Chapter No. 2
"Customizing the User Interface"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.2 "Customizing the User Interface"
A synopsis of the book’s content
Information on where to buy this book

About the Author
Yik Yang is a test engineer living in Chicago who has specialized in automation and data analysis. Having worked in multiple fields such as semiconductor, automotive, and power, he has experience with different types of automation and understands what are the industries' needs.

He started his career after receiving his Bachelor's and Master's degrees in Electrical Engineering at Virginia Tech. In his career, he worked on automation projects that used CompactDAQ, PXI, FPGA, and so on in LabVIEW. He has also spent a lot of time with Lean Six Sigma and statistical analysis with JMP. He is a certified Professional Engineer (PE) in North Carolina and a Certified LabVIEW Developer (CLD).

I would like to thank my wife Qian and my son Elijah. They have given me a tremendous amount of love and support in the process of finishing this book. Most importantly, I would like to thank God, as without him I would not be alive and capable of writing this book after my terrifying bicycle accident, when I was directly hit by a truck.

For More Information:
LabVIEW Graphical Programming Cookbook

LabVIEW is a graphical programming language by National Instrument. Mainly, it is used in test and automation fields for instrument control, data acquisition, and so on. In this book, we will cover different areas in LabVIEW programming with practical examples that follow sound coding standard and design rules.

What This Book Covers

Chapter 1, Understanding the LabVIEW Environment, covers common settings and functions in the LabVIEW environment.

Chapter 2, Customizing the User Interface, covers different functions used in creating an UI.

Chapter 3, Working with Common Architectures, covers common architectures in LabVIEW.

Chapter 4, Managing Data, covers how to use memory efficiently and control the data flow.

Chapter 5, Passing Data, covers different methods of passing data.

Chapter 6, Error Handling, covers different methods of error handling.

Chapter 7, Working with Files, covers how to work with different file types.

Chapter 8, Understanding Data Acquisition, covers acquiring data with different types of instruments.

Chapter 9, Simplifying Code, covers ways to simplify code.

Chapter 10, Working with External Code and Applications, covers how to use external code and application within LabVIEW.

For More Information:
Customizing the User Interface

In this chapter, we will cover:

- Customizing controls
- Adding a menu for runtime
- Creating a dialog
- Sizing the dialog automatically
- Using 2D picture control
- Updating controls with an action engine
- Creating a simple animation
- Creating subpanels

Introduction

This chapter presents tips on creating a user interface. We will demonstrate different UI features in UI design, such as customizing a control, adding runtime menu, creating a dialog, using 2D picture control, creating simple animation, and creating subpanels. Ways to manage UI, such as sizing a dialog automatically and updating controls with action engine, will also be demonstrated.

For More Information:
Customizing the User Interface

Customizing controls

In LabVIEW, there are different styles of controls to choose from. Control styles such as classic, modern, and so on would fulfill the majority of UI design needs. In addition, there are other styles that can be downloaded from OpenG. Nevertheless, at times, we need a control to look in a particular way and we can customize such special controls in LabVIEW.

Getting ready

To complete this recipe, the LabVIEW development environment is required. In this example, LabVIEW 2012 is used. To customize the numeric control, one picture of a blue car and one picture of a yellow car are needed.

How to do it...

Follow the given steps to implement the recipe:

1. Open a new project and VI.
2. We will start building the front panel. To create a track with a race car, right-click on the front panel and go to Classic | Numeric | Horizontal Slide.
3. Right-click on the control and select Make Type Def.
4. Right-click on the control and select Open Type Def.
5. In the type def, click on Change to Customize Mode (wrench icon).
6. Right-click on the knob and select Import from File at Same Size, as shown in the following screenshot:

For More Information:

7. Browse to the blue car image and adjust the size of the car.
8. Right-click on the control and select **Change to Indicator**.
9. Save the control and go to **File | Apply Changes**.
10. Repeat steps 2-8 for the yellow car image.
11. To create a signal that starts/stops the race, right-click on the front panel and go to **Modern | Boolean | Round LED**.
12. Right-click on **Round LED** and select **properties**.
13. Keep the On color as green and change Off color to red.
14. Enable **Show boolean text and Multiple strings**.
15. For the On text, type **GO**, and for the Off Text, write **STOP**.
16. Left-click on LED and drag its corner to enlarge it.
17. Right-click on the control and select **Change to Indicator**.
18. To create a status indicator, right-click on the front panel and go to **Classic | String & Path | Simple String**.
19. To make the surrounding of the Simple String transparent, press **Ctrl + Shift + right-click** on the front panel and select the coloring tool.
20. With the coloring tool, right-click on the surrounding of the Simple String indicator and select transparency for both background and foreground color. We get to see the following screenshot for the front panel after executing steps 1-17:

![Screenshot of front panel](image)

21. The front panel is completed, and we will work on the block diagram. We will use the state machine architecture. We will start by placing a while loop in the block diagram, and place a case structure within the while loop.
22. Create and save a `enum` type def. with states: Initialize, Idle, Run, Done, and Quit.

