6 Using Decision-Making Structures

In this lesson you will learn to create different decision-making structures and be able to identify applications where using these structures can be beneficial.

Topics

+ Case Structures
+ Event-Driven Programming

Exercises

Exercise 6-1 Temperature Warnings With Error Handling

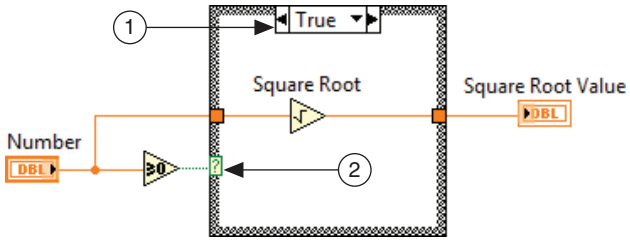
Exercise 6-2 Converting a Polling Design to an Event Structure Design

A. Case Structures

Objective: Recognize and use the basic features and functionality of Case Structures.

Activity 6-1: Case Structures Review

Figure 6-1. Case Structures Review



1 Case Selector Label
2 Selector Terminal

1. What is the purpose of the Case Structure?
   a. Execute one of its subdiagrams based on an input value
   b. Repeat a section of code until a condition occurs
   c. Execute a subdiagram a set number of times

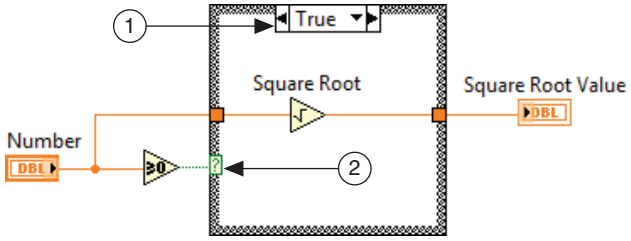
2. How many of its cases does a Case Structure execute at a time?
   a. All of them
   b. One

3. What is the purpose of the Case Selector Label?
   a. Lets you wire an input value to determine which case executes
   b. Show the name of the current state and enable you to navigate through different cases

4. What is the purpose of the Selector Terminal?
   a. Lets you wire an input value to determine which case executes
   b. Show the name of the current state and enable you to navigate through different cases

Activity 6-1: Case Structures Review - Answers

Figure 6-2. Case Structures Review: Answers

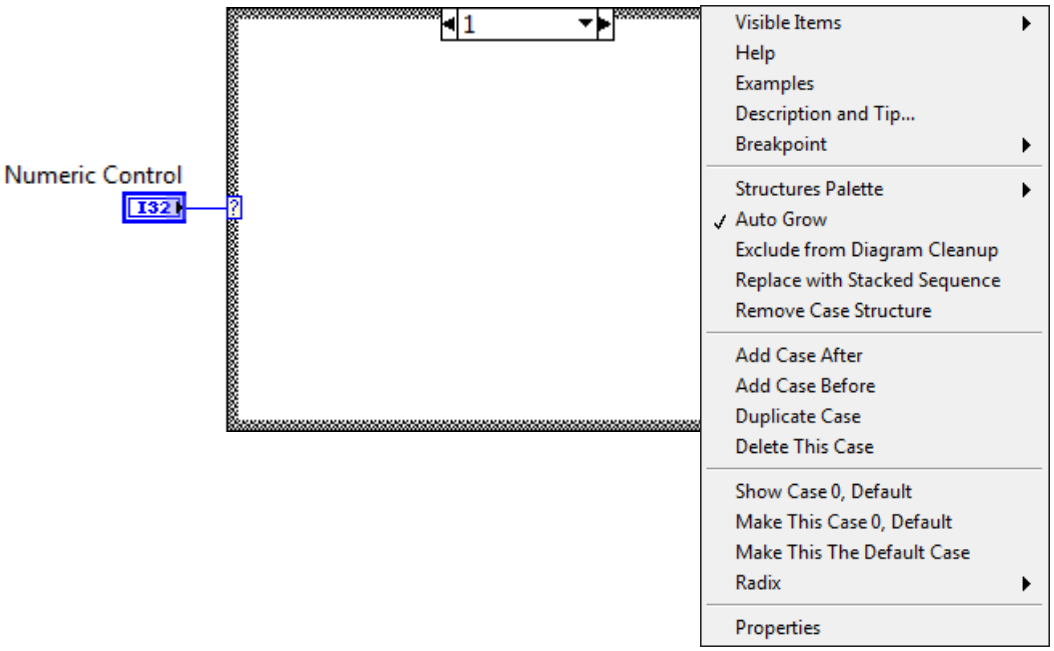


1	Case Selector Label	2	Selector Terminal

1. What is the purpose of the Case Structure?
   a. **Execute one of its subdiagrams based on an input value**
   b. Repeat a section of code until a condition occurs
   c. Execute a subdiagram a set number of times
2. How many of its cases does a Case Structure execute at a time?
   a. All of them
   b. **One**
3. What is the purpose of the Case Selector Label?
   a. Lets you wire an input value to determine which case executes
   b. **Show the name of the current state and enable you to navigate through different cases**
4. What is the purpose of the Selector Terminal?
   a. **Lets you wire an input value to determine which case executes**
   b. Show the name of the current state and enable you to navigate through different cases

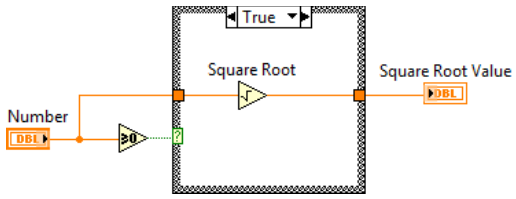
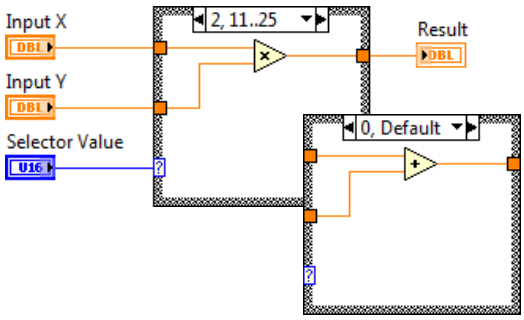
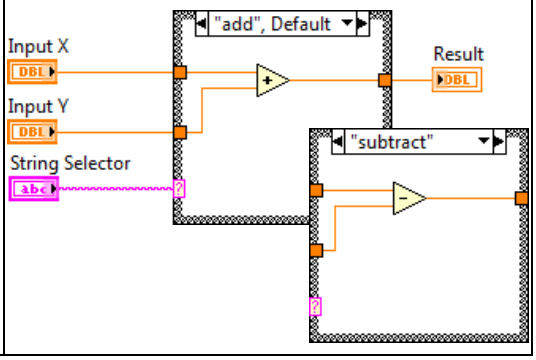
Demonstration: Case Structures

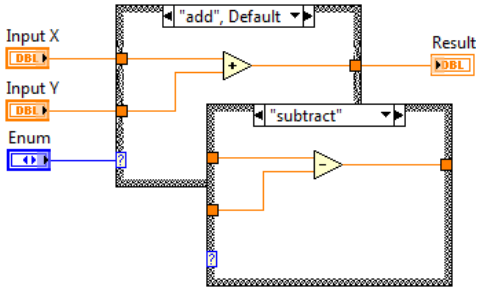
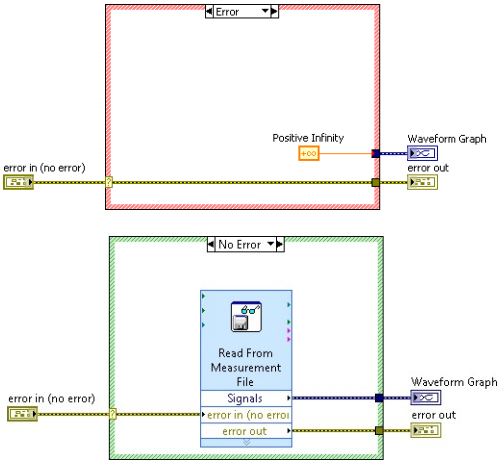
Right-click the Case structure to display the shortcut menu. The shortcut menu gives you options for configuring a Case structure.



