

Execution Structures: Detailed Explanation

While Loops

Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a While Loop, shown in **Figure 1**, executes the code it contains until a condition is met.

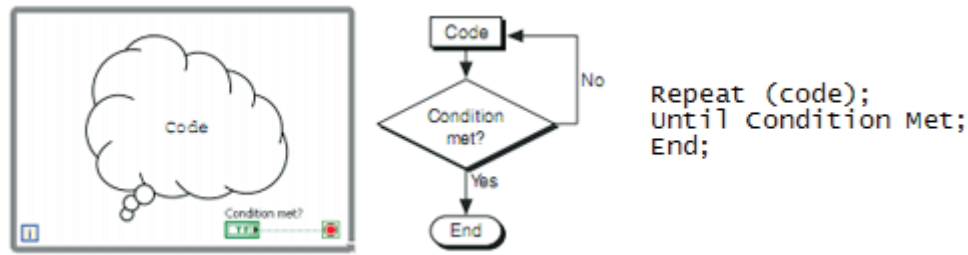


Figure 1. A While Loop in LabVIEW, a Flowchart of the While Loop Functionality, and a Pseudo Code Example of the Functionality of the While Loop

The While Loop is located on the **Structures** palette. Select the **While Loop** from the palette and then use the cursor to drag a selection rectangle around the section of the block diagram you want to repeat. When you release the mouse button, a While Loop boundary encloses the section you selected.

Add block diagram objects to the While Loop by dragging and dropping them inside the While Loop.

The While Loop executes the code it contains until the **Conditional Terminal**, an input terminal, receives a specific Boolean value.



You also can perform basic error handling using the conditional terminal of a While Loop. When you wire an error cluster to the conditional terminal, only the TRUE or FALSE value of the **Status** parameter of the error cluster passes to the terminal. Also, the **Stop if True** and **Continue if True** shortcut menu items change to **Stop if Error** and **Continue while Error**.

The **Iteration Terminal** is an output terminal that contains the number of completed iterations. The iteration count for the While Loop always starts at zero.

Note: *The While Loop always executes at least once.*

Infinite Loops

Infinite loops are a common programming mistake that involves a loop that never stops. If the **Conditional Terminal** is Stop if True, you place the terminal of a Boolean control outside a While Loop, and the control is FALSE when the loop starts, you cause an infinite loop.

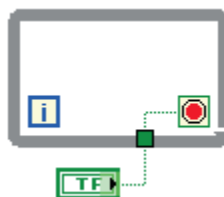


Figure 2. Boolean Control Outside While Loop

Changing the value of the control does not stop the infinite loop because the value is only read once, before the loop starts. To use a control to stop a While Loop, you should place the control terminal inside the loop. To stop an infinite loop, you must abort the VI by clicking the **Abort Execution** button on the toolbar.

In **Figure 3**, the While Loop executes until the Random Number function output is greater than or equal to 10.00 and the Enable control is TRUE. The And function returns TRUE only if both inputs are TRUE. Otherwise, it returns FALSE.

In **Figure 3**, there is an infinite loop because the random function never generates a value greater than 10.00.

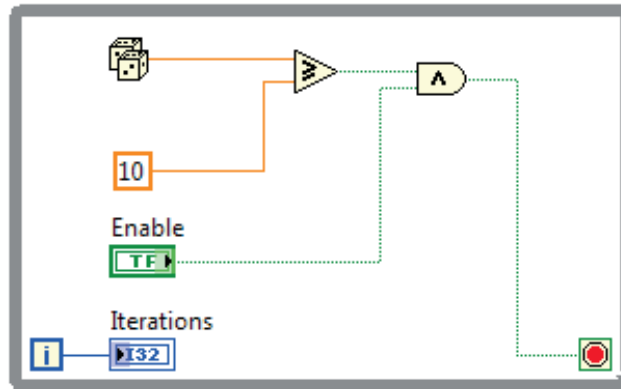


Figure 3. Infinite Loop

Structure Tunnels

Tunnels feed data into and out of structures. The tunnel appears as a solid block on the border of the While Loop. The block is the color of the data type wired to the tunnel. Data passes out of a loop after the loop terminates. When a tunnel passes data into a loop, **the loop executes only after data arrives at the tunnel.**

In **Figure 4**, the **Iteration Terminal** is connected to a tunnel. The value in the tunnel does not get passed to the Iterations indicator until the While Loop finishes executing. Only the last value of the **Iteration Terminal** displays in the Iterations indicator.

