a Send VI but no Do; instead they provide a Read Attributes VI which is used by their Concrete children. Concrete children provide a Do (which uses their parents’ Read Attributes), but no Send.

The following diagram summarizes the artifacts created for each message type.



Scripting operations for these artifacts are very similar (where present) across all three message types. **Message Scripter.lvclass**, a template method abstract class, defines the general algorithm (detailed below) and performs scripting operations common to all three types. Children of this class - **Abstract Message Scripter.lvclass**, **Concrete Message Scripter.lvclass**, and **Coupled Message.lvclass** – provide the scripting calls specific to each type. (Coupled Message.lvclass is the scripter for standard messages.)

Message Scripter also contains (as attributes) an instance of a **Prototyper.lvclass**, and an instance of a **Send Builder.lvclass.** These classes invoke scripting code that need to be shared by two of the three scripters. Prototypers determine a message’s data payload and the icon of the Send VI. Send Builders create the message’s Send VI. Both of these types of strategy classes are used by the Coupled and Abstract Message scripters.

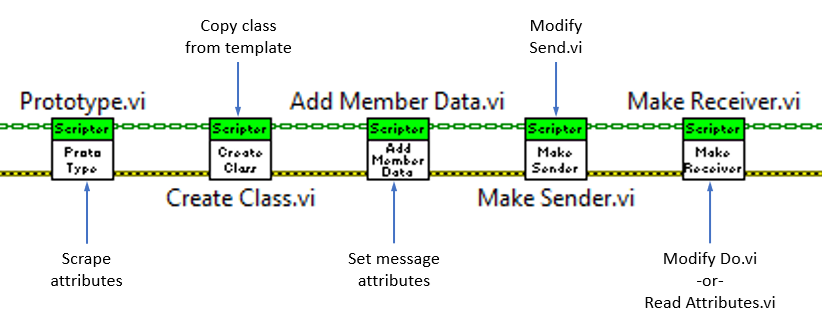
Prototypers inherit from a class that formats the Send VI’s icon. **Target Method Icon Reader.lvclass** derives the icon for coupled messages from the target method’s icon, while **Abstract Method Icon Prototyper.lvclass** creates the standard icon for abstract messages.

The following table summarizes the scripters, prototypers, and builders used to construct each message type.

|  |  |  |  |
| --- | --- | --- | --- |
| Message Type | Scripter | Prototyper | Send Builder |
| Standard (Coupled) | Coupled Message Scripter.lvclass | Target Method Prototyper.lvclass | Concrete Send Renamer.lvclass |
| Abstract  (Reference Method) | Abstract Message Scripter.lvclass | Reference Method Prototyper.lvclass | Concrete Send Renamer.lvclass |
| Abstract  (User-defined) | Abstract Message Scripter.lvclass | User Defined Prototype.lvclass | Concrete Send Renamer.lvclass |
| Concrete Child | Coupled Message Scripter.lvclass | Target Method Icon Reader.lvclass | n/a |

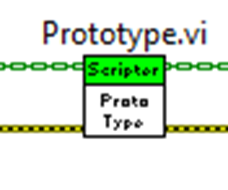
**The Algorithm**

**Message Maker.lvlib:Message Scripter.lvclass:Create Message.vi** is thetemplate method that governs creating a message. The algorithm consists of five dynamic dispatch VIs:



Each of these VIs is described in a following section.

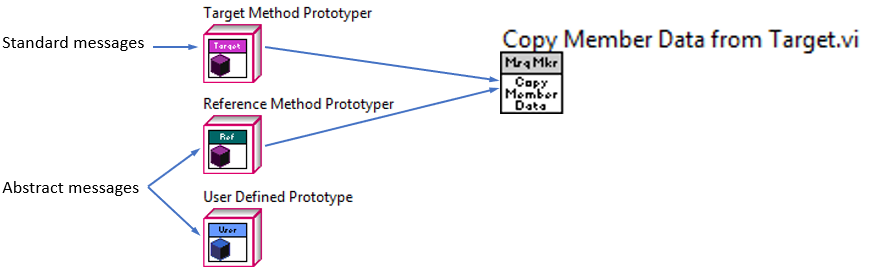
**Prototyping**



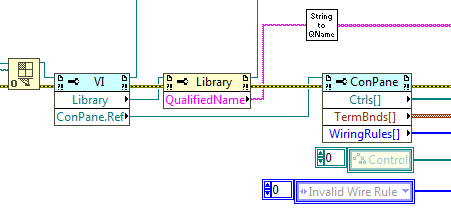
Overridden by Abstract Message Rescripter.