Selector Terminal Data Types

You can wire a variety of data types to the Selector Terminal. The Case structure configuration changes based on the type of data that you connect.

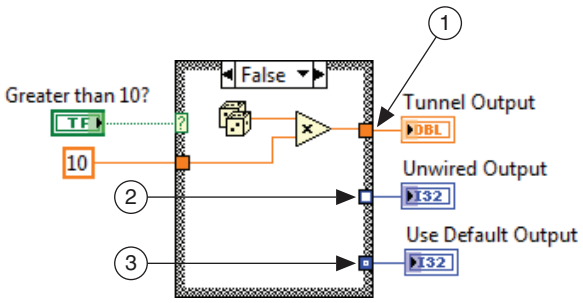
Data Type	Example
Boolean A newly-created Case structure defaults to a Boolean input. The Case Structure includes a True case and a False Case.	
Integer Case Structure has any number of cases. Specify a Default case. The numeric representation of the integer input will determine the range of possible values for the Case Selector Label. You can specify ranges of values for the Case Selector Label. Use Radix option in shortcut menu to specify whether the Case Selector Label displays values in decimal, hexadecimal, octal, or binary.	
String Case Structure has any number of cases. Specify a Default case. By default, string values are case sensitive. Shortcut menu includes option for **Case Insensitive Match** for the string text.	

Data Type	Example
Enum Possible to ensure that the Case Structure includes a case for every item in the enum. Right-click the border of the Case Structure and select **Add Case for Every Value** to create a case for every item.	
Error Cluster Case Structure includes an Error Case and a No Error Case. Wire an error cluster to the terminal to execute code if there is no error and skip code if there is an error.	

Input and Output Tunnels

As with other types of structures, you can create multiple input and output tunnels.

- Input tunnels are available to all cases if needed.
- Output tunnels require that you define a value for each case.
- Be cautious using the Use Default If Unwired option.
  - Adds a level of complexity to the code.
  - Can complicate debugging your code.
  - The default value may not be the value that you expect.

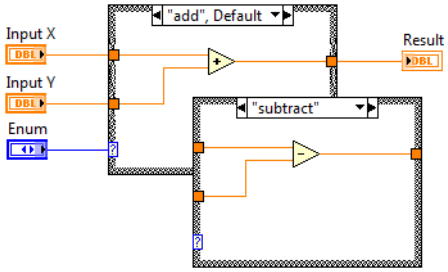


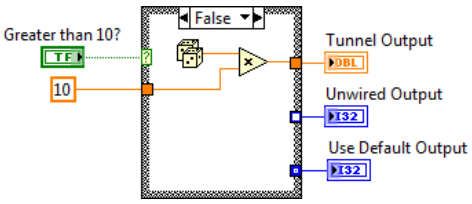
1 A tunnel that has been wired for all cases.
2 A tunnel that has not been wired.
3 A tunnel that is marked as Use Default If Unwired. Right-click on the tunnel and select **Use Default if Unwired**.

Data Type Wired to Tunnel	Default Value
Numeric	0
Boolean	FALSE
String	empty ("")

Demonstration: Selector Terminal Types and Tunnels

- Create Case structures using different data type selectors.
- Create different types of output tunnels.





Exercise 6-1 Temperature Warnings With Error Handling

Goal

Modify a VI to use a Case structure to make a software decision.

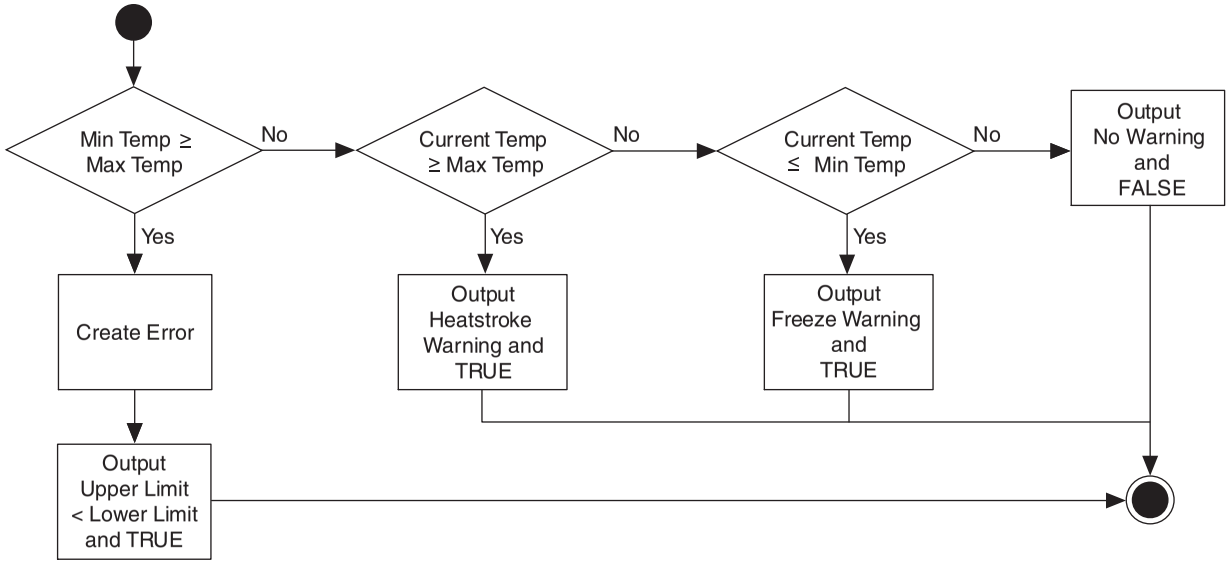
Scenario

You created a VI where a user inputs a temperature, a maximum temperature, and a minimum temperature. A warning string generates depending on the relationship of the given inputs. However, a situation could occur that causes the VI to work incorrectly. For example, the user could enter a maximum temperature that is less than the minimum temperature. Modify the VI to generate a different string to alert the user to the error: **`Upper Limit < Lower Limit`**. Set the **Warning?** indicator to TRUE to indicate the error.