---

**For More Information:**

23. For the while loop, create four shift registers by right-clicking and selecting *Add Shift Register*; one for *BlueCar*, one for *YellowCar*, one for the *Result*, and one for the *State*. Wire these four registers and set the state shift register to *Initialize*. Place a *Wait Until Next ms Multiple* node inside the while loop and wire 500 ms to it. Create a local variable for the *Trigger* Boolean control and set it to *false* to initialize it. The stop condition is set with the Boolean default value, which is a *false*. The state machine and its *Initialize* state are shown in the following screenshot:

![State Machine Diagram](image1.png)

24. Build the "**Idle**" state with an event structure, as shown in the following screenshot:

![Event Structure](image2.png)

25. Build the second event case within the *Idle* state, as shown in the following screenshot:

![Second Event Case](image3.png)

For More Information:

26. Build the "Run" state, as shown in the following screenshot:

27. Build the "Done" state, as shown in the following screenshot:

For More Information:
28. Build the "Quit" state, as shown in the following screenshot:

![Screenshot](image)

**How it works...**

The program shown in the previous section is a racing simulation. Two customized controls with yellow and blue race cars are created, and they are driven by random numbers. A big LED is used to start the race, and the status of the race is shown with a simple string indicator. The images of the race cars were obtained from the Internet and a free software called GIMP ([http://www.gimp.org/](http://www.gimp.org/)) was used to make the background of the images transparent. For the status indicator, a Classic Simple String is used so that the background can be colored as transparent; the background will not show up and surround the car. To work with images in LabVIEW, Portable Network Graphics (png) format is preferred.

The program starts with the **Initialize** state to initialize all indicators and shift registers, and it proceeds to the **Idle** state. In the **Idle** state, it has an event structure. The program will not pass this state unless **Trigger** has a value change event. Once the Trigger value is changed, the program moves to the **Run** state. If the **Panel Close?** filter event occurs instead, the program will discard the **Panel Close?** event and stop the program. In this case, the **Panel Close?** event is filtered out, so the panel will not close. In the **Run** state, two separate random number generators are used to accumulate the distance traveled by each car. If either of cars crosses the finish line (within 50 ms), the program moves to the **Done** state; if not, the program comes back to the **Run** state. If the blue car wins, a Boolean value true is the output. If the yellow car wins, a Boolean value false is the output. The Done state will output the result based on the Boolean value and transit the program into the **Quit** state, where the program wraps everything up.

**See also**

- For more information on the state machine architecture, refer to Chapter 3, *Working with Common Architectures*

**For More Information:**
# Adding a menu for runtime

To make a program more professional, runtime menu is essential. In this recipe, we will create a program with runtime shortcut menu that is invoked by right-clicking on an indicator, and a runtime menu that resides on the top menu bar of the program.

## How to do it...

Follow the given steps to create the example for this recipe:

1. Open a new project and VI.
2. Create a chart indicator by right-clicking on the front panel and navigate to Modern | Graph | Waveform Chart.
3. Create a Boolean stop by right-clicking on the front panel and navigating to Modern | Boolean | Stop Button. First, we will start by creating a right-click shortcut menu for the chart.
4. Right-click on the chart and go to Advanced | Run-Time Shortcut Menu | Edit....
5. In the editor, select Custom in the top menu bar.
6. Under Item Properties, select User under Item Type and enter Increment as the Item Name.
7. Click on the add button on the top left to add another menu entry.
8. Under Item Properties, select User under Item Type and enter Decrement as the Item Name.
9. Click on the add button on the top left to add another menu entry.
10. Under Item Properties, select Separator under Item Type.
11. Click on the add button on the top left to add another menu entry.
12. Under Item Properties, go to Application item | Visible Items | Entire Menu.

