**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.

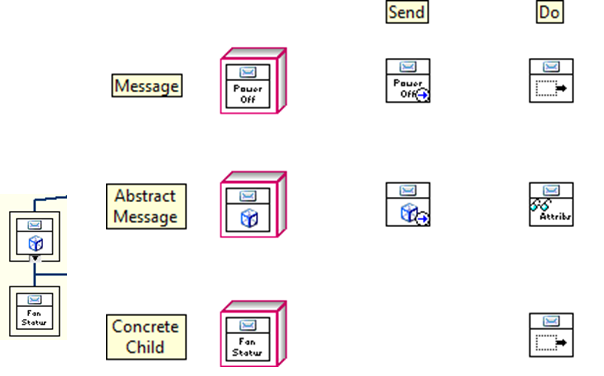
The providers will also create abstract messages and their child concrete implementations. (See the note about abstract messages immediately following this section.)

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

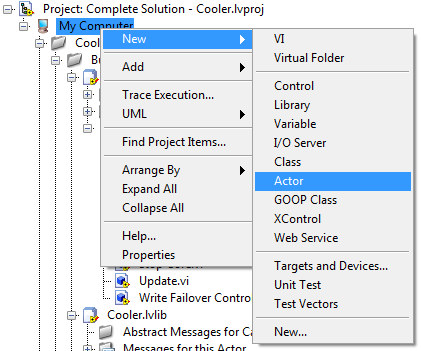
****

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**

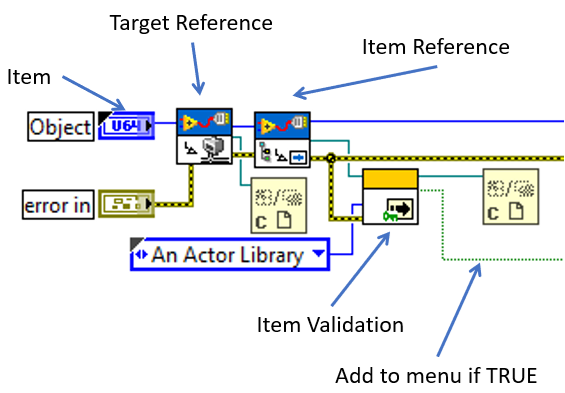
Message Maker Provider, Actor Message Maker Provider, and Message Rescripter are the suite’s secondary providers. They are responsible for creating and updating messages for actors and actor interfaces.

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 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.

Executing Scripting Code

**Error Handling**

**Creating Actors and Interfaces for Actors**

**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).

**Prototyping**

**Class Creation**

**Adding Member Data**

**Creating the Send VI**

**Creating the Receiver VI**

**Rescripting Messages**

**PPL Support**

**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>