Design

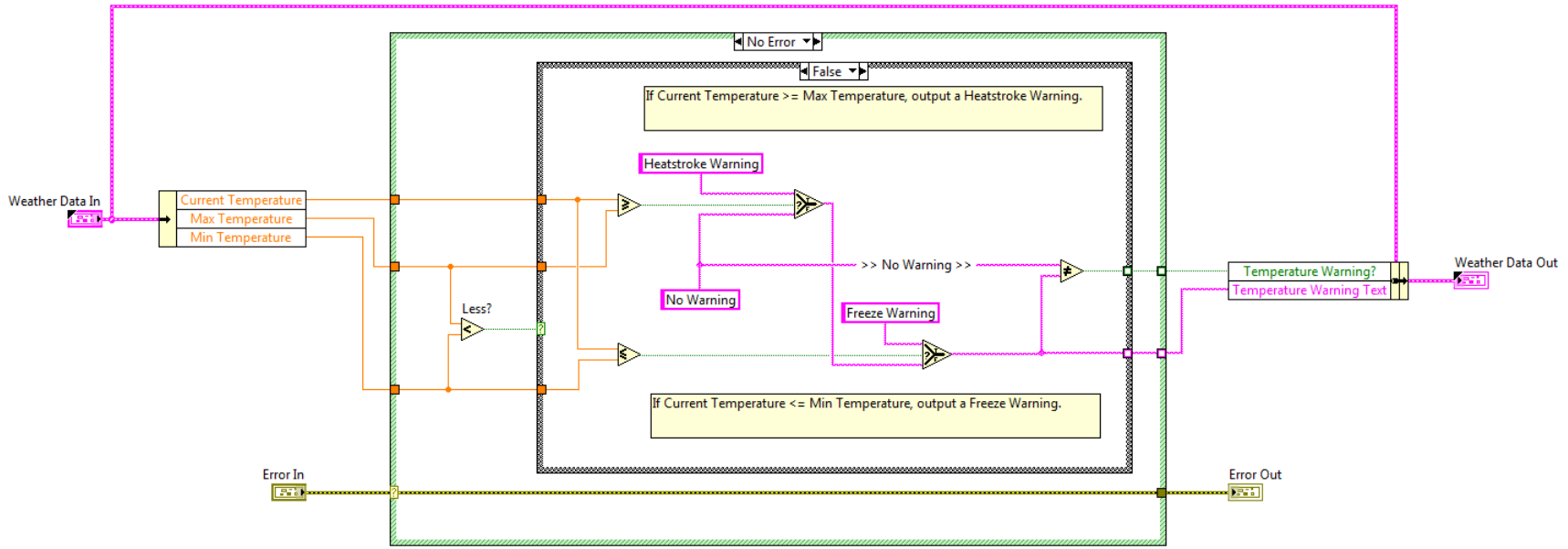
Modify the flowchart created for the original Temperature Warnings VI as shown in Figure 6-3.

Figure 6-3. Modified Temperature Warnings Flowchart



You must add a Case structure to the Temperature Warnings VI to execute the code if the maximum temperature is less than or equal to the minimum temperature. Otherwise, the VI does not execute the code. Instead, the VI generates a new string and the **Warning?** indicator is set to TRUE.

Figure 6-4. Original Temperature Warnings VI Block Diagram



Implementation

1. Open `Weather Warnings.lvproj` in the `<Exercises>\LabVIEW Core 1\Weather Warnings` directory.
2. Open **Temperature Warnings.vi** from the **Project Explorer** window.
3. Open the block diagram and create space to add the Case structure.

☐ Select the **Weather Data In type-defined cluster terminal, the Unbundle by Name function, and the Error In terminal**.

Tip To select more than one item press the <Shift> key while you select the items.

☐ While the objects are still selected, use the left arrow key on the keyboard to move the controls to the left.

Tip Press and hold the <Shift> key to move the objects in five pixel increments.

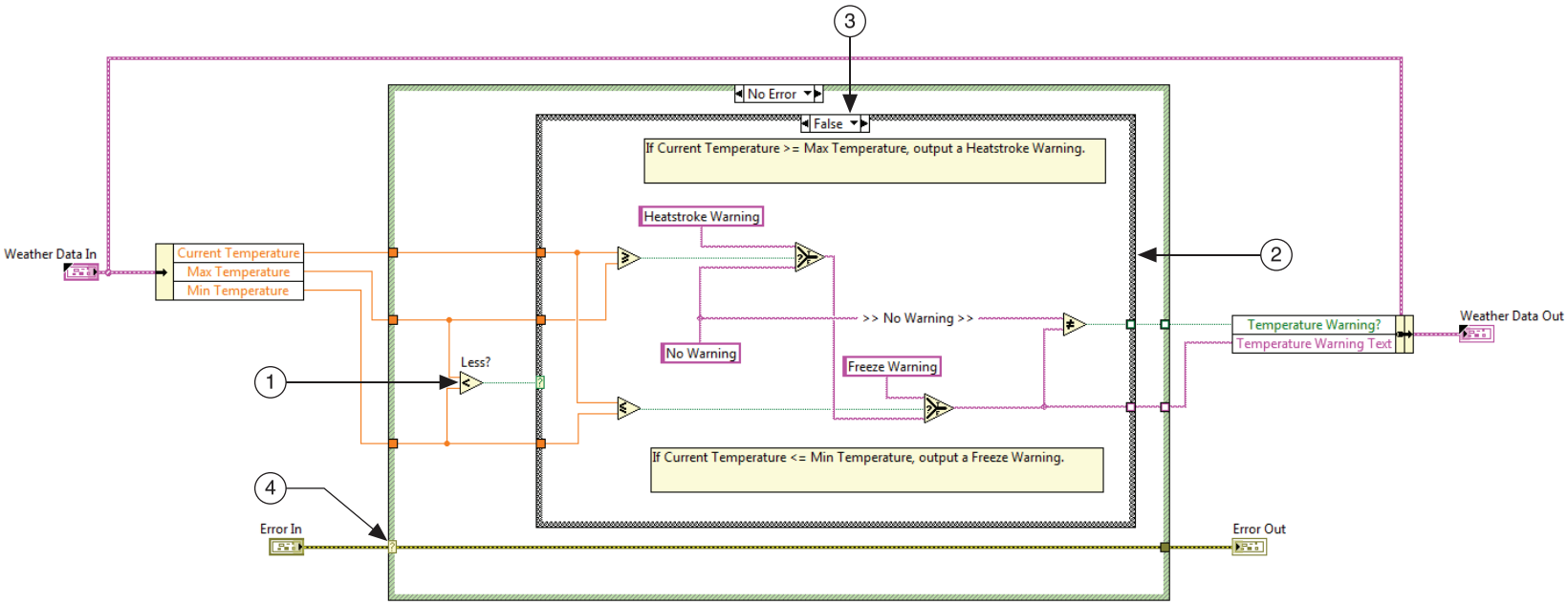
Tip Press the <Ctrl> key and use the Positioning tool to drag out a region of the size you want to insert.

☐ Select the **Weather Data Out type-define cluster terminal, the Bundle by Name function, and the Error Out terminal**.

- ☐ While the terminals are still selected, use the right arrow key on the keyboard to move the indicators to the right.
- ☐ Select the wire connecting the **Weather Data In** terminal and the **Bundle by Name** function.
- ☐ While the wire is still selected, use the up arrow key on the keyboard to move the wire upward.