Prototype.vi determines the attributes of the new message class by invoking its prototyper’s Prototype.vi. The Abstract Message Rescripter interrogates the message class being rescripted.

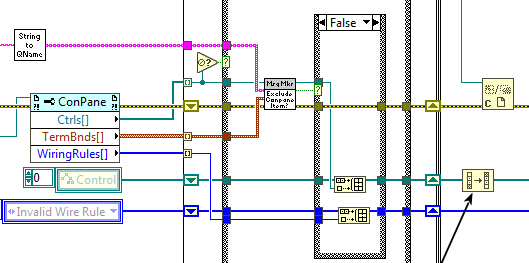
The Target Method and Reference Method versions of Prototype.vi invoke **Message Maker.lvclass:Copy Member Data from Target.vi.**

****

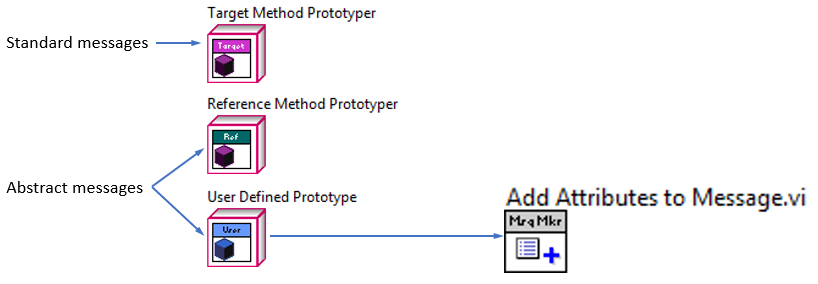
This VI obtains references to all the controls associated with the target method’s connector pane.



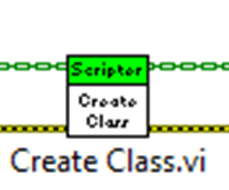
We ignore the target method’s class input, provided it is the upper left connection in the connector pane. We also ignore an error input if it is the lower left connection.



User Defined Prototype.lvclass:Prototyper.vi determines the message’s attributes using **Add Attributes to Message.vi**. This VI is a dialog box.



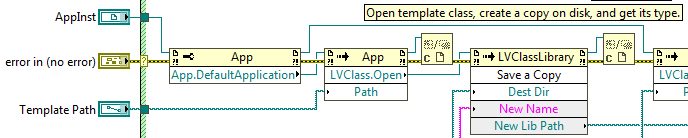
**Class Creation**



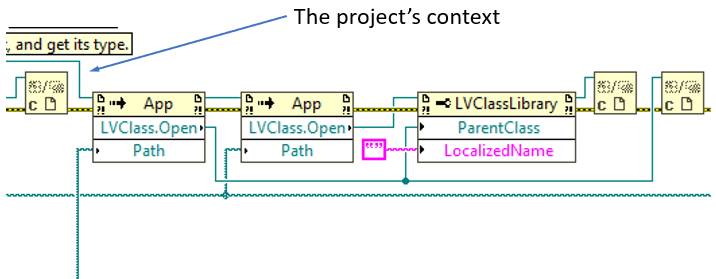
Overridden by Abstract Message Rescripter, Concrete Message Rescripter, and Coupled Message Rescripter.

Create Class.vi ensures that the new message name is unique. It then obtains the path to the appropriate template and the message’s parent class, and then invokes **Message Maker.lvlib:Copy Class.vi.**

Copy Class.vi just copies the class template. The new class is created in the default application, not the provider or project applications.

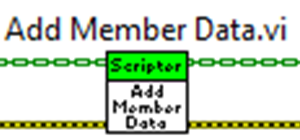


Create Class.vi then opens the parent message class and the new class in the project’s context, and sets the new message’s parent class. (The parent class is either Message.lvclass, or, in the case of abstract messages, a parent specified by the user.)



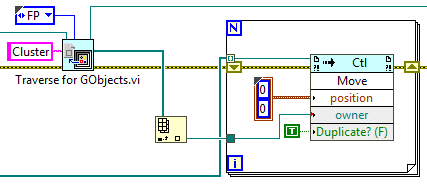
Creating the class in a separate context, and then setting its inheritance helps avoid application cross-linking and loading issues. Specifically, it avoids adding the message template class to the project as an “item in memory”, which would cause issues when creating subsequent messages.