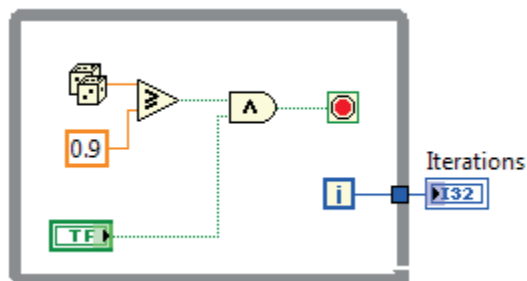


Figure 4. While Loop Tunnel

For Loops

A For Loop executes a subdiagram a set number of times. **Figure 5** shows a For Loop in LabVIEW, a flowchart equivalent of the For Loop functionality, and a pseudo code example of the functionality of the For Loop.

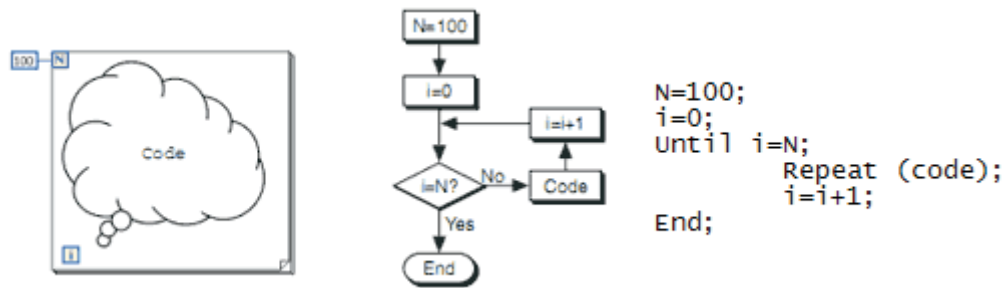


Figure 5. For Loop in LabVIEW, a Flowchart Equivalent of the For Loop Functionality, and a Pseudo Code Example of the Functionality of the For Loop.

The For Loop is located on the **Structures** palette. You also can place a While Loop on the block diagram, right-click the border of the While Loop, and select **Replace with For Loop** from the shortcut menu to change a While Loop to a For Loop.



The **Loop Count** terminal is an input terminal whose value indicates how many times to repeat the subdiagram.



The **Iteration Terminal** is an output terminal that contains the number of completed iterations. The iteration count for the For Loop always starts at zero.

The For Loop differs from the While Loop in that the For Loop executes a set number of times. A While Loop stops executing only if the value at the **Conditional Terminal** exists.

The For Loop in **Figure 6** generates a random number every second for 100 seconds and displays the random numbers in a numeric indicator.

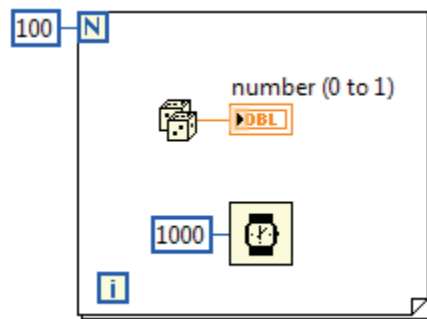


Figure 6. For Loop Example

Adding Timing to Loops

When a loop finishes executing an iteration, it immediately begins executing the next iteration, unless it reaches a stop condition. Most often, you need to control the iteration frequency or timing. For example, if you are acquiring data, and you want to acquire the data once every 10 seconds, you need a way to time the loop iterations so they occur once every 10 seconds. Even if you do not need the execution to occur at a certain frequency, you need to provide the processor with time to complete other tasks, such as responding to the user interface.