---

For More Information:
Customizing the User Interface

13. Navigate to File | Save | Save to file and save the menu as RunTimeShortcut.rtm in the same folder as the VI. The following screenshot is of a completed Shortcut Menu:

14. After we finish the right-click shortcut menu, we will proceed to adding a custom top menu bar.

15. To create custom top menu bar for the front panel, go to Edit | Run-Time Menu....

16. In the editor, select Custom in the top menu bar.

17. Under Item Properties, select User under Item Type and enter Tasks as the Item Name.

18. Click on the add button on the top left to add another menu entry.

19. Under Item Properties, select User under Item Type and enter Multiplied by 2 as the Item Name.

20. Click the right arrow to make Multiplied by 2 a subitem of Tasks.

21. Click on the add button on the top left to add another menu entry.

22. Under Item Properties, select User under Item Type and enter Divided by 2 as the Item Name.

23. Click on the right arrow to make Multiplied by 2 a subitem of Tasks.

24. Go to File | Save and save the file as RunTimeTop.rtm in the same folder as the VI.

For More Information:
25. Create the front panel as shown in the following screenshot:

26. Create the block diagram as shown in the following screenshot:

27. Create the second event case, "Chart": Shortcut Menu Selection (User), as shown in the following screenshot. For the inner case structure, the "Decrement" case is shown. The case structure has two more cases that are not shown: Increment and Default. For the Increment case, use the increment node. For the Default case, just wire the input wire through.

For More Information:
28. Create the last event case as shown in the following screenshot. For the inner case structure, the "Divided by 2" case is shown. The case structure has two more cases that are not shown. They are Multiplied by 2 and Default. For the Multiplied by 2 case, use the multiply node with 2 and the input wire as inputs. For the Default case, just wire the input wire through.

![Screenshot of menu selection](image)

**How it works...**

The program generates random numbers and displays them on a chart. The user can choose to scale or shift the numbers on the chart.

The event structure has three cases. The timeout case is triggered once every second, and it will generate an array of 10 random numbers and displays them on a chart. If the user right-clicks on the chart and selects **Increment/Decrement**, the shortcut menu selection event is triggered, and the values of the chart are shifted accordingly. If the user selects the **Tasks** menu at the top and selects **Multiplied/Divided by 2**, the menu selection event is triggered and the values of the chart are scaled accordingly.

### Creating a dialog

Dialog is a simple pop up that will gather information from user for the program. In this recipe, we will create a simple dialog.

**How to do it...**

Dialog is a SubVI that would pop up to gather information. To create a simple dialog, we need to execute the following steps:

1. Create a new project and VI.
2. Create the front panel as shown in the next screenshot. The set of controls on the right are within a type-def. cluster and it needs to be created and saved.
3. To set the default values for all the controls, we can either write a value to each control by using a local variable, or set a default value manually. To set a default value manually, enter the default value into the control. Right-click on the control. Select **Data Operations** and **Make Current Value to Default**.

For More Information:

4. On the front panel, click on the upper-right corner of the 4 x 2 x 2 x 4 connector pane and then click on Info Cluster to position the **Info Cluster** on the connector pane.

5. Create the block diagram as shown in the following screenshot:

![Block Diagram Screenshot]

For More Information:  
6. On the front panel top menu bar, go to File | VI Properties. In the edit dialog, select the Dialog option. Check the following screenshot. Under Window Run-Time Position, we can select at what location of the monitor the dialog appears.

7. Resize the front panel so that the cluster is out of sight.

8. To use the dialog VI in other applications, we simply drop the dialog VI into the application and wire up all the inputs and outputs. When the application execution flow arrives at the dialog VI, it will pop up and gather the information intended.

How it works...

This VI is used as a SubVI inside a program. When the program calls this dialog VI, the dialog pops up, and the program thread that depends on the data of the dialog stops. The user would enter all the information and click on OK. The program populates a cluster and passes the data out of SubVI and into the program that calls it.

To make the dialog pop up, its appearance must be set to dialog inside VI properties. The option causes the VI to show it in the front panel during execution. Creating controls to get data from users and use a cluster to pass this data seems redundant. It is possible to use a cluster to get and pass the user data. However, when the controls are within the cluster, it has less flexibility for decoration.

For More Information:
See also

- If dialog is used to display a message without needing to gather information from users while allowing the program to continue, a dynamically-called dialog should be used. For further reference on this topic, refer to the Calling a VI dynamically recipe in Chapter 3, Working with Common Architectures.

Sizing the dialog automatically

The size of the dialog is adjusted and saved manually, so that only what is intended to be seen is shown when the dialog appears. However, every time when a dialog is modified, the size may need to be adjusted again. To avoid the trouble of adjusting the front panel every time when a change is made, this recipe provides a way to size a dialog automatically.

How to do it...

