UI Tools

Current version: 1.3.0
Author: François Normandin
LabVIEW Version: >=8.6
License Type: BSD

Distributed on the LabVIEW Tools Network.

Online support thread: http://lavag.org/topic/11045-cr-ui-tools

Description:

This package contains tools for designing user interfaces.

A first palette helps create special effects using transparency of front panel. Using them allows to quickly create fade-ins or outs using linear or exponential variation of the intensity.

A second palette contains VIs to calculate the position of GObjects for many purposes like alignment, snap, mouse follow, etc.

A third palette contains VIs to create dialog boxes based on class instances. A One-button base class and a two-button class are provided to be overloaded.

A fourth palette includes some VIs to move objects programmatically on the front panel, using a basic deceleration algorithm to provide an impression of a smooth movement.

The packaged VIs are all namespaced using a suffix "__lava_lib_ui_tools" which should not conflict with any of your own code.

Includes:

- Front Panel Transparency (Fade In & Fade Out)
- Utilities (Alignment, Snap)
- Dialog (OOP based, extensible)
- Engine (Beta) for object movement

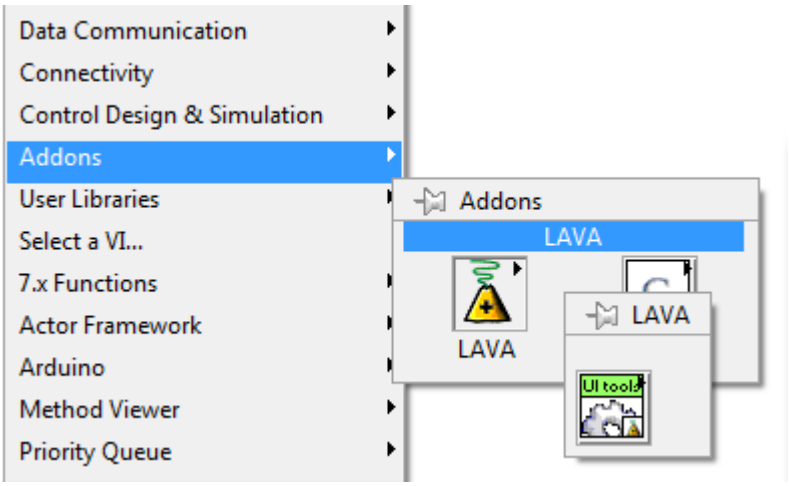
Installation instructions:

This package is distributed on the LabVIEW Tools Network and can be installed directly in the addon folder of any LabVIEW version from 8.6 to now. The addon installs automatically under the LAVA palette of the addon submenu.

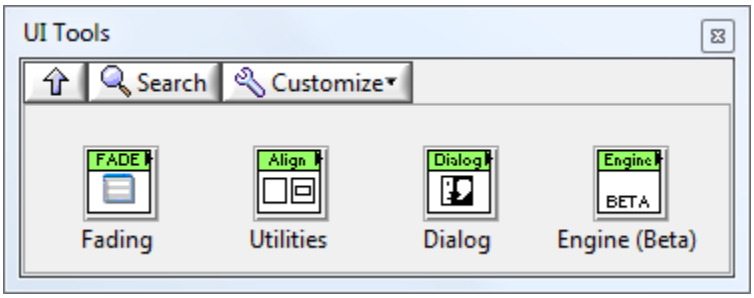
Code Example Index

How to use UI Tools.

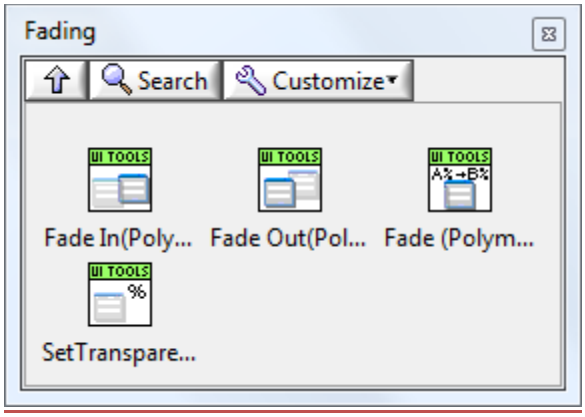
The UI Tools Palette is installed under “Addon > LAVA” palette.



Four palettes compose the UI Tools addon.



FADING PALETTE:

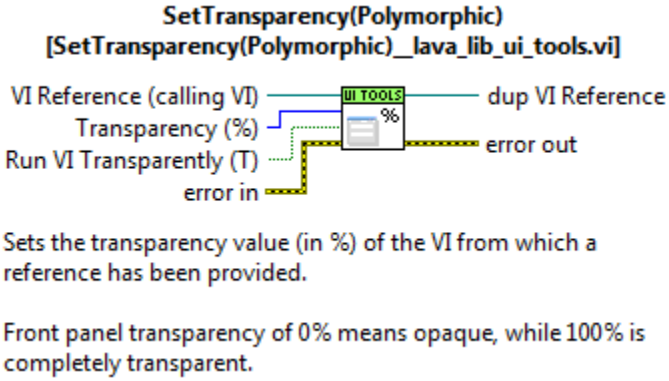


*Set Transparency.vi

Fade In (Polymorphic).vi
Fade Out (Polymorphic).vi
Fade (Polymorphic).vi

**for clarity, the namespace "__lava_lib_ui_tools" suffix has been removed from the VI names in this document.*

Use "*Set Transparency.vi*" to directly set the transparency of a front panel. This function simply replaces the Invoke Node used on a Front Panel reference. If you drop this VI on a block diagram without wiring a front panel reference, the parent VI 's front panel will be called via the Call Chain. The transparency input can either be a U8 (0-100%), a string or a variant, so that you can use a numeric or a string, which can be useful if you use a text argument such as the JKI state machine.



Use “*Fade In (Polymorphic).vi*” or “*FadeOut (Polymorphic).vi*” to go to 0% or 100% transparency respectively. If you wish to set the fading effect to stop to any other transparency level, use “*Fade (Polymorphic).vi*” and specify the target transparency. These VIs will adapt to I32 input, string input or any other numeric (variant input).

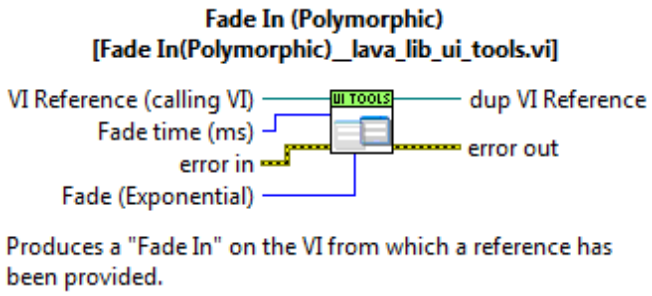
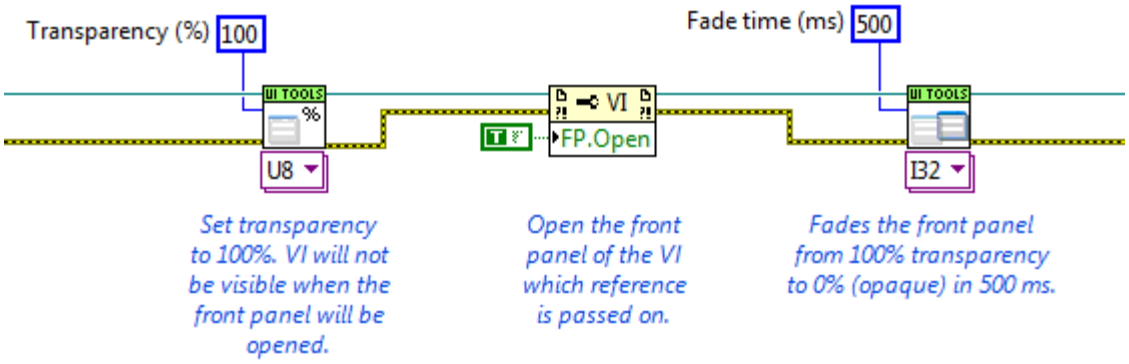


Figure 1.0 Example of good code for fade-in effect on a VI.



IMPORTANT NOTE: If the VI Reference input is not wired in, each primitive gets a reference to the calling VI. Upon exiting the node, the VI reference will be automatically closed if no valid reference was passed. Therefore, it is IMPERATIVE to wire a valid reference if the user wants to wire the nodes in sequence. See figure 1.1 for demonstration of this behavior.