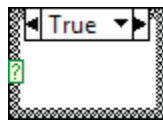
Wait Function

Place a wait function inside a loop to allow a VI to sleep for a set amount of time. This allows your processor to address other tasks during the wait time. Wait functions use the millisecond clock of the operating system.

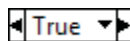


The Wait (ms) function waits until the millisecond counter counts to an amount equal to the input you specify. This function guarantees that the loop execution rate is at least the amount of the input you specify.

Case Structures



A Case structure has two or more subdiagrams, or cases. Only one subdiagram is visible at a time, and the structure executes only one case at a time. An input value determines which subdiagram executes. The Case structure is similar to switch statements or if...then...else statements in text-based programming languages.



The **Case Selector** label at the top of the Case structure contains the name of the selector value that corresponds to the case in the center and decrement and increment arrows on each side. Click the **Decrement** and **Increment** arrows to scroll through the available cases. You also can click the down arrow next to the case name and select a case from the pull-down menu.



Wire an input value, or selector, to the **Selector Terminal** to determine which case executes. You must wire an integer, Boolean value, string, or enumerated type value to the **Selector Terminal**. You can position the **Selector Terminal** anywhere on the left border of the Case structure. If the data type of the selector terminal is Boolean, the structure has a TRUE case and a FALSE case. If the **Selector Terminal** is an integer, string, or enumerated type value, the structure can have any number of cases.

Note: By default, string values you wire to the selector terminal are case sensitive. To allow case-insensitive matches, wire a string value to the selector terminal, right-click the border of the Case structure, and select Case Insensitive Match from the shortcut menu.

If you do not specify a default case for the Case structure to handle out-of-range values, you must explicitly list every possible input value. For example, if the selector is an integer and you specify cases for 1, 2, and 3, you must specify a default case to execute if the input value is 4 or any other unspecified integer value.

Note: You cannot specify a default case if you wire a Boolean control to the selector. If you right-click the case selector label, **Make This The Default Case** does not appear in the shortcut menu. Make the Boolean control TRUE or FALSE to determine which case to execute.

Right-click the Case structure border to **add, duplicate, remove, or rearrange** cases as well as select a **default case**.

Selecting a Case

Figure 7 shows a VI that uses a Case structure to execute different code dependent on whether a user selects °C or °F for temperature units. The top block diagram shows the TRUE case in the foreground. In the middle block diagram, the FALSE case is selected. To select a case, enter the value in the **Case Selector** identifier or use the Labeling tool to edit the values. After you select another case, the selected case is displayed on the block diagram, as shown in the bottom block diagram of **Figure 7**.