To start building the VI to size a dialog automatically, we need to execute the following steps:

1. We open a new project and VI.
2. Create the block diagram as shown in the following screenshots. The six values contained in `enum` are Initialize, Find Max Height, Set Origin, Set Panel Bounds, Center FP, and Shutdown. The first state `Initialize` gets the VI reference for the front panel which we would like to autosize. The pane reference is obtained, assuming that there is only one pane on the front panel. The references for all the decorations on the front panels are obtained.
3. Create the next case, "Find Max Height". It examines the height of all the decoration on the front panel and extracts the reference of the decoration with the maximum height. Refer to the following screenshot:

4. Create the next case "Set Origin", as shown in the following screenshot. It sets the origin of the front panel pane to the upper-left hand corner of the largest decoration:
5. Create the next case "Set Panel Bounds", as shown in the following screenshot. It sets the front panel bound to the largest decoration.

6. Create the next case "Center FP", as shown in the following screenshot. It centers the front panel when shown on the monitor.
7. Create the final state "Shutdown", as shown in the following screenshot. It closes all the references from property nodes used in previous states. It stops the state machine by setting the stop condition to true.

8. In the front panel, connect the path control used in the Initialize state to the upper-left hand corner of the terminals pattern. See the following screenshot:

9. Let's build a VI that uses the SubVI we built in steps 1-8. Build the front panel as shown in the following screenshot:

For More Information:
10. Build the block diagram as shown in the following screenshot:

How it works...

This recipe builds a SubVI AutoResizeVI.vi that is called by another VI to resize its front panel around a decoration frame. It uses a simple state machine as its architecture.

First, the state machine enters the **Initialize** state to open the references of the caller VI, caller VI's panel, caller VI's Pane, and caller VI's decorations on its front panel. The next **Find Max Height** state, the reference for the decoration with the max height is extracted from the decoration references array. In the next state **Set Origin**, the coordinates of the upper-left hand corner of the decoration is set to be the Pane's origin. In the next state **Set Panel Bounds**, the boundary of the decoration is set equal to the boundary of the front panel. In the **Center FP** state, the front panel is moved to the center of the monitor. Finally, in the **Shutdown** state, all references are closed, and the stop condition for the while loop is set to true to exit the program.

In the caller VI, it calls **AutoSize.vi** to resize its front panel to the size of its decoration. Then, it will wait for the user to click on **exit** to leave the program.

For More Information:


**See also**

- This recipe uses the state machine. For more information on the state machine architecture, refer to the Using the state machine architecture recipe in Chapter 3, Working with Common Architectures

## Using 2D picture control

In some test applications, it is beneficial to display test data in a real-time map, which contains test results with location information. For example, if we were to test a batch of products arranged in a rectangular grid, using a map to display data will provide a very good visual to display the location of the product and its test result. In this recipe, we will create a map that shows the test results for products in different coordinates, with the results shown in different colors.

### How to do it...

In this example, we will create a VI that calculates all the coordinates where we would like to map our test results. First, we will create a VI to calculate the coordinates.

1. Create the Coordinates.vi VI to generate an array of coordinates used for Draw Rectangle.vi. See the following screenshot:

   ![Coordinates.vi VI](image)

2. Create the main VI that calls Coordinates.vi. See the following screenshot:

   ![Main VI](image)

---

For More Information:

3. Right-click on 2D picture control and deselect **Erase First**.

**How it works...**

The example creates a 1D array of information to draw 50 x 50 rectangles in a large rectangular drawing area with Coordinates.vi. In each element of the array, it contains the top, bottom, left, and right of the rectangle in pixel.

The 1D array is indexed by a for loop that randomly draws each rectangle in the array with a different color. The 2D picture indicator is set to not erase every time it draws, so all rectangles will show up in the 2D picture indicator. See the following screenshot for a map drawn with rectangles with 8 pixels on each side:

For More Information:

Customizing the User Interface

Updating controls with an action engine

A complex application could consist of SubVIs that are multiple layers deep. It would be difficult for such a SubVI to update an indicator of the main VI. In this recipe, we will demonstrate an example to update controls of a main VI from a SubVI through an action engine.

How to do it...

Let's create the action engine. We will start with the Initialize command.

1. Create the action engine as shown in the following screenshot. The first command is Initialize. It obtains the reference of the VI with controls which we would like to update, obtains its panel reference, and the references of all the controls on the front panel of the VI.

For More Information:
2. Create the next command "Update Value" of the action engine, as shown in the following screenshot. With the label of a control, it finds the reference associated with the control, and update the value of the control. This is done on the control that the user would like to update.