Figure 1.1 Example of code that will not behave as expected.
(Effect of leaving the VI reference input terminal unwired)

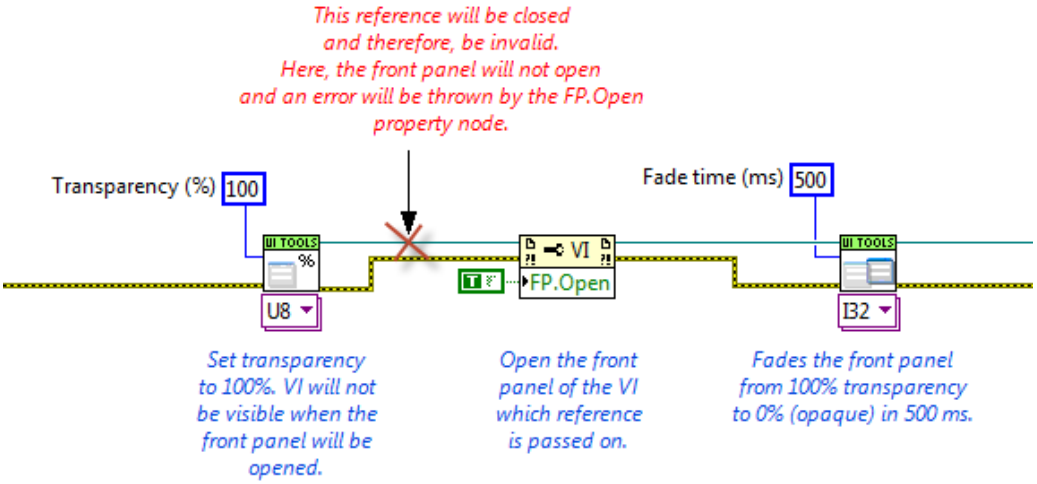
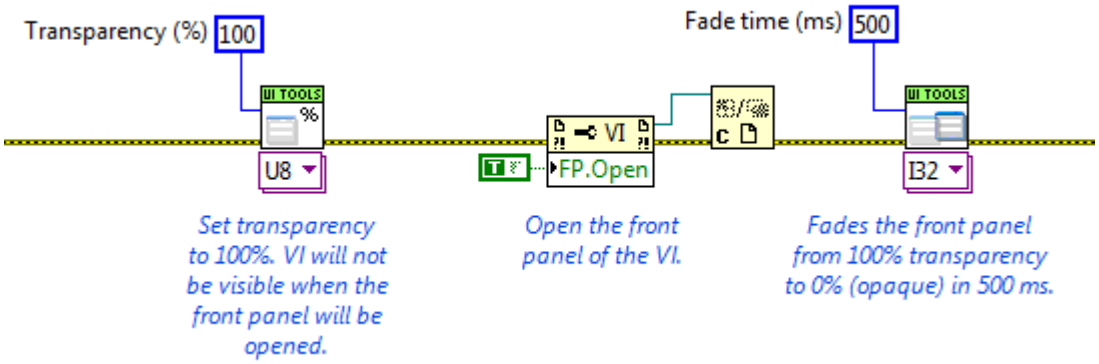


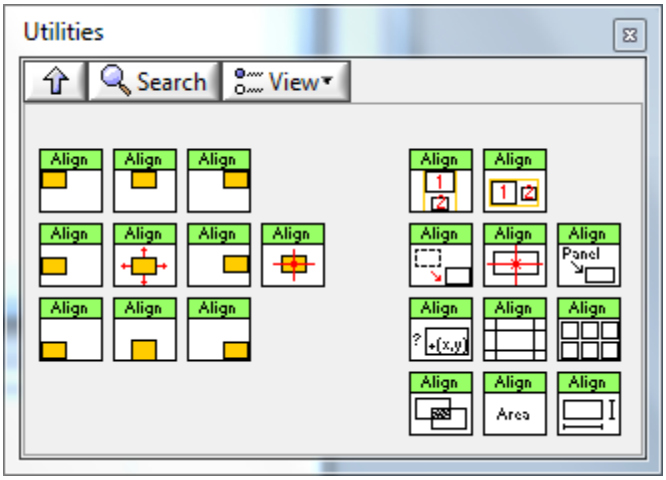
Figure 1.2 Example code for fade-in effect of the current VI.
(Each node will open a reference to the calling VI and close it afterwards)



Note that the code shown in figure 1.2 will make the front panel flicker if the front panel is not closed upon execution of the code. This code assumes that the VI runs with its front panel closed.

Search the LabVIEW examples using *Fade* keyword for quickly finding an example VI for this palette.

ALIGNMENT UTILITIES:

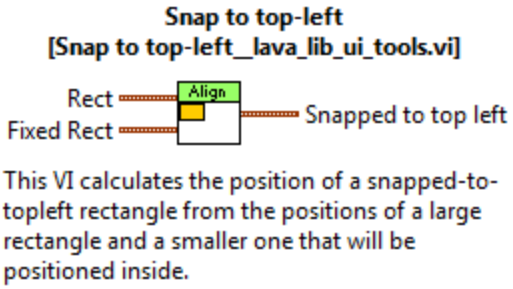


Snap to top-left	Snap to top	Snap to top-right	
Snap to left	Snap to Center	Snap to right	Snap to Point
Snap to bottom-left	Snap to bottom	Snap to bottom-right	

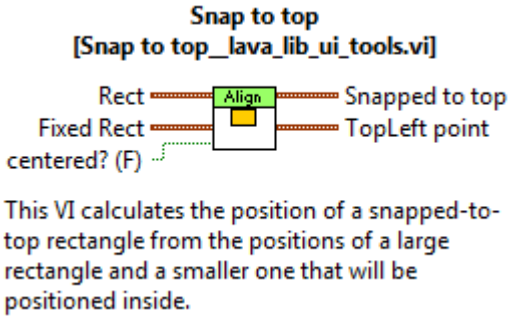
Distribute Rectangles Vertically
Distribute Rectangles Horizontally
Calculate GObject Rectangle
Calculate Center
Windows Bounds To Rectangle
Is Point In Rectangle?
Divide Rectangle
Divide in Squares
Calculate Intersection
Calculate Area
Rectangle Size

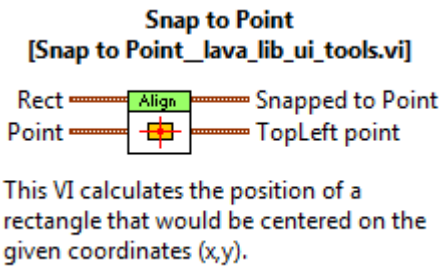
“Snap to...” VIs are utilities to calculate the position of a rectangle with relative to a fixed rectangle in order to provide positioning. There are two types of VIs for this palette subset: Corners and Cardinal points.

The four VIs representing the corners (top-left, top-right, bottom-left, bottom-right) return the coordinates for the rectangle to snap to a fixed rectangle. They simply require two rectangles as input and return another rectangle as output.



The other VIs of this subpalette are representing the cardinal points and the center (top, left, bottom, right, center, point). These VIs give the user the option to center or not the rectangle in the middle of the destination border. They also add the functionality of calculating the top-left point of the resultant position, which is a useful shortcut.





Figures 1.3a to 1.3c feature an example of using the snap to API to calculate a target position for a GObject. The nodes (on block diagram) used to extract the objects' rectangles will be explained later. The user will note the difference in final object position when using a centered or uncentered snapping.