**Adding Member Data**



Overridden by Coupled Message Scripter.

Add Member Data.vi uses the control references obtained by Prototype.vi to populate the attributes cluster of the new class. The actual work is performed by **Message Maker.lvlib:Add Member Data to Private Data Control.vi**. After opening the private data control of the new message class, the VI adds attributes to it as shown here:



**Traverse for GObjects.vi** traverses a front panel or block diagram and returns a reference to each instance of the type that it finds. In this case, we are traversing the message class’s private data control. It only has one control, the attributes cluster. The VI then iterates over the array of control references, and uses the Control.Move method to add a copy of each control to the cluster. Control.Move adds or moves a control within the indicated owner, which can be anything that can hold a control reference (cluster, front panel, etc.).

The Coupled Message Scripter override of this VI invokes **Message Maker.lvlib:Set Message Clas Icon.vi** to set the icon of the message class.

**Creating the Send VI**



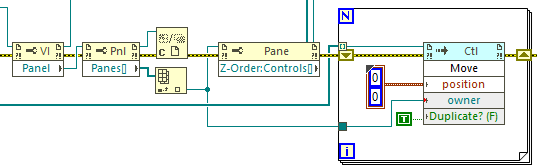
Although it is marked as dynamic dispatch, no scripter class overrides this method; only the base implementation is used.

This VI invokes **Build.vi** from the appropriate Send Builder. For coupled and abstract messages, this is the override provided by **Concrete Send Renamer.vi**. Concrete child messages don’t have Send VIs; in such instances, the base version of Build.vi is invoked, which does nothing.

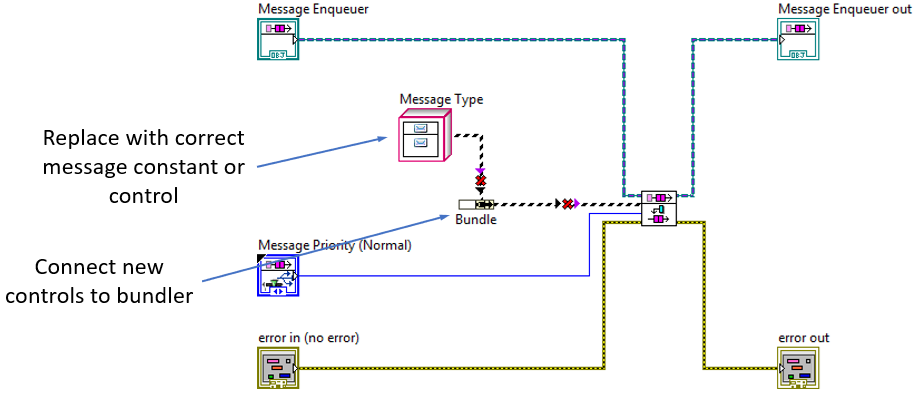
When a Send VI is created, Build.vi first invokes **Message Maker.lvlib:Create Send Method.vi**. This VI copies the Send Template.vi in the template class, under a name of the form Send <target method>.vi. It then deletes the template VI. Build.vi then invokes **Message Maker.lvlib:Build Send.vi**.

If the project includes an Actor Framework PPL, Build Send.vi first uses **Message Maker.lvlib:Replace Enqueue if Using PPL.vi** to replace the Actor Framework Message Enqueuer, Message Enqueuer out, and Message Priority controls/indicators used by the Send VI with their PPL equivalents.

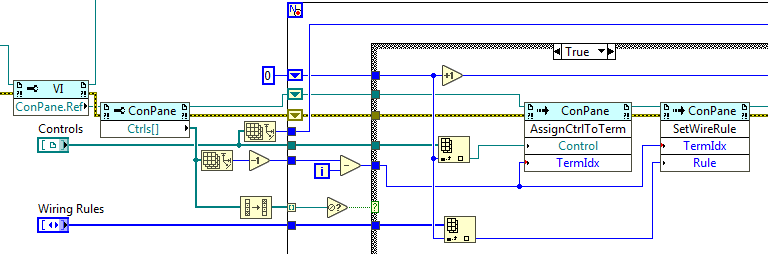
Next, Build Send.vi invokes **Message Maker.lvlib:Add Controls to Method.vi** to add controls to the Send VI’s front panel. This VI obtains a reference to the active front panel pane, and then uses Control.Move to place the controls.