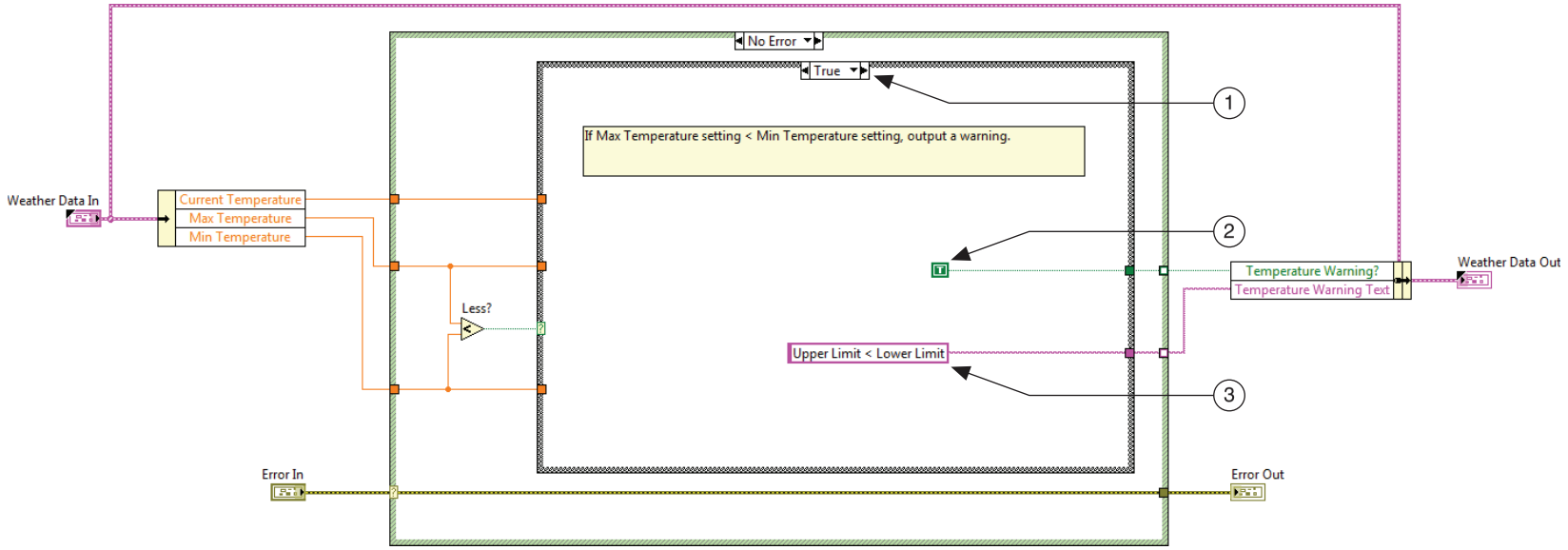
4. Modify the block diagram similar to that shown in Figure 6-5, Figure 6-6, and Figure 6-7. This VI is part of the temperature weather station project.

Figure 6-5. Temperature Warnings VI Block Diagram—No Error, False Case



1 **Less?**—Compares the Max Temperature and Min Temperature. Make sure the Less? function is outside the Case structure.
2 **Case Structure**—Do not include the **Weather Data In**, **Error In**, **Weather Data Out**, or **Error Out** terminals in the Case structure because these controls and indicators are used by both cases.
3 Set True and False cases—With the True case visible, right-click the border of the Case structure and select **Make this Case False**.
4 **Case Structure**—Wire the **Error In** terminal to the selector terminal to create No Error and Error cases. By default, the Case structure has True and False cases. These cases change to Error and No Error cases only after you wire Error In to the selector terminal.

Figure 6-6. Temperature Warnings VI—No Error, True Case



1 True case—If the Max Temperature is set lower than the Min Temperature, the True case executes. Click the case selector label to choose the True case.
2 **True Constant**—When the True case executes, the **Temperature Warning?** LED illuminates in the **Weather Data Out** cluster.
3 **String Constant**—If the Max Temperature is set lower than the Min Temperature, the warning `Upper Limit < Lower Limit` displays on the front panel. Enter the text in the String Constant.

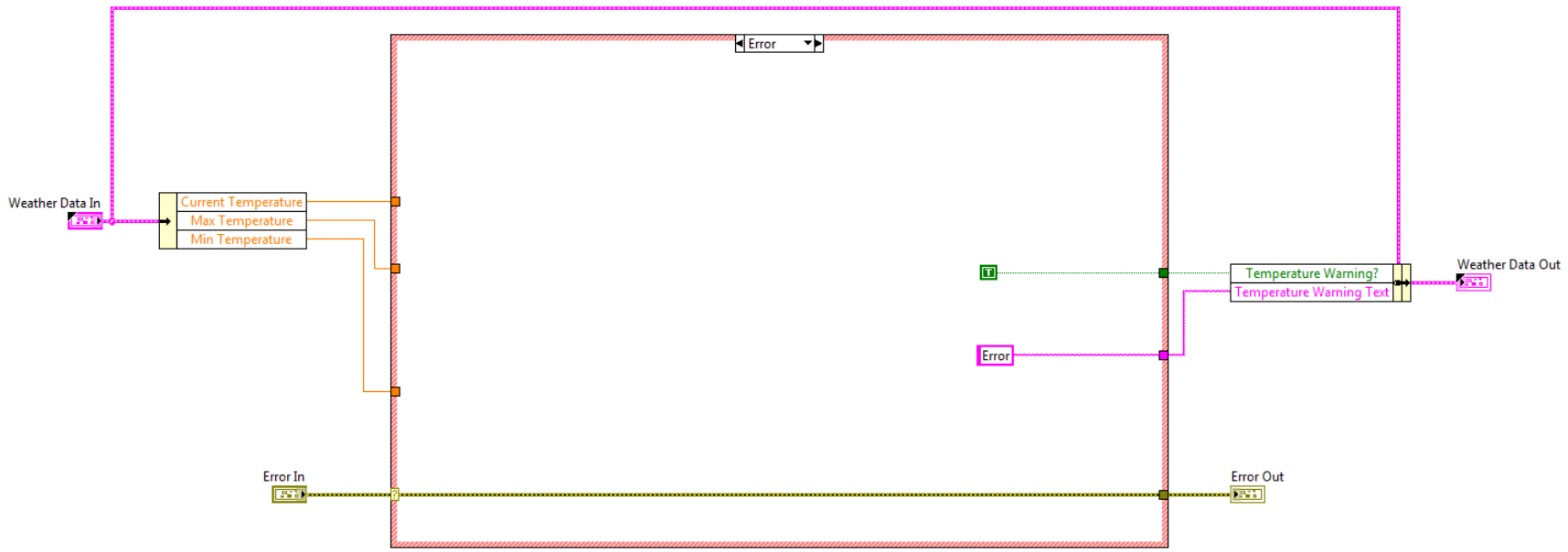
5. Predict the values for Temperature Warning Text and Temperature Warning? given each set of inputs.

Table 6-1. Predict Values for Temperature Warnings VI

Current Temperature	Max Temperature	Min Temperature	Temperature Warning Text	Temperature Warning?
30	30	10		
25	30	10		
10	30	10		
25	20	30		

6. Create the Error case in the outer Case structure so this VI can be used as a subVI.

Figure 6-7. Temperature Warnings VI—Error Case



7. Save the VI.

Test

1. Switch to the front panel of the VI.
2. Test the VI by entering values from Table 6-2 in the **Current Temperature**, **Max Temperature**, and **Min Temperature** controls and running the VI for each set of data.

Table 6-2 shows the expected Temperature Warning Text and Temperature Warning? Boolean value for each set of data.

Table 6-2. Testing Values for Temperature Warnings VI

Current Temperature	Max Temperature	Min Temperature	Temperature Warning Text	Temperature Warning?
30	30	10	**Heatstroke Warning**	True
25	30	10	**No Warning**	False
10	30	10	**Freeze Warning**	True
25	20	30	**Upper Limit < Lower Limit**	True

☐ Do these values match the values that you predicted?

☐ What happens if you set the value for all three inputs to 10?

☐ How could you address this problem?

3. Test the Error case. To use this VI as a subVI, the VI must be able to handle an error coming into the VI. Test the Error case to make sure that this VI can output the error information it receives.

   ☐ On the front panel, use the Operating tool to click the **status** Boolean indicator inside the **Error In** cluster so that the indicator turns red and enter 7 in the **code** control.

   ☐ Run the VI. The error information you entered passes through the Error case in the VI and is output in the **Error Out** cluster.