Figure 1.3a Aligning a GObject on the front panel
(Initial position of the GObject to be moved & block diagram)

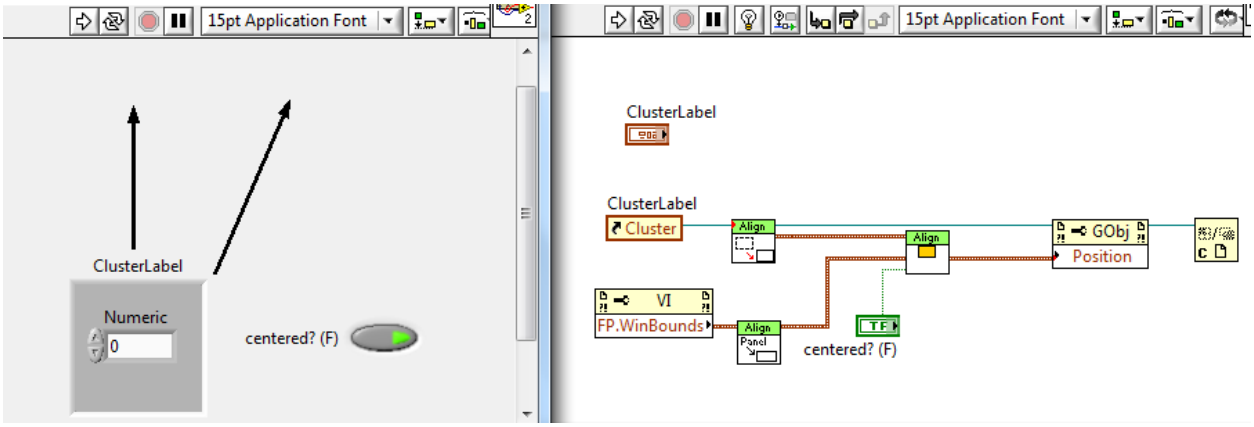
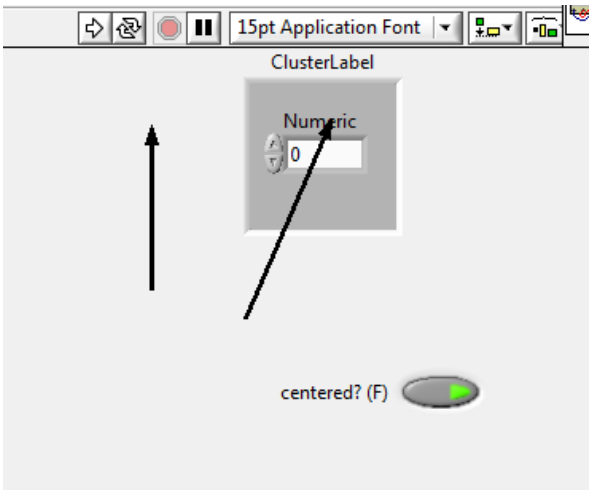
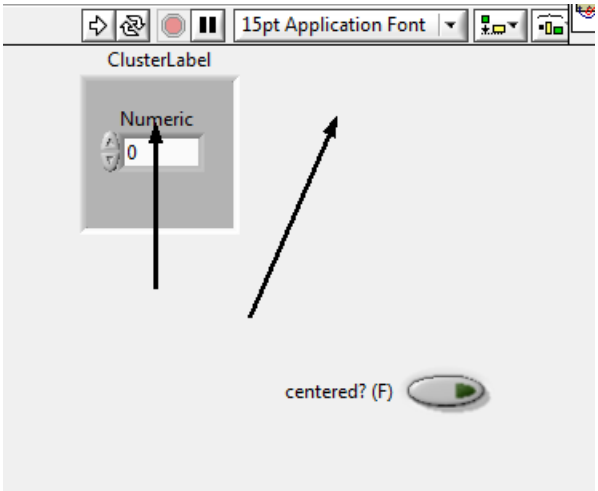


Figure 1.3b-c Aligning the GObject centered (b) and uncentered (c) on FP.
(Results of running the VI)

b)



c)

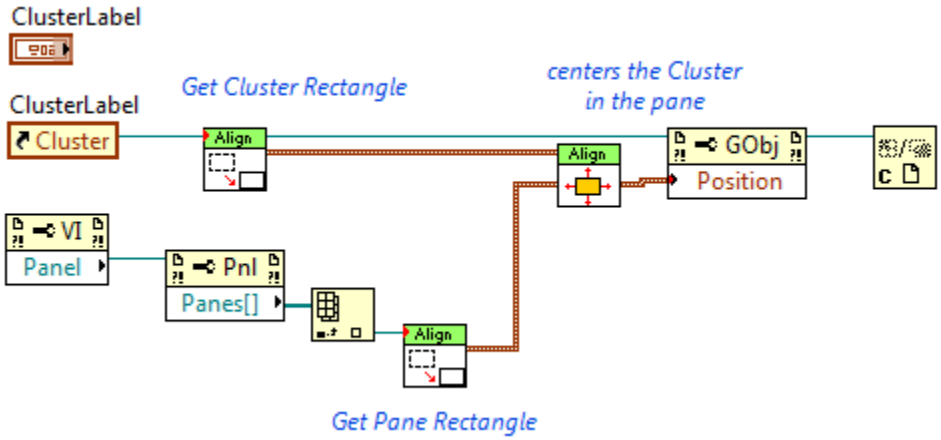


USEFUL TIP: Use the Snap to Point VI to have a GObject snap to coordinates, whether from a mouse click or some predefined coordinates.

“Calculate GObject Rectangle” is a VI that accepts *any GObject reference* and returns the rectangle associated with its position and height. The output rectangle will include the label or captions. This utility is mainly used to quickly feed the snap VIs with the proper formatted rectangle cluster. This includes decorations and containers such as a cluster, a tab or a subpanel.

IMPORTANT NOTE: Although the GObject definition includes the “Panes”, it is not recommended to use use a pane reference with this primitive. The user will find that the code works well even with panes, that's not the problem, however a pane dimension changes as the GObjects are moved around. It results that running the same code multiple times will need yield the same result if the pane's dimensions are changed as a result of moving the GObject around. To convince yourself, run the code in figure 1.4 multiple times and you will see the cluster move out of view as the pane expands towards the bottom-right corner... Be also advised that the same behavior can be problematic with any automatically resizable container such as moving objects within a tab or a cluster.

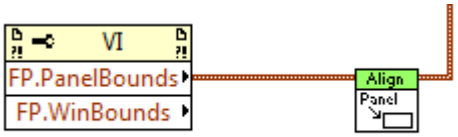
Figure 1.4 Moving a GObject in a pane



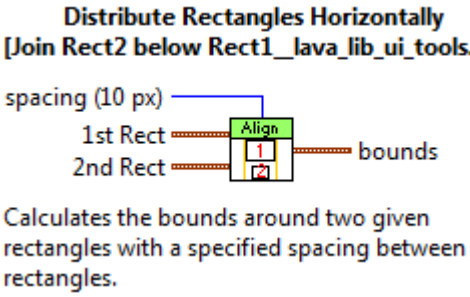
(Run multiple times to see why results can vary depending on initial conditions)

“Windows Bounds To Rectangle” is essentially the same functionality than the “ Calculate GObject Rectangle” except it is meant to be used with VI refnums. This VI will accept either the “Panel Bounds” or the “Windows Bounds”. Note that using the panel bounds is better as it excludes the scroll bars, menu bar, title, etc.

Figure 1.5 Extracting the rectangle cluster from a VI reference

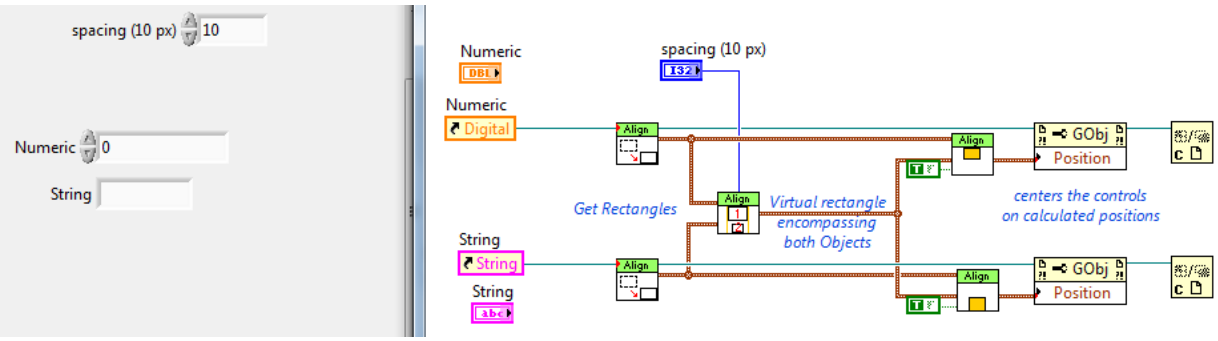


“Distribute Rectangles Vertically/Horizontally” are utilities that will calculate the rectangle that would best encompass two given rectangles, taking the first rectangle as the fixed

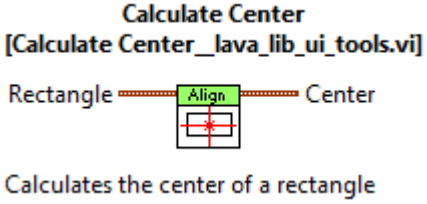


point reference for the top-left position.
The resulting rectangle can then be used to compute the best location of the two objects as shown in figure 1.6. Here, the first object (Digital class) is snapped to top and the second object (String class) is snapped to bottom. If using the horizontal alignment node, use the “snap to left or right” nodes instead.

Figure 1.6 Moving two GObjects to align vertically with a desired spacing.