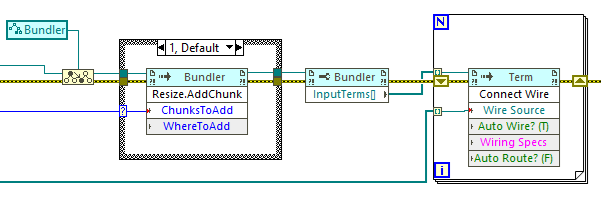
Send Build.vi then replaces the Message Type block diagram constant (for a coupled message) or front panel control (for an abstract message) with an instance of the new message type.



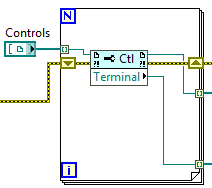
If any controls were added to the front panel, Send Build.vi invokes **Message Maker.lvlib:Controls to Connector Pane.vi** to assign the new controls to connector pane terminals. Here is the relevant code:



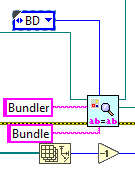
This is followed by a call to **Message Maker.lvlib:Wire FP Controls to Bundler.vi**, which connects the new front panel controls to a bundler that sets the message’s attributes. The bundler must be resized first:



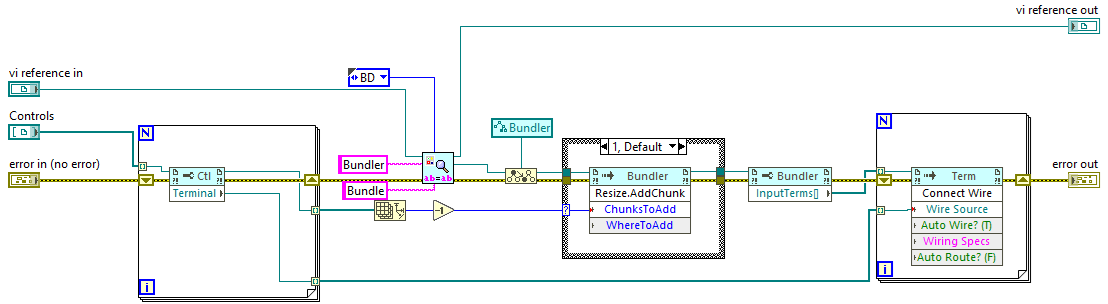
Note that the Wire Source input to Terminal.Connect Wire is NOT a control reference; it is a terminal reference. Terminal references are obtained from control references with a call to Control.Terminal:



We find the bundler with a call to **TREF Find Object By Label.vi**:



Here is the complete VI:



If the message has no data, the bundler is deleted and the message class input directly wired to Enqueue.vi, via a call to **Message Maker.lvlib:Wire Class to Enqueuer.vi**.

Send Build.vi wraps up by invoking VI.Block Diagram:Cleanup and **Message Maker.lvlib:Clean Up Panel.vi**, a rather complex VI that places the Send VI’s front panel controls in a manner similar to their arrangement in the connector pane.

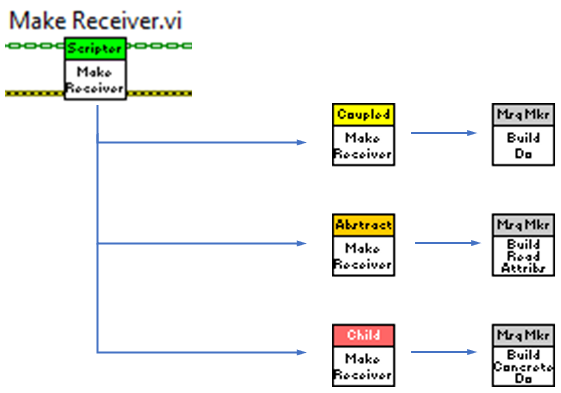
Once Send Build.vi completes, Build.vi invokes **Message Maker.lvlib:Send Icon.vi** to set the new VI’s icon.

**Creating the Receiver VI**



Overridden by Abstract Message Scripter, Coupled Message Scripter, Concrete Message Scripter, Concrete Message Rescripter, and Coupled Message Rescripter.