   ☐ Display the block diagram, select the No Error case, highlight execution, and then run the VI again to see the error pass through the Error case.

   ☐ On the front panel, right-click the border of the **Error Out** cluster and select **Explain Error** to display information about the error that was returned.

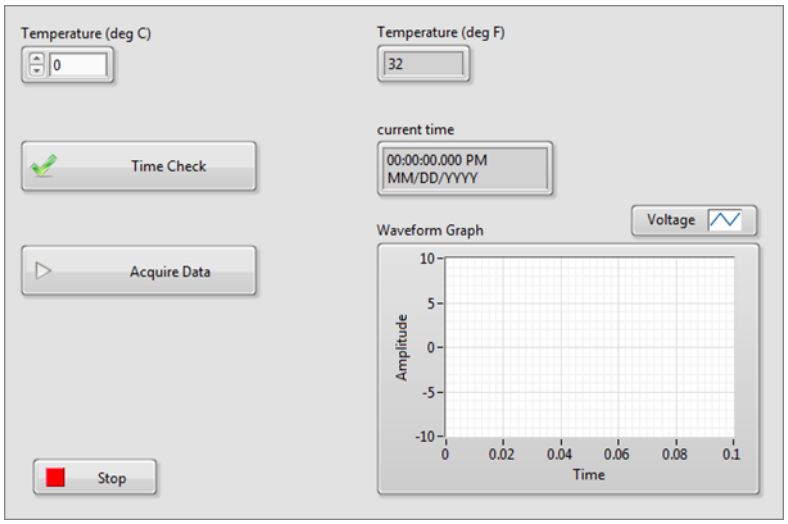
4. Save and close the VI.

End of Exercise 6-1

B. Event-Driven Programming

Objective: Recognize basic features and functionality of event structures.

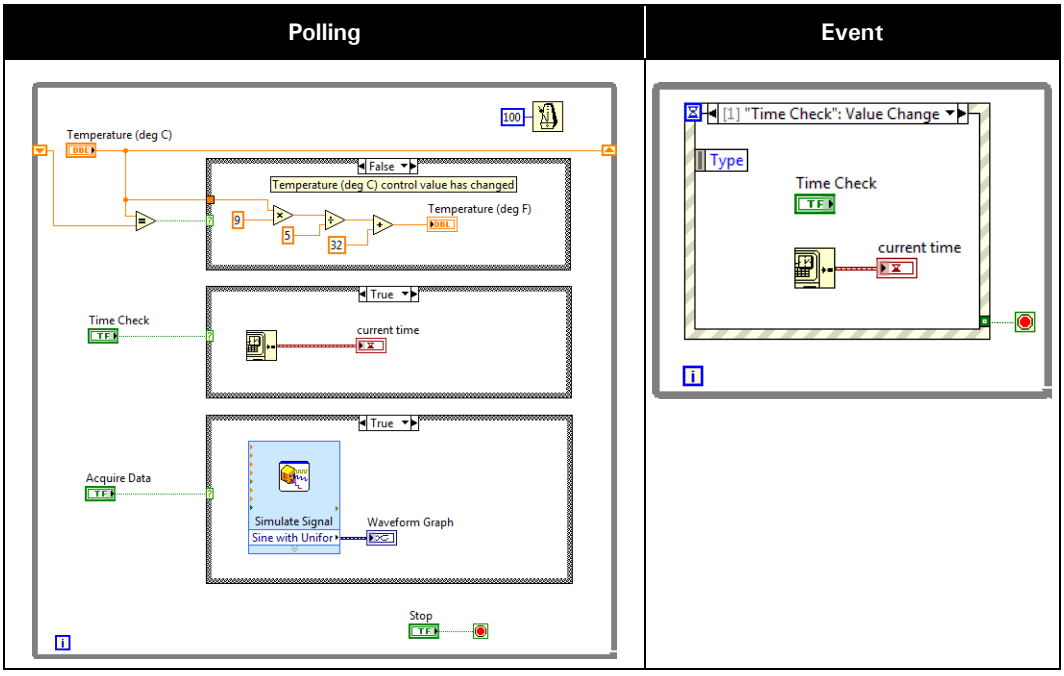
Demonstration: Event-Driven Scenario



Event-driven programming Method of programming where the program waits on an event to occur before executing the code written to handle that event.

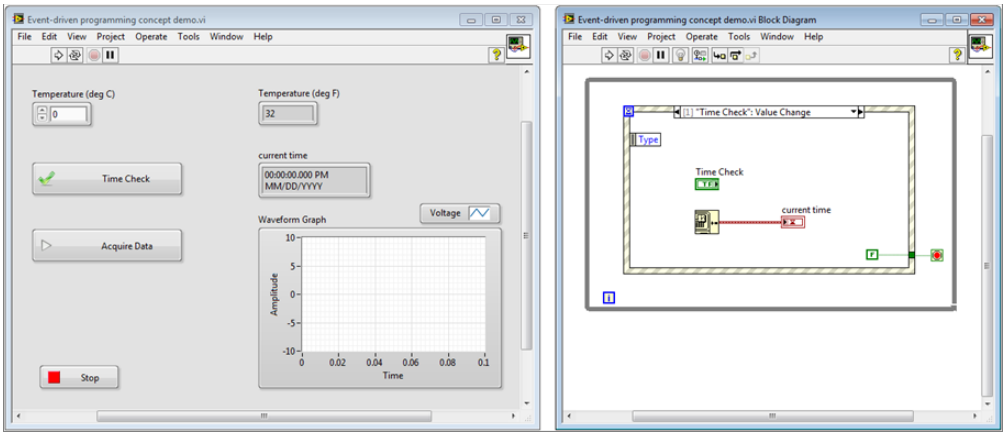
Event An asynchronous notification that something has occurred. Events can originate from the user interface, external I/O, or other parts of the program. In this course, you will only learn about user interface events, which include mouse clicks, key presses, and value changes of a control.

Polling Versus Events



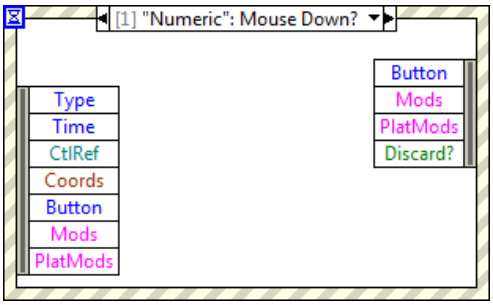
Multimedia: Event-Driven Programming

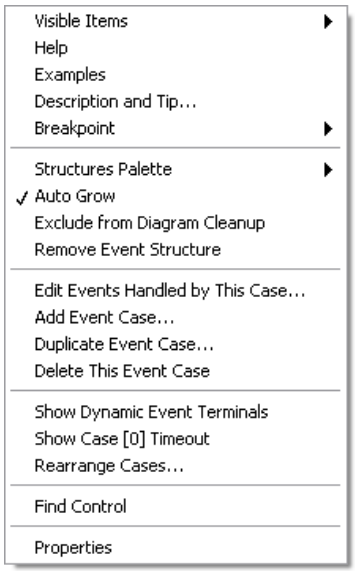
Complete the multimedia module, *Event-Driven Programming*, available in the `<Exercises>\LabVIEW Core 1\Multimedia\` folder to learn about programming with events.