“Calculate Center” is used to calculate the center of a given rectangle. Used in conjunction with the “Calculate GObject Rectangle” or “Windows Bounds to Rectangle” utilities, it allows to quickly get the center of those objects. Use in tandem with the “Snap to point” node and you have a versatile utility for any of your applications.



“Is point in Rectangle” provides validity checking for coordinates or points.

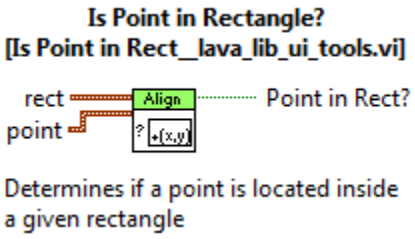
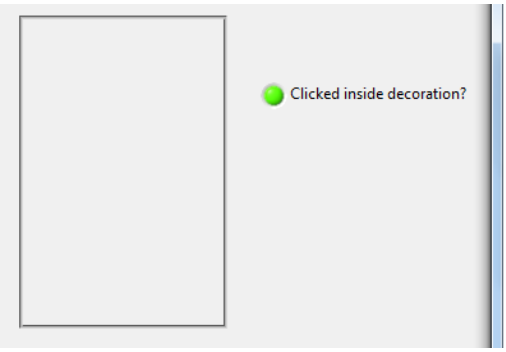
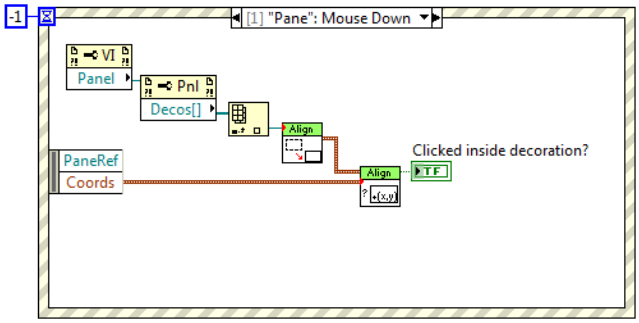


Figure 1.7 Example of area mapping
(similar to area HTML tagging)





USEFUL TIP: The “Is point in Rectangle” can be used in tandem with “Divide Rectangle” or “Divide in Squares” to detect if the mouse has entered a certain predefined area. These primitives can be used to validate where an object is being dragged or dropped. They can also check if the mouse cursor is within a certain distance of the borders of a container, for example a dynamic toolbar or perhaps an icon palette...

"*Divide Rectangle.vi*" is a utility that calculates regions on the edge of a rectangle. This is useful to calculate if an object or mouse is inside a certain region near a corner or edge center. For example, it provides the logic to decide if an object should snap to the edge/corner or not. "Band" is the width of the side zones. It should be smaller than half the width or height to perform correctly.

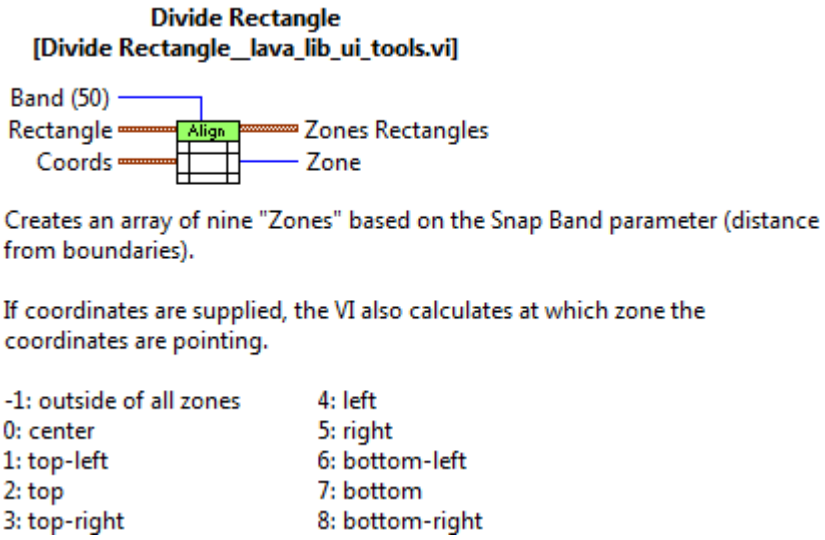
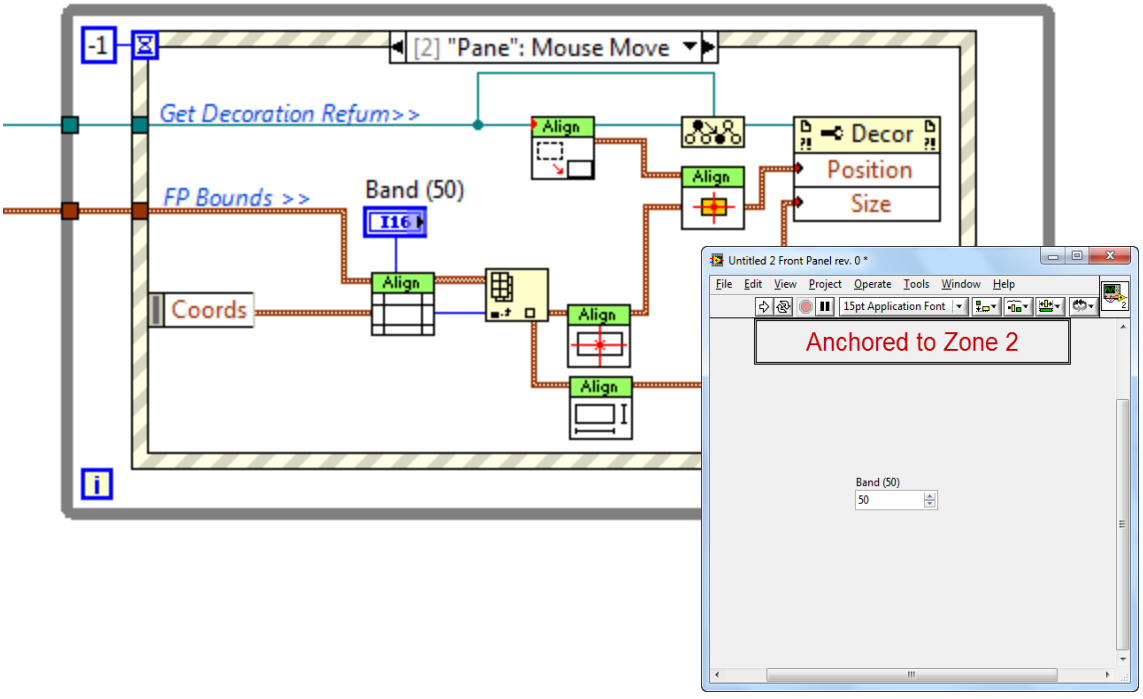
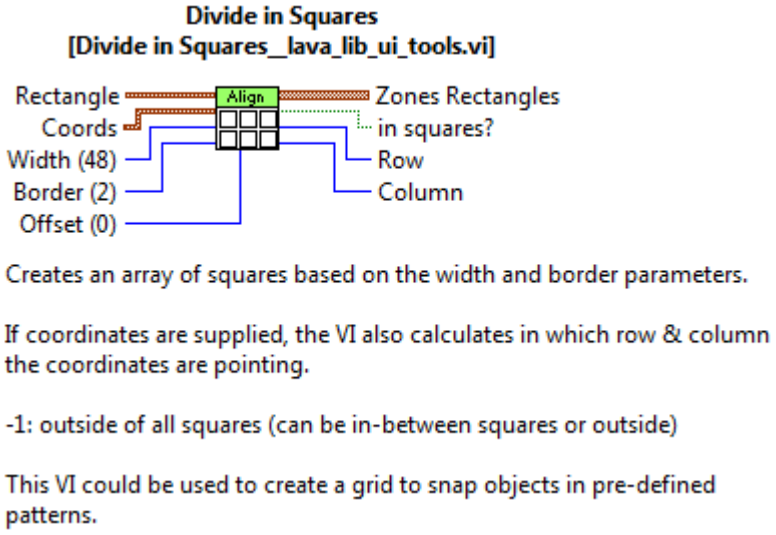


Figure 1.8 Example code for highlighting a certain area on the front panel
(A decoration is fitted to highlight the zone detected)

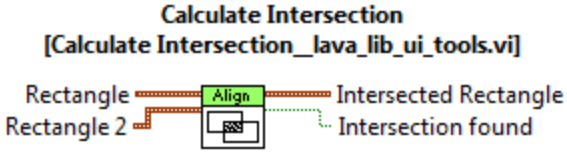


"*Divide in Squares.vi*" is a utility to divide a large rectangle in multiple square areas. It provides the logic to determine the boundaries of an object if one wishes to make palette or something equivalent.