Make Receiver.vi performs many of the same operations as Make Sender.vi, just on the Do.vi or Read Attributes.vi of the new message class. Each message type has a different target for these operations, so each invokes a different build VI in Message Maker.lvlib:



The three VIs are **Build Do.vi** (for coupled messages), **Build Concrete Do.vi** (for concrete child messages), and **Build read Attributes.vi** (for abstract messages). All are members of Message Maker.lvlib.

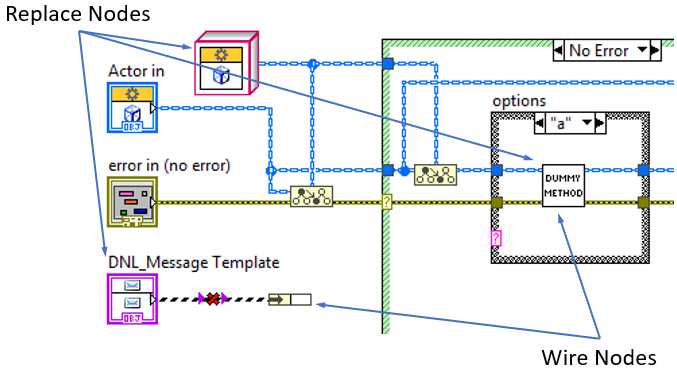
Abstract messages don’t include a Do VI. Instead, they provide a data accessor. Concrete children of that abstract message use the accessor in their implementations of Do.vi.

Build Do.vi

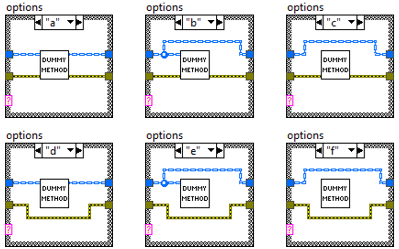
As with Build Send.vi, this VI modifies a method copied from the message class template (Do.vi, in this case).

Build Do.vi first finds and replaces the Actor.lvclass constant on the block diagram with an instance of the target actor. Then, if the application uses an Actor Framework PPL, it swaps the Actor.lvclass control and indicator for their PPL counterparts. Then the VI replaces a dummy VI (Dummy Actor Method.vi) with the method to be invoked by the message, and wires connector pane items of that method to the unbundler of the message’s data.

This diagram illustrates the GObjects that are replaced, and the nodes that are wired.

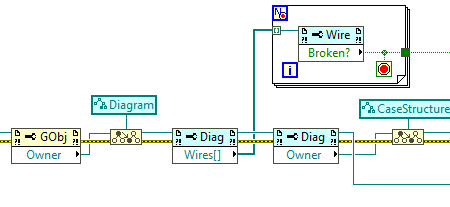


Actor methods are expected to provide some combination of controls and indicators for the actor itself and for error handling, but any given method can supply all, some, or even none of these controls/indicators. To manage this complexity, the template Do method wraps the dummy method in a case structure, with a case for each supported combination, as shown here:

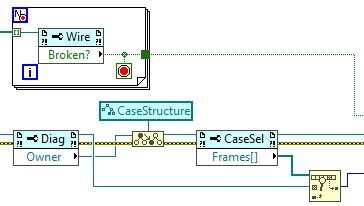


Build Do.vi uses the scripting VI **Special Replace of SubVI Node.vi** to find the first instance of the dummy method, and replace it with the target method. If the target method doesn’t have the correct inputs and outputs for that case, one or more wires will be broken, indicating that the wiring arrangement doesn’t suit the target method. **Rewire Do.vi** detects those broken wires. If it finds any, it deletes that case from the case structure. Otherwise, it deletes the entire case structure, leaving just the desired arrangement.

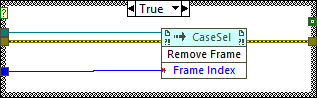
Broken wire detection is shown here. Note that the target VI’s owner is the diagram from one frame of the case structure.



We use the reference of the owning diagram to identify the frame of the case structure under inspection, as shown here.

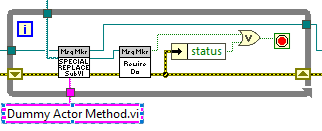


If the frame contains any broken wires, we can simply delete it.



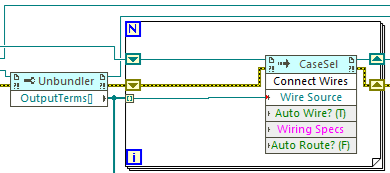
If there are no broken wires, we have found the desired wiring arrangement, so we select that frame to be visible, and then remove the case structure. This deletes any other frames that may remain.