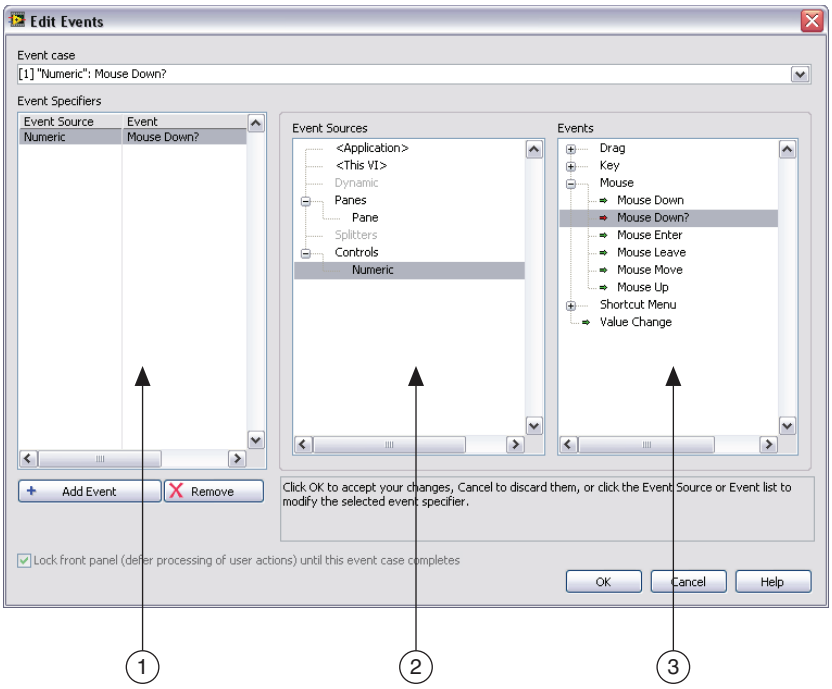
Configuring the Event Structure

You can select which events the Event structure implements by right-clicking the border and selecting **Edit Events Handled by This Case** from the shortcut menu





Edit Events Dialog Box

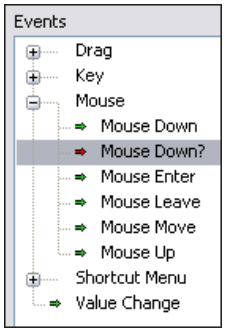


1	Configured events	2	Event sources	3	Events

Notify and Filter Events

LabVIEW categorizes user interface events into two different types of events:

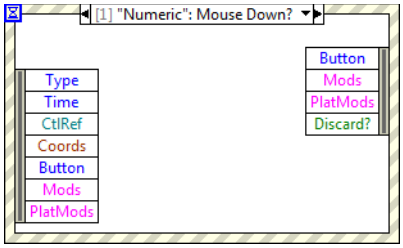
- Notify—(Green arrow) Notify events inform you that a user action occurred. LabVIEW has already performed the default action associated with that event.
- Filter—(Red arrow) Filter events allow you to validate or change the event data before LabVIEW performs the default action associated with that event. You also can discard the event entirely to prevent the change from affecting the VI.

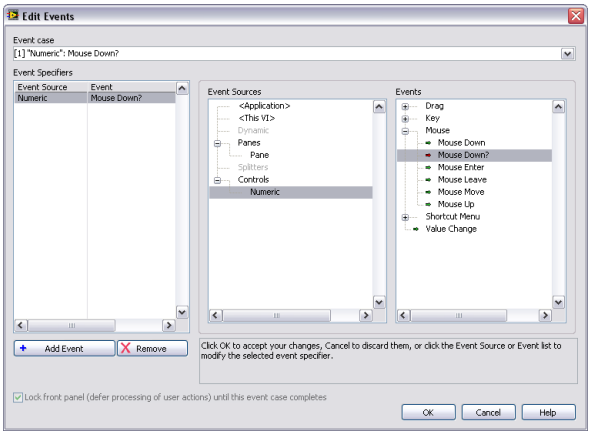


Note A single case in the Event structure cannot handle both notify and filter events. A case can handle multiple notify events but can handle multiple filter events only if the event data items are identical for all events.

Demonstration: Configure and Use Events

Configure and use an Event structure to create a VI that responds to user interface events using event-driven programming.





Exercise 6-2 Converting a Polling Design to an Event Structure Design

Goal

To convert a polling-based application to an event-based application and compare the different in performance.

Description

First you observe the behavior of a polling VI.

Next, you modify the polling VI to create a more efficient, event-driven VI and observe the changes in behavior.

Finally, you add different types of events to the VI.

Table 6-3 lists the events you will implement in the UI Event Handler VI you create.

Table 6-3. User Interface Events

Event	Event Description
“Stop”: Value Change	Stops the While Loop.
“Time Check”: Value Change	Displays a time stamp when you click the **Time Check** button.
“Pane”: Mouse Down	Displays the coordinates of the front panel point you click.
Panel Close?	Handles the event in which the user tries to close the running VI by clicking the window close button.
“Stop”: Mouse Enter	Produces a beep when the mouse cursor moves over the **Stop** button.

Observing the Polling VI Behavior

1. Open and run Polling.vi.
   - ☐ Open the `Events.lvproj` file in the `<Exercises>\LabVIEW Core 1\Events` directory and open the Polling VI from the project.
2. Examine the performance of a polling VI using the Windows Task Manager.
   - ☐ Press the <Ctrl-Alt-Delete> keys and select **Start Task Manager** from the menu.
   - ☐ Click the **Performance** tab in the **Windows Task Manager** window.
   - ☐ Run the VI.

- ☐ Notice how high the CPU usage is.
- ☐ Stop the VI and notice how the CPU usage drops.

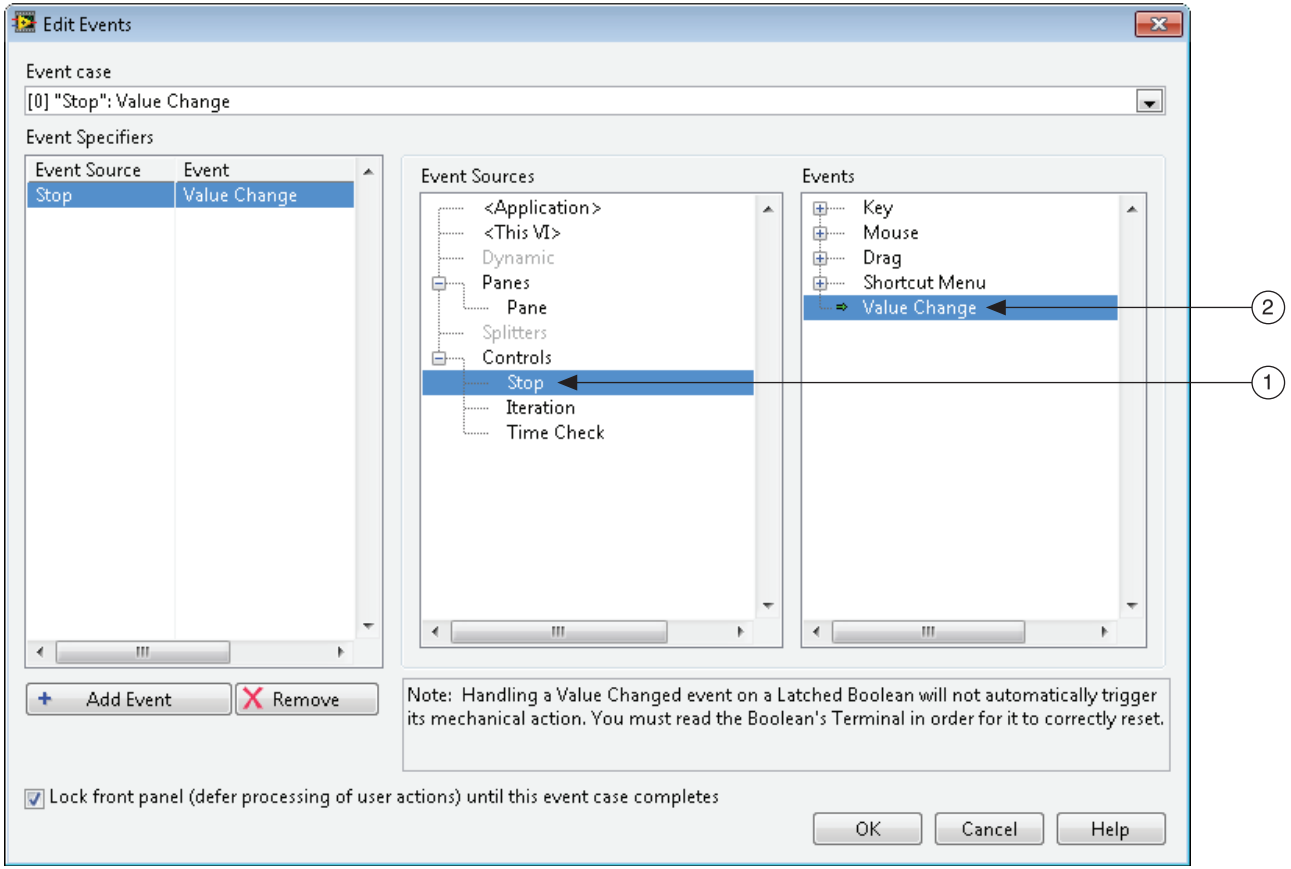
3. Open the block diagram, turn on execution highlighting, and run the VI again.
4. Notice how often the **Time Check** terminal sends data to the Case structure and how often the While Loop iterates.
5. Stop the VI and turn off execution highlighting.