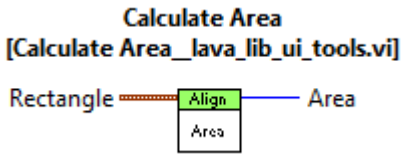


USEFUL TIP: Dividing an area in multiple squares can be useful if one wants to define areas where action is possible. For example, a palette or toolbar can be made with objects (icons, controls) that will snap within a defined rectangle. Drag it from one square to the other to avoid any object from snapping where it is not supposed to.

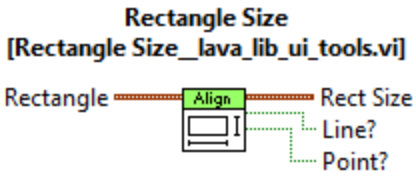
"Calculate Intersection" is a simple utility that calculates the intersection between two rectangles. You can find any object overlapping or test whether an object is "snappable" to a certain zone.



“Calculate Area” is a simple tool to get the area in pixels square (height x width). This can be used to determine the relative weight of two rectangles if, for example, more than one overlapping zone if detected.



“Calculate Rectangle Size” consists of a utility to extract the dimensions of a rectangle. Useful for resizing an object to fit a rectangle. Examples range from decorations to subpanels.



If the rectangle has no dimensions (0px by 0 px), then the “Point?” output will be True. If the rectangle has either width or height that is 0 px, then the “Line?” output will be True.

USEFUL TIP: The “Line?” and “Point?” are provided to be used with the “Calculate intersection” VI where the result of this node could be a line or a point if the studied rectangles are adjacent but not overlapping...

Legacy VIs (backward-compatibility with previous versions) have been kept in a subfolder but are no longer shown in the palette. They consist of “centered snaps”, which is now an option on the “snap” VIs (wrapper functions).

Search the LabVIEW examples using *Snap* keyword for quickly finding an example VI for this palette.

DIALOG PALETTE :

This palette currently consists of 4 classes, with the One-button dialog class being the base class for the others. Extensions are planned for this set of classes. It is meant as a series of classes that will extend the LabVIEW UI to behave as a set of jQuery classes (for those familiar with the javascript environment for webpages)

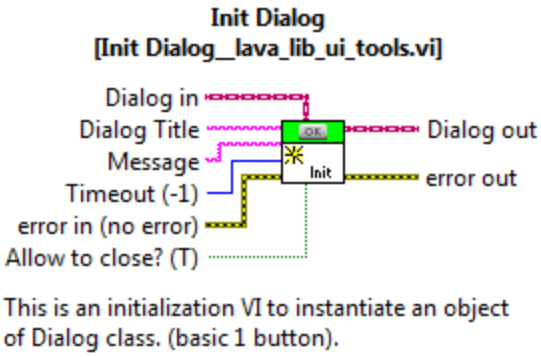
IMPORTANT NOTE FOR USERS NOT FAMILIAR WITH OBJECT-ORIENTED PROGRAMMING IN LABVIEW:

DO NOT BE AFRAID! The API is very easy to use and you do not need to know any of the specifics of LVOOP to enjoy it. Experienced users will enjoy the benefits of inheritance and exploit more features, but rest assured that you can still use this set of VIs just as you would with your usual LabVIEW primitives.

The following pages will take you on a tour to how to use them.

The base class (**One-button dialog**) is shown on the top row, with the green header.

“Init Dialog” is the first step for configuring a new dialog. It is a LVOOP constructor and is the basis for all other classes in this series of dialog objects. We will use another node to actually display the message. This is only an initialization node.



You do not need to wire the “Dialog in” terminal. This terminal is provided for inheritance and will be wired only for dynamic dispatching (see figure 2.4).

Initialize the message string with the message you wish to display to the user.

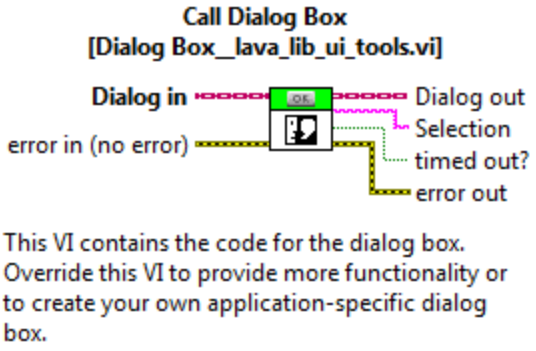
Fill in a dialog box title to display it in the title bar. If you leave this terminal unwired or provide an empty string, the title bar will be hidden if you do not allow the user to close the dialog box.

The timeout is in milliseconds and is set by default to -1 (no timeout).

If you allow the user to close the dialog box, the title bar will be shown even if the title is empty. Furthermore, to prevent the configuration of a dialog box that cannot be closed, the “Allow to close” is set to True whenever the timeout is unlimited (-1). That is, you cannot be stuck with a modal dialog box that you can never close. Beware however that setting a timeout with a large time could provide a virtual deadlock situation.

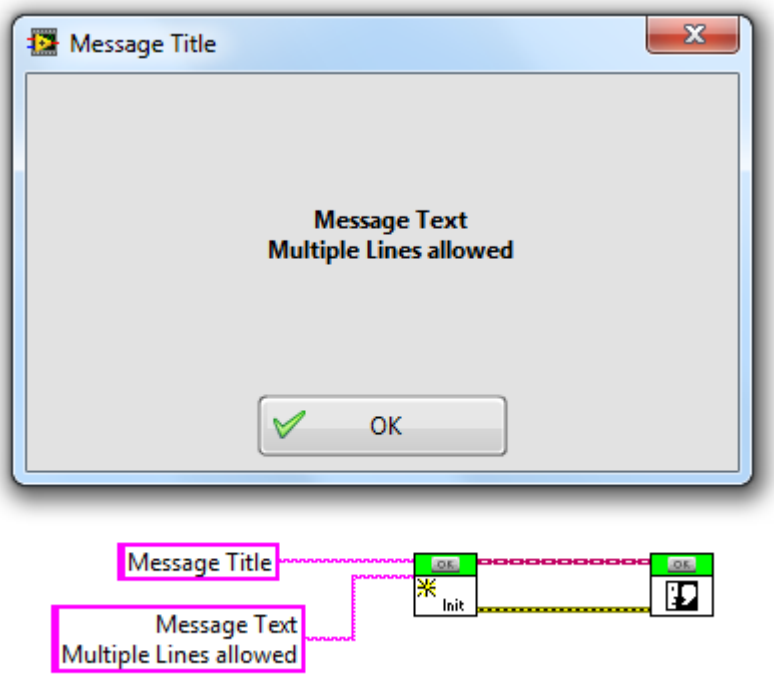
USEFUL TIP: You can create an array of pre-configured dialogs and call any of them later with the “Call Dialog Box” VI, using the Index Array primitive.

“Call Dialog Box” is the core of the execution of this set of classes.



When you execute this node, the aforementioned dialog box will be displayed. The “selection” output terminal returns the boolean text of the pressed button. If the dialog box times out or the panel is closed by the user, the selection returns an empty string.

Figure 2.1 Example code a simple Dialog Box with an OK button.



“Call Dialog Box (with Blackening effect)” is a wrapper function that will create a blackening of the screen before displaying the dialog box. When the dialog is closed, the screen fades back to normal. You can adjust the desired transparency programmatically from 0 to 100%, where 100% is completely transparent background (therefore, the same as using the “Call Dialog Box” node.) and 0% is a completely opaque background.