![Update Value Screenshot](image)

3. Create `FindControlRef.vi` that is used in the preceding action case. It is used in other action cases as well. It extracts the control reference of the user-specified control. The build-in VI, called "Open VI Object Reference", can also perform the same function. See the following screenshot:

![FindControlRef Screenshot](image)
4. Create the next action case "Update DisEnable", as shown in the following screenshot. It finds the reference of the control with label that the user specified and then enables/disables the control as specified by the user.

5. Create the next action case "Update InVisible", as shown in the following screenshot. It finds the reference of the control with label that the user specified and then makes the control invisible/visible as specified by the user.

6. Create the last action case "Shutdown", as shown in the following screenshot. It closes all the references of the controls on the front panel.

For More Information:
7. Create the example, as shown in the following screenshot, to use the action engine:

How it works...

The action engine performs three functions: update value, visible status, and enable status of controls. It has two other functions as well: **Initialize** and **Shutdown** for configuration purposes. To use the action engine, the **Initialize** function needs to be executed first, so that all the references of the front panel controls are saved in the action engine. When the action engine is no longer needed, the **Shutdown** function is executed to close all the references. The other functions allow users to update the value, visible state, and enable state of a control by referring to its name. The example shows how the action engine is used.

See also

- To learn more about action engine, refer to the Creating an action engine recipe in Chapter 3, Working with Common Architectures

For More Information:

Creating a simple animation

A picture is worth a thousand words, and an animation is worth even more. In this recipe, we will create a simple animation with a picture ring.

Getting ready

To complete this recipe, the LabVIEW development environment is required. In this example, LabVIEW 2012 is used. For the animation, a set of fan blades pictures that are differed by 30 degrees are needed from 0 to 330 degree (12 pictures total). The pictures must be sized appropriately for the application.

How to do it...

We will start by creating the picture ring with snapshots of a blade's motion, which we will iterate through to create the animation:

1. Create a picture ring by right-clicking on the front panel and navigating to Classic | Ring & Enum | Pict Ring.
2. Drag the fan blade picture at 0 degree into the picture ring.
3. Right-click on the picture ring and select Add Item.
4. Repeat steps 2 and 3 until all pictures are added. For the last picture, step 3 is to be omitted.
5. Right-click on the control and select Make Type Def. Open the type def . and save it in the application folder.
6. Create the example, as shown in the following screenshot, to use the picture ring.
7. The front panel of the example is shown in the following screenshot. Use the decoration to create the base of the fan:

![Fan Animation](image)

**How it works...**

The animation is created by looping through and displaying a series of pictures in a loop. The example allows users to change the speed of the fan and to stop the fan with the stop Boolean.

**Creating subpanels**

Using subpanels allows the flexibility to embed multiple VIs into one at runtime. In this recipe, we will demonstrate how to use subpanels.

For More Information:  
How to do it...

We need to perform the following steps to make use of subpanels:

1. Create the state machine, as shown in the following screenshot. The first state **Initialize** creates three VI references by opening the VI template `SimpleAnimationExample.vit` three times. Only a VI template can be opened in this manner to create three VI references.

2. Create the next state "**Insert VIs**", as shown in the following screenshot. It builds an array with the references of the three subpanels that are on the front panels. The for loop inserts the three VIs opened previously into the subpanels.

For More Information:

3. Create the next state "Run VIs", as shown in the following screenshot. It runs the VIs inserted into the subpanels.

4. Create the next state "Exit", as shown in the following screenshot. It contains an event structure that will wait for the user to exit the program indefinitely.
In the previous recipe, open `SimpleAnimationExample.vi` and do a save as `*.vit` in another file. This recipe will call this template into the memory three times for three subpanels, and each time a new reference is generated. See the following front panel of the example:

![Front panel of the example](image)

**How it works...**

This example loads three separate instances of `SimpleAnimationExample.vi` into memory so that they can be executed independently. The example uses the state machine. It enters into the `Initialize` state first, and uses the template file `SimpleAnimationExample.vit` created previously to create three instances of the same VI. The next state, `Insert Vis`, inserts all the subpanels references into an array and iterates through each element in the array to insert the VIs into the subpanels. The next state, `Run Vis`, starts the VIs in each subpanel. `Wait Until Done` is set to `false`, so the program will not wait for the reference VI to finish running before proceeding. `Auto Dispose Ref` is set to `false`, so the main program that calls the reference VI is responsible to dispose of the references of referenced VIs. The last state, `Exit`, waits for the user to click on exit to terminate the program.

**See also**

- This recipe uses a state machine and dynamically calls VI. For more information on the state machine, refer to the Using the state machine architecture and Calling a VI dynamically recipe in Chapter 3, Working with Common Architectures
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.