Modifying the Polling VI to Use Events Instead of Polling

1. Save Polling VI as `UI Event Handler.vi` so you can modify it.
   - ☐ Select **Open additional copy** and add the copy to the project.
2. Close Polling.vi.
3. Open the block diagram of UI Event Handler.vi and move the **Stop** terminal and the **Time Check** terminal outside the While Loop. You move these terminals into the appropriate event cases later in this exercise.
4. Delete the Case structure and clean up any broken wires.
5. Place an Event structure inside the While Loop between the iteration terminal and the conditional terminal.
6. Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu.

7. Configure the event as shown in Figure 6-8.

Figure 6-8. Configuring the "Stop": Value Change Event

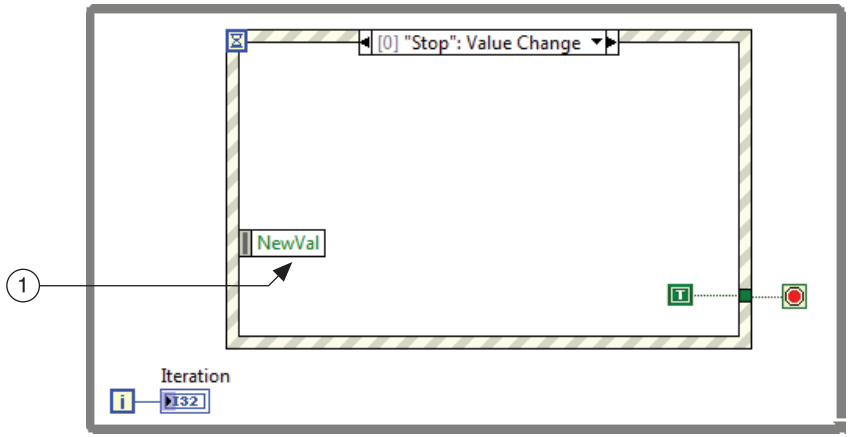


1 Click **Stop** in the **Event Sources** panel.
2 Click **Value Change** in the **Events** panel.

8. Click **OK** to close the dialog box.

9. Place a True constant inside the new **"Stop": Value Change** event and wire it to the conditional terminal of the While Loop as shown in Figure 6-9.

Figure 6-9. Event Structure with "Stop": Value Change Event



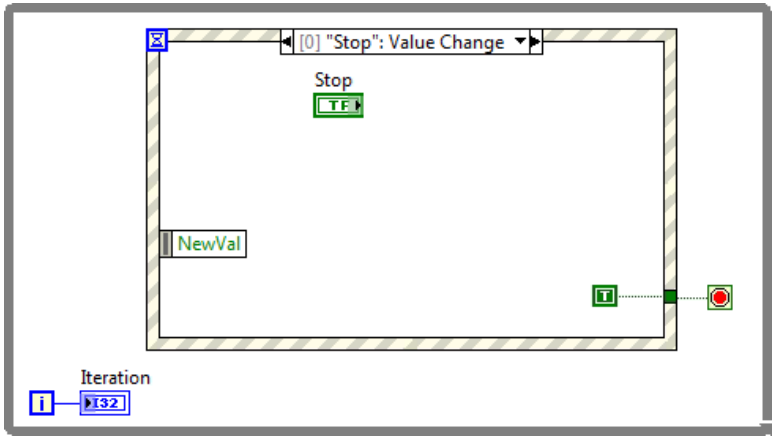
1 NewVal event data—Resize the event data items list so that only one item displays. Click the item and select **NewVal**.

Observing the Behavior of the Event-Driven VI

1. Run the VI.
2. Notice that the **Iteration** indicator does not increment.
3. Switch to the block diagram and enable execution highlighting.
4. Notice that the While Loop is executing the first iteration. The Event structure is waiting for an event.
5. Disable execution highlighting and switch back to the front panel.
6. Click the **Stop** button to stop the VI.
7. Notice that the VI stops running even though the **Stop** button is disconnected.
8. Notice that the **Stop** button stays depressed even though the mechanical action is set to **Latch When Released**. The reason the button stays depressed is because the VI stopped running after you clicked the button.
9. Reset the **Stop** button by clicking it again.

10. Drag the terminal of the **Stop** button into the "Stop": Value Change event as shown in Figure 6-10.

Figure 6-10. "Stop": Value Change Event with Stop Button Terminal



11. Run the VI and click the **Stop** button again.

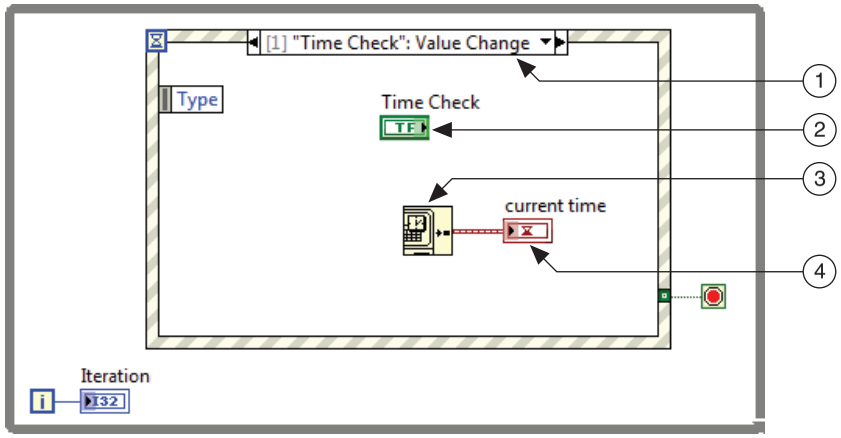
12. Notice this time the VI stops and the button resets.