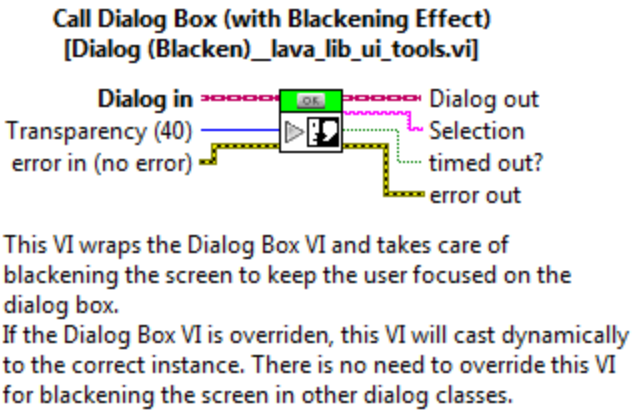
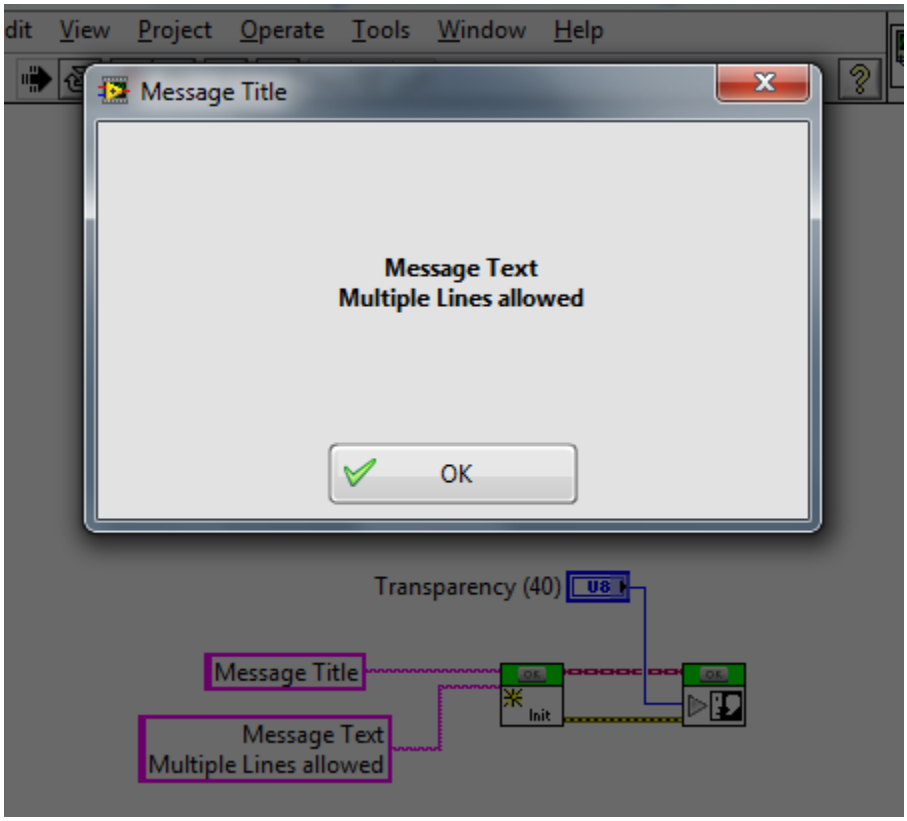
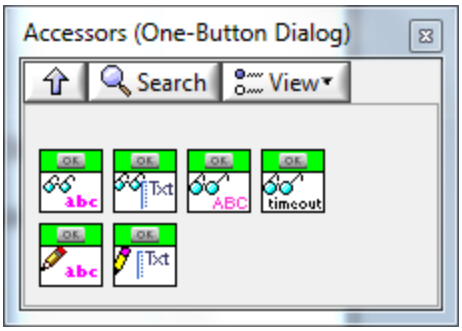


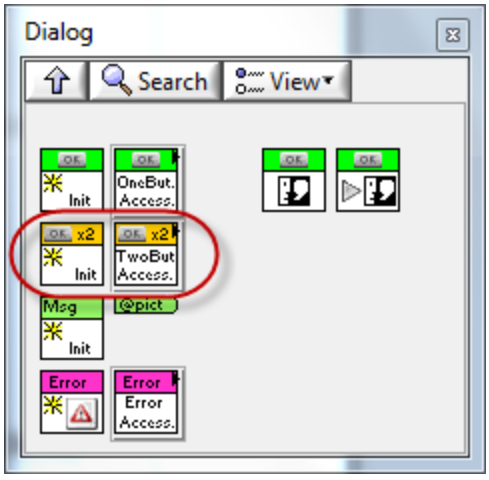
Figure 2.2 Example code a simple Dialog Box with an OK button with a blackened background.



A subpalette with accessor VIs is provided to set or get parameters after initialization of the object has been done. You can change the message or title without re-initializing a new instance of the class. These accessors are also useful for child classes.

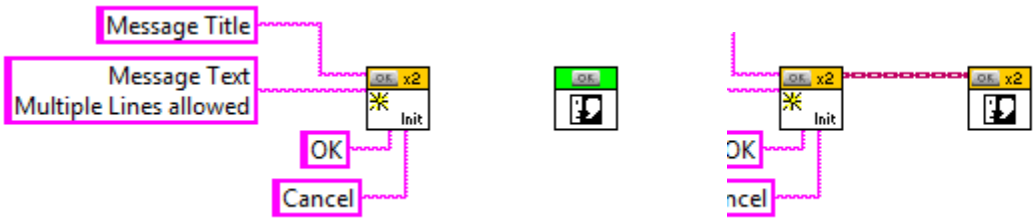


The base class is being extended with the obvious **Two-Button dialog** which is shown in the second row of the dialog palette, with the yellowish headers.



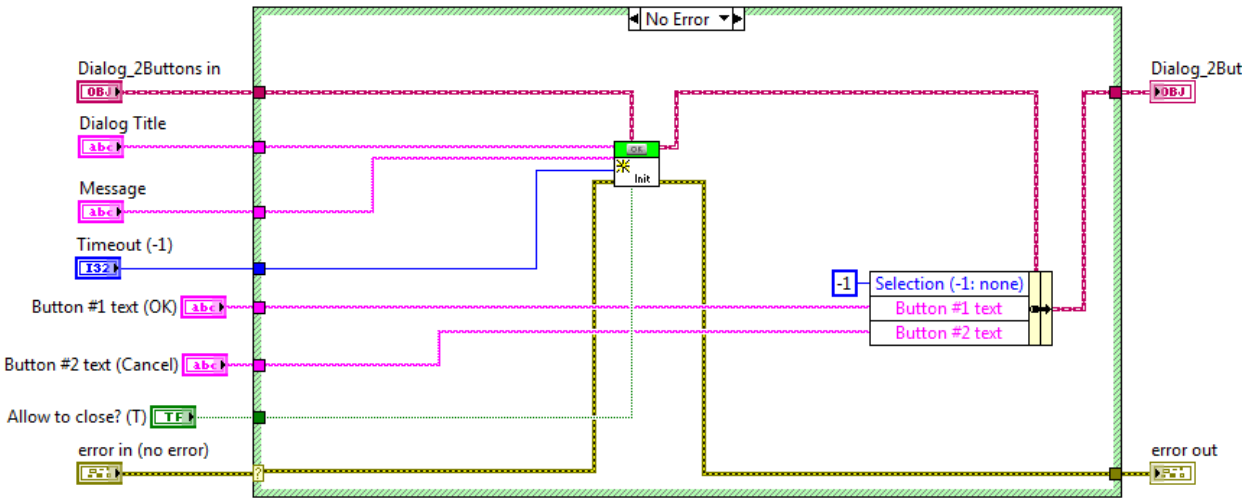
This is where the full benefits of object-oriented programming comes into play. The user will notice that there are no “Call Dialog Box” nodes for the other classes displayed on this palette. This is because you do not need them. Since all other classes inherit the basic functionalities of the parent class (One-Button Dialog), we will use its “Call Dialog Box” nodes with any other classes that inherit from it. See figure 2.3.

Figure 2.3 Dynamic Dispatching when wiring the child class to the Call Dialog Box node.