This while loop on the block diagram of Build Do.vi manages the search. You can see the loop terminates as soon as Rewire Do.vi finds a good match.



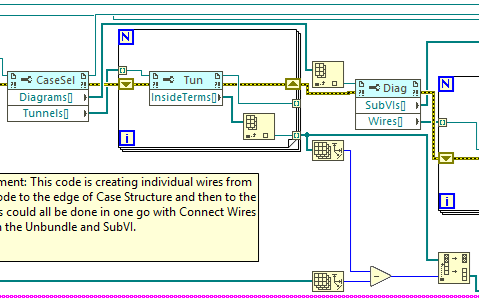
Note that this process makes the order of the frames of the case structure important. The first frame will break if the target method lacks any of the expected connections. The last frame requires no connections, and thus will never break. The middle frames require some, but not all connections.

Once the dummy method has been replaced with the target method, **Wire FP Controls to UnBundler.vi** connects the target methods controls to the unbundler for the message’s data. This VI first finds the unbundler and the error/no error case structure that wraps the target method. It then connects each output terminal of the unbundler to the case structure, as shown here.

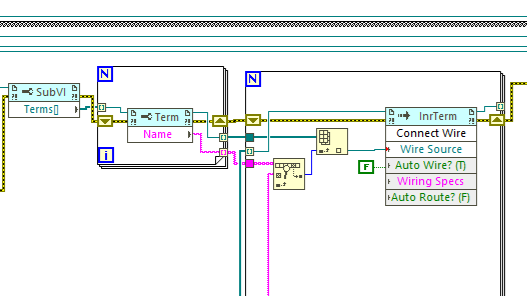


It then identifies the inside terminals of the first case (the no error case, and discarding the preexisting actor/error cluster terminals), and the unconnected terminals of the target method (the only subVI in the case). It then connects the two by terminal name.

Here is how the inside terminals are identified:



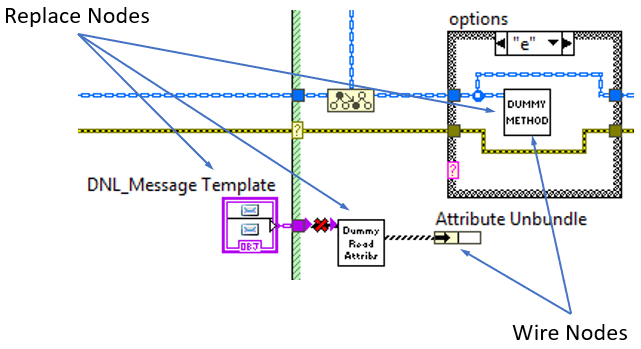
The terminals are connected as shown here.



The balance of Build Do.vi changes the name of the message control, sets some VI options, performs a diagram cleanup, and saves the VI.

Build Concrete Do.vi

This VI builds the Do VI of a concrete child message. It is almost identical to Build Do.vi, with the notable exception that it must find and replace a second dummy VI – Dummy Read Attributes.vi. This VI is replaced with the Read Attributes.vi of the child methods parent class (see Build Read Attributes.vi, below).

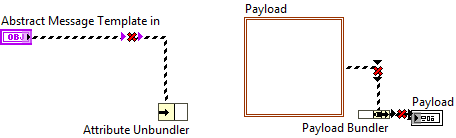


Build Concrete Do.vi finds and replaces Dummy Actor Method.vi using **Find and Replace SubVI.vi** instead of Special Replace of SubVI Node.vi. The reason for the difference is not clear. Find and Replace SubVI.vi is also used to replace Dummy Read Attributes.vi. If the parent abstract message contains no data, Dummy Read Attributes.vi is simply deleted.

Build Read Attributes.vi

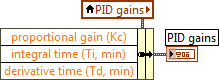
Abstract messages must provide a read accessor for message data; this accessor is called in the Do methods of every child of the message. **Build Read Attributes.vi** builds this accessor.

As with the other builders, Build Read Attribute.vi modifies a template VI, Read Attributes.vi. Here is the initial block diagram of that VI:



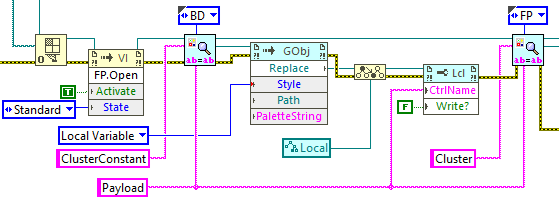
The Build Send VIs expect message class data in a cluster, so Read Attributes returns a payload cluster. The scripting tools need a way to name the terminals of the payload bundler. This function is normally served by a cluster constant, but populating both that constant and the Payload indicator is tedious.