Programming the "Time Check": Value Change Event

1. Add a new event case and create a “Time Check”: Value Change event as shown in Figure 6-11.

   ☐ Right-click the event structure and select **Add Event Case**.

Figure 6-11. Event Structure with "Time Check": Value Change Event



1 In the **Edit Events** window, select **Time Check** in the **Event Sources** panel and **Value Change** in the **Events** panel.
2 Move the Time Check terminal from outside the While Loop into the "Time Check": Value Change event case.
3 Get Date/Time In Seconds—Creates a time stamp in memory.
4 Indicator—Displays the **current time** output of the Get Date/Time In Seconds function.

2. Run the VI.

3. Click the **Time Check** button to see the current time display in the **current time** indicator.

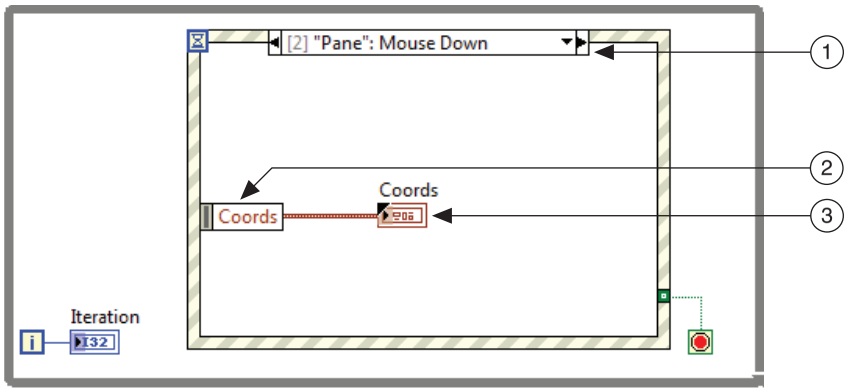
4. Display the Task Manager window and notice that CPU usage has decreased when you use events instead of polling.

5. Stop the VI.

Adding More Notify Events to the VI

1. Add a new event case and create a Mouse Down event as shown in Figure 6-12.

Figure 6-12. Event Structure with "Pane": Mouse Down Event



1 In the **Edit Events** window, select **Panes»Pane** in the **Event Sources** panel and **Mouse»Mouse Down** in the **Events** panel.
2 Coords event data—Click the event data node and select **Coords»All Elements**.
3 Coords indicator—Right-click the output of the **Coords** event data item and select **Create»Indicator** from the shortcut menu.

2. Run the VI.

3. Click on different parts of the front panel.

   ☐ Notice that the **Coords** indicator displays the coordinates for each point you click.

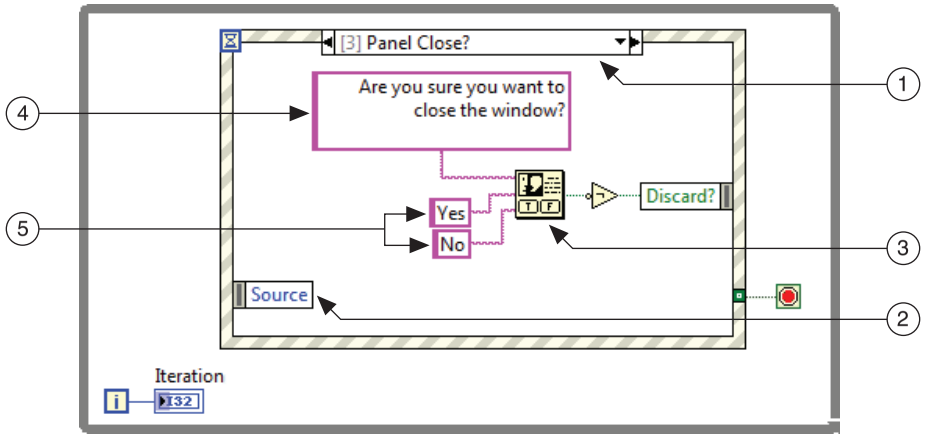
   ☐ Notice that the other events continue to behave as before.

4. Stop the VI.

Adding Filter Events to the VI

1. Add a new event case and create a Panel Close? event as shown in Figure 6-13.

Figure 6-13. Event Structure with Panel Close? Event



1 After you add the event, in the **Edit Events** window, select **<This VI>** in the **Event Sources** panel and **Panel Close?** in the **Events** panel.
2 Event data node—Click the Event Data Node and select **Source** from the menu.
3 Two Button Dialog function and Not function—Wire the **T button?** output to the Not function and wire the Not function to the Discard? event filter node.
4 String constant—Wire **Are you sure you want to close the window?** to the **message** input.
5 Yes and No string constants—Wire **Yes** to the **T button name ("OK")** input and wire **No** to the **F button name ("Cancel")** input.

2. Save and run the VI.
3. Click the "X" at the top-right of the window of the front panel.
4. Notice that clicking the **No** button cancels the event and returns to the VI.
5. Clicking the **Yes** button stops and closes the VI.
6. Stop the VI if necessary.

Challenge

1. If you have a sound card, add an event that produces a sound when the cursor is over the Stop button.

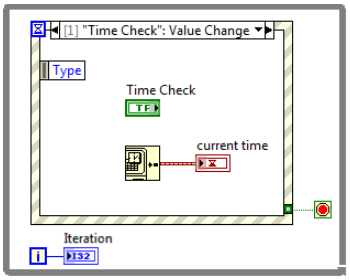
Tip Use Quick Drop to find the Beep VI.

End of Exercise 6-2

Caveats and Recommendations

The following list describes some of the caveats and recommendations to consider when incorporating events into LabVIEW applications.

- Place only one Event structure in a loop.
- Use a Value Change event to detect value changes.
- Keep event handling code short and quick.
- Place Boolean control terminals inside an event case for latched operations to work properly.
- Avoid using an Event structure outside of a loop.
- Avoid configuring two Event structures for the same event.



Think about the VIs that you will need to develop at your job.

Will you use event-based programming to implement any of your VIs? Why or why not?

Additional Resources

Learn More About	LabVIEW Help Topic or ni.com
Event-driven programming	*Locking Front Panels* *Choosing How the Event Structure Monitors For Events* *Viewing Enqueued Events at Run Time* *Event-Driven Programming* *Events in LabVIEW*
Deciding whether to use events	*Caveats and Recommendations when Using Events in LabVIEW*
Part of the Event structure	*Event Structure* *Configuring Events Handled by the Event Structure*
Determining which notifier to use	*Determining Which Type of User Interface Events to Use*

Activity 6-2: Lesson Review

1. Which of the following can NOT be used as the case selector input to a Case structure?
   a. Error cluster
   b. Array
   c. Enum
   d. String

2. How many events can an Event structure handle each time it executes?
   a. As many as have occurred since the last time the event structure executed
   b. One per configured event case
   c. One

3. Which statements about event-driven programming versus polling are true?
   a. Events execute on demand.
   b. Event-driven programming is less CPU-intensive.
   c. Event structures handle all events in the order the occur.
   d. Polling may fail to detect a change.

Activity 6-2: Lesson Review - Answers

1. Which of the following can NOT be used as the case selector input to a Case structure?
   a. Error cluster
   b. Array
   c. Enum
   d. String

2. How many events can an Event structure handle each time it executes?
   a. As many as have occurred since the last time the event structure executed
   b. One per configured event case
   c. One

3. Which statements about event-driven programming versus polling are true?
   a. Events execute on demand.
   b. Event-driven programming is less CPU-intensive.
   c. Event structures handle all events in the order the occur.
   d. Polling may fail to detect a change.