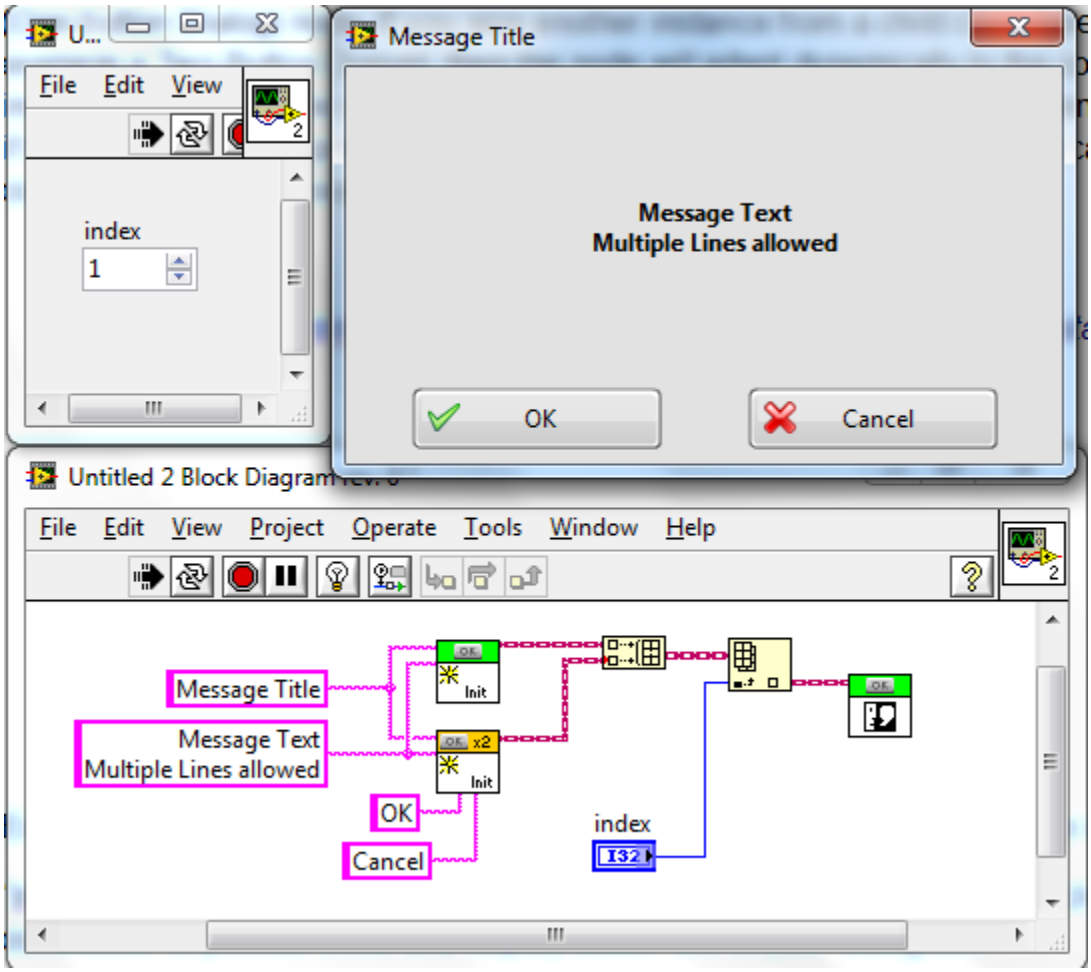
When you drop the “Call Dialog Box” node on the front panel, it shows as the base class (One-button Dialog) node. If you wire another instance from a child class type (for example a Two-Button Dialog), then the node will adapt dynamically to the correct instance. This looks like a simple polymorphism, but this one will also happen at runtime if the class instance type is unknown at runtime. Figure 2.4 shows the inside of the Two-Button Dialog initialization node.

Figure 2.4 Reusing the base class initialization node.



Inside the Two-Button Dialog Initialization node, we use the “Dialog in” terminal to inform the initialization node of the parent class that the object on the wire is of type “Two-Button Dialog”.

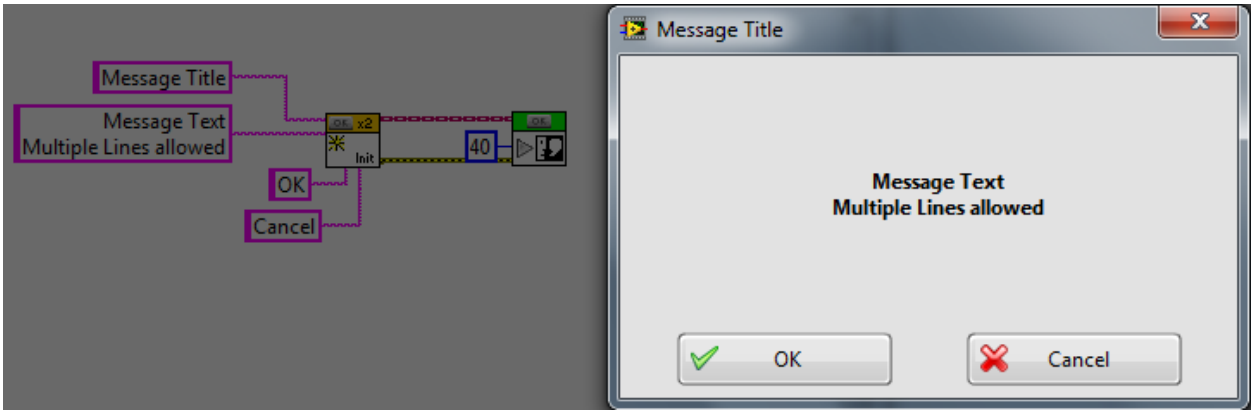
Figure 2.5 Runtime selection of the correct dynamicly-dispatched instance of the Call Dialog Box node.



If index were set to “0”, the One-Button Dialog would be called. This is possible because the child instances can “travel” on the parent class wire datatype. This is the main advantage of dynamic dispatching and inheritance.

USEFUL TIP: Replace the “Call Dialog Box” with the “Call Dialog Box (with blackening effect)” VI and you will have wrapped the functionality of the base class into all the other child classes...

Figure 2.6 Example code for a universal Dialog Box Caller.
(How cool is this, huh?)



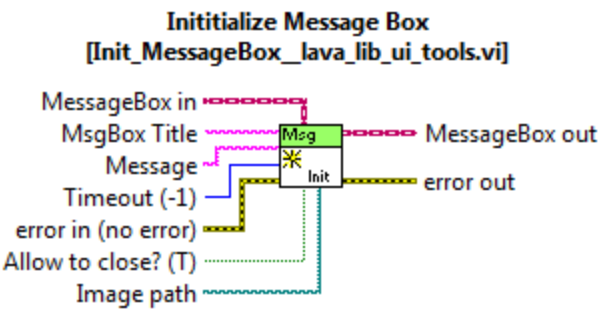
You'll notice that the “Call Dialog Box (with blackening effect)” VI does not have an override in the Two-Button Dialog class. Therefore, the base class node will always be the one being called, yet the code for Two-Button Dialog will be executed properly, with a blackening effect. No need to recode it multiple times, it is in the basic functionality of the parent class, therefore it is now a basic functionality of all the derived classes we will build on it.

“Message Box” class.

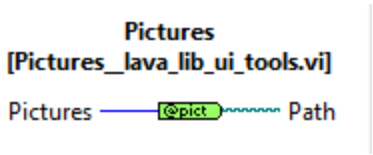
A third member of this class hierarchy is the “Message Box” with no buttons. It inherits all the functionalities of the One-Button Dialog.

SIDENOTE: One could ask why this is not the base class... and this person would be perfectly right! The reason is simply historical as the first class that was coded was a One-Button dialog. The author chose to keep it as a base class for backward-compatibility. That's a small inconvenient since all future classes could simply inherit from the Message class, including another type of “One-Button Dialog”.

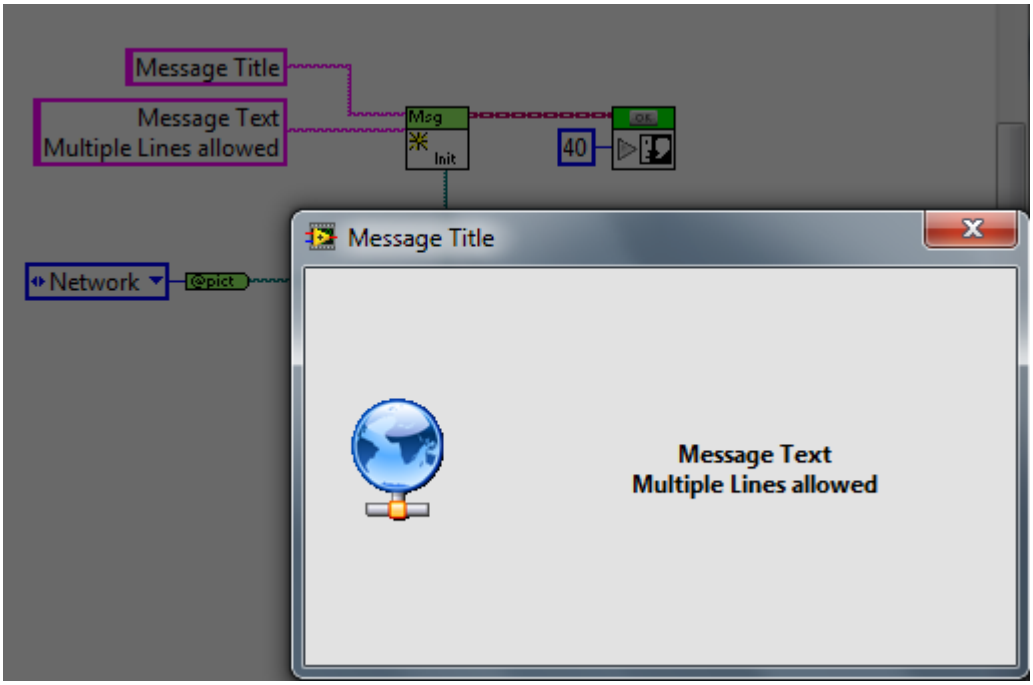
The main addition to this class is to provide a container for an image. In the absence of any images, a simple text message is displayed.