The solution used here is to replace the cluster constant with a local variable linked to Payload. The result might look like this:



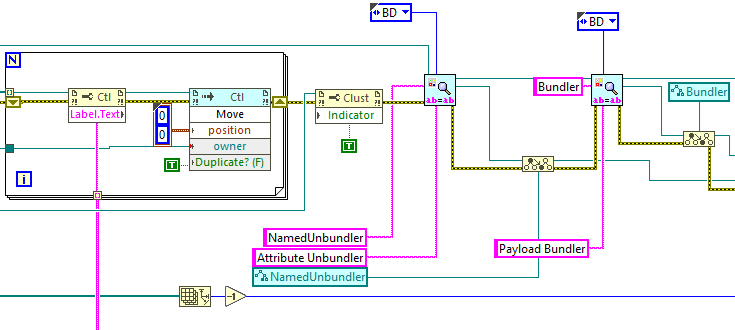
At the time Build Read Attributes.vi was written, there was an issue that prevented simply placing a local variable on the block diagram of the template and then writing the appropriate VI Server code, so an empty cluster constant was used instead, as shown above.

Build Read Attributes.vi first finds and replaces the cluster constant with a local variable, and then links the variable to Payload, as shown here:

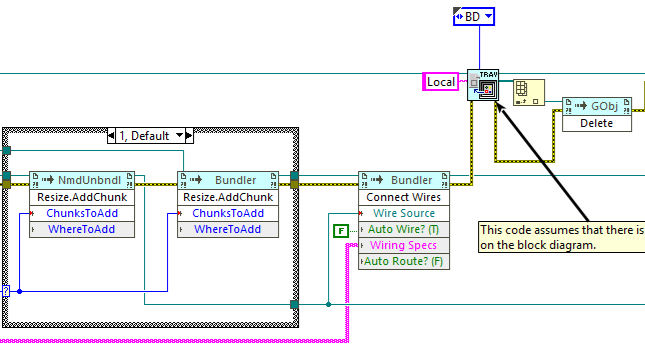


It then invokes **Wire Read Attributes.vi** to add controls to Payload, resize the Attribute unbundler and Payload bundler, and connect the unbundler terminals to their bundler counterparts.

Wire Read Attributes.vi adds controls to Payload and obtains bundler unbundler references thus:



The text of the control labels is used later to connect the bundler/unbundler terminals. Here, you can see the VI resizing the bundler and unbundler and making the connections.



Once wiring is complete, we no longer need the local variable, so we delete it. (This is one reason why we don’t simply make Payload a typedef, the other being that we only use Payload once.)

When Wire Read Attribute.vi returns, Build Read Attributes.vi renames the message class control, cleans up the block diagram, and saves the VI.

**Rescripting Messages**

**Reference Links**

**On Project Providers**

The LabVIEW Wiki entry is here:

<https://labviewwiki.org/wiki/Project_Provider_Framework>

The LabVIEW Project Providers forum:

<https://forums.ni.com/t5/LabVIEW-Project-Providers/bd-p/bymqyodmkc>

Written documentation and examples:

<https://forums.ni.com/t5/LabVIEW-Project-Providers/Project-Providers-Documentation/td-p/3492573>

A presentation by David Ladolcetta:

<https://www.youtube.com/watch?v=xXGro_DylHs>

The slide deck for that presentation:

<https://forums.ni.com/t5/Developer-Center-Resources/Customize-the-LabVIEW-Project-Explorer-Using-the-Project/ta-p/3532774>

**VI Scripting**

The LabVIEW Wiki entry has a number of great links:

<https://labviewwiki.org/wiki/VI_Scripting>

Trevor Christman’s introduction to VI Scripting:

<https://forums.ni.com/t5/Past-NIWeek-Sessions/Introduction-to-VI-Scripting-in-NI-LabVIEW/ta-p/3496554>

Darren Nattinger on VI Scripting:

<https://forums.ni.com/t5/Community-Documents/Don-t-Wait-for-LabVIEW-R-amp-D-Implement-Your-Own-LabVIEW/ta-p/3794651>