“Pictures” is a node that will get the filepath from a list of PNG images packaged with the class. If the user builds an executable with this node, he should make sure that the list of pictures are added to a directory named “Pictures”, located relative to the base directory of the executable.



The filepath of the initialization node can also be wired with an absolute path for any pictures of type PNG. (JPG and BMP will be added to a future version)



“**Error Message**” is the fourth member of this class series. It gives a simple interface for displaying an error message to the user. It features a “Show Details” boolean terminal to show or hide the details of the Call Chain. The user can toggle it from the main dialog box display. All that is needed is to wire the upper “Error” terminal and call the dialog box like the other classes.

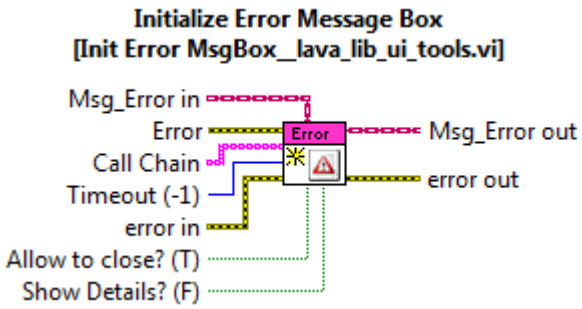
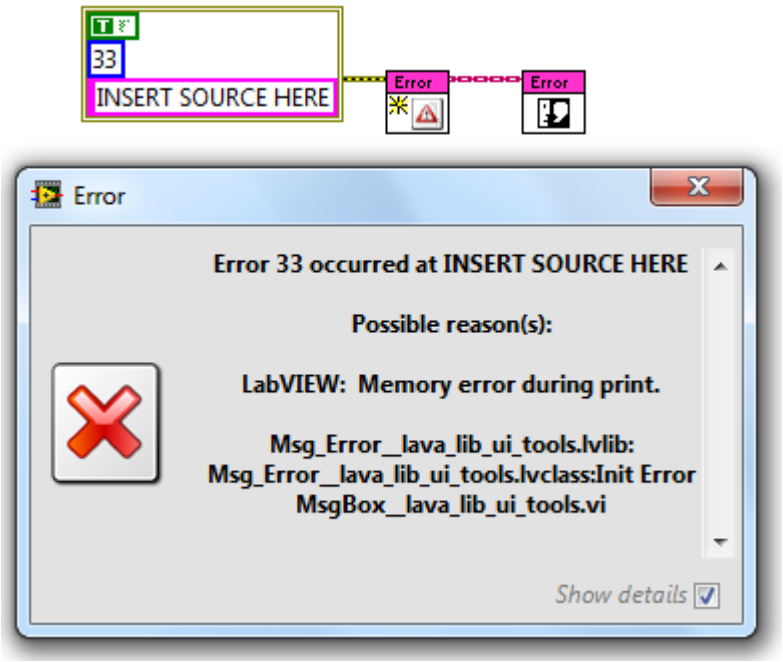


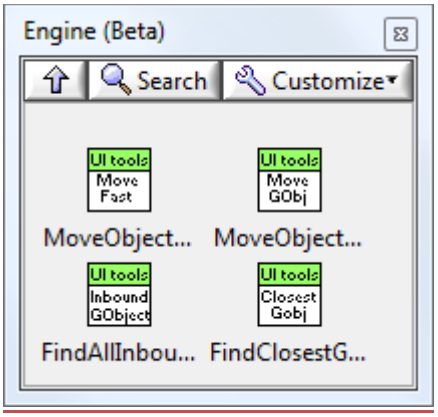
Figure 2.7 Example code for the Error Message dialog box.



Upcoming extension classes include an Image Gallery. Stay tuned.

Search the LabVIEW examples using *Dialog* keyword for quickly finding an example VI for this palette.

ENGINE (BETA):



Move Object on Front Panel
Move Object Fast
Find All Inbound GObjects
Find Closest GObject

"*MoveObjectonFP.vi*" is a utility that will translate an object (GObject refnum) to a set of front panel coordinates at a specified speed (slow, fast, instantaneous) using a basic algorithm that gives the effect of a deceleration.

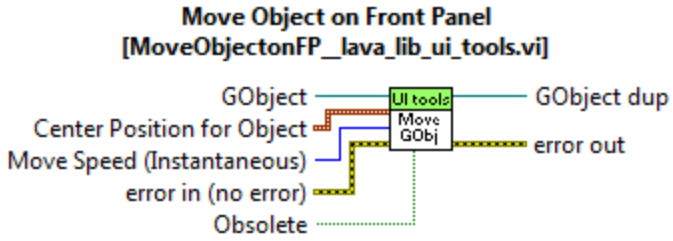
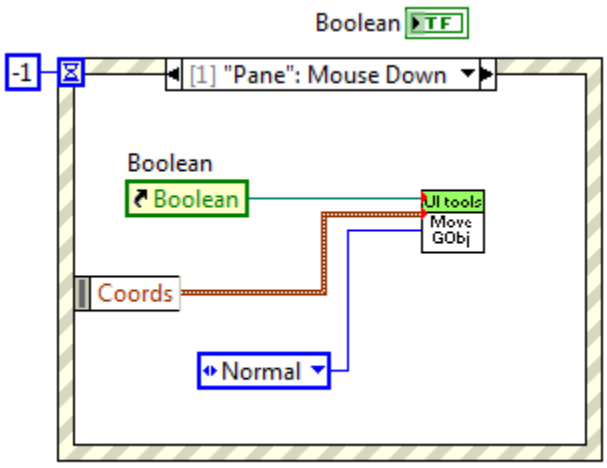
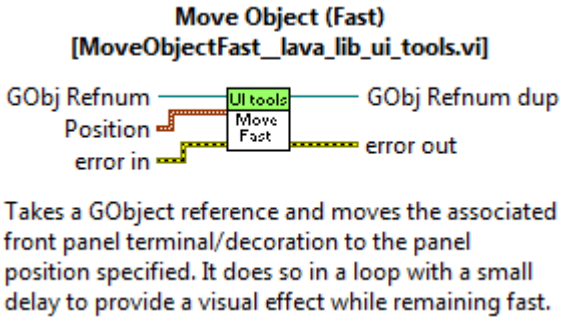


Figure 3.1 Example code for moving an object on the front panel.



(Setting the speed to "Normal" will move the object to its new location in less than a second)

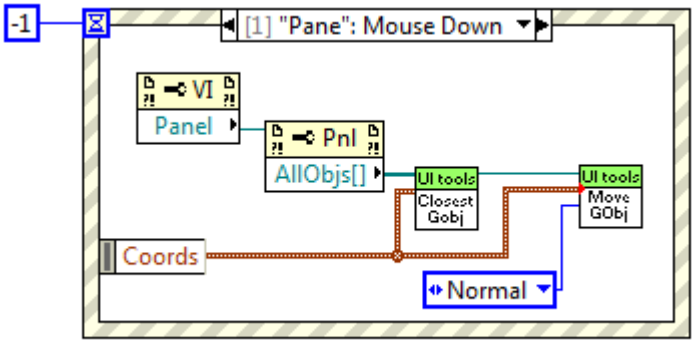
"*MoveObjectFast.vi*" is the same utility, but wrapped-up to simplify the connector pane. Simply give a set of front panel coordinates and a reference, most frequently using a mouse event in a "User Event" case as shown in figure 3.1.



"*FindAllInboundGObjects.vi*" is a utility that will return the GObjects that encompass the coordinates supplied. For example, if you monitor a MouseDown Event, you can return the list of objects (more than one if objects superimpose) that include the coordinates of the mouse when it was clicked.

"*FindClosestGObject.vi*" is a utility that finds the closest object to the specified coordinates. It is sometimes useful to know which object is closest in case that there are many objects nearby and the click was outside the previous utilities scope.

Figure 3.2 Example code for moving the closest GObject to the selected coordinates.



Search the LabVIEW examples using *Move* or *Toolbars* keywords for quickly finding an example VI for this palette.