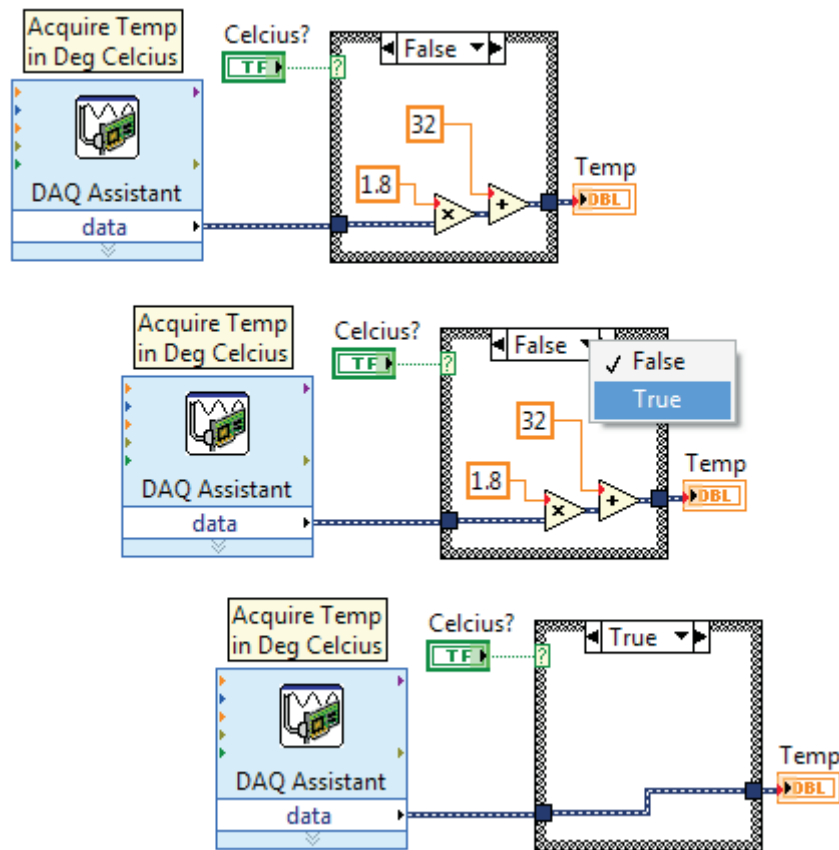


Figure 7. Changing the Case View of a Case Structure

If you enter a selector value that is not the same type as the object wired to the selector terminal, the value appears red. This indicates that the VI will not run until you delete or edit the value. Also, because of the possible round-off error inherent in floating-point arithmetic, you cannot use floating-point numbers as case selector values. If you wire a floating-point value to the case, LabVIEW rounds the value to the nearest integer. If you type a floating-point value in the case selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute.

Input and Output Tunnels

You can create multiple input and output tunnels for a Case structure. Inputs are available to all cases, but cases do not need to use each input. However, you must define an output tunnel for each case. Consider the following example: a Case structure on the block diagram has an output tunnel, but in at least one of the cases, there is no output value wired to the tunnel. If you run this case, LabVIEW does not know which value to return for the output. LabVIEW indicates this error by leaving the center of the tunnel white. The unwired case might not be the case that is visible on the block diagram. To correct this error, display the case(s) that contain(s) the unwired output tunnel and wire an output to the tunnel. You also can right-click the output tunnel and select **Use Default If Unwired** from the shortcut menu to use the default value for the tunnel data type for all unwired tunnels. When the output is wired in all cases, the output tunnel is a solid color.

Avoid using the **Use Default If Unwired** option. Using this option does not document the block diagram well, and can confuse other programmers using your code. The **Use Default If Unwired** option also makes debugging code difficult. If you use this option, be aware that the default value used is the default value for the data type that is wired to the tunnel. For example, if the tunnel is a Boolean data type, the default value is FALSE.







Data Type	Default Values
Numeric 	0 
Boolean 	FALSE 
String 	empty ("") 

Table 1. Data Type Default Values

Other Structures

LabVIEW has other, more advanced types of execution structures such as Event structures (used to handle interrupt-driven tasks like UI interaction) and Sequence structures (used to force execution order), which exceed the scope of this introductory material. To learn more about these structures, reference the appropriate LabVIEW Help topic.

Passing Data Between Loop Iterations in LabVIEW

When programming with loops, you often must access data from previous iterations of the loop in LabVIEW. For example, if you are acquiring one piece of data in every iteration of a loop and must average every five pieces of data, you must retain the data from previous iterations of the loop.

Shift Registers



Use shift registers when you want to pass values from previous iterations through the loop to the next iteration. A shift register appears as a pair of terminals directly opposite each other on the vertical sides of the loop border.

The terminal on the right side of the loop contains an up arrow and stores data on the completion of an iteration. LabVIEW transfers the data connected to the right side of the register to the next iteration. After the loop executes, the terminal on the right side of the loop returns the last value stored in the shift register.

Create a shift register by right-clicking the left or right border of a loop and selecting **Add Shift Register** from the shortcut menu.

A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type.

You can add more than one shift register to a loop. If you have multiple operations that use previous iteration values within your loop, use multiple shift registers to store the data values from those different processes in the structure, as shown in **Figure 8**.

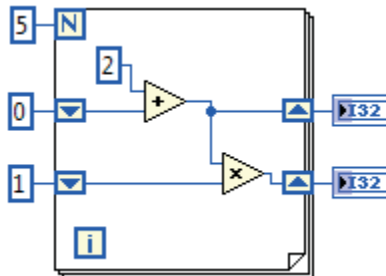


Figure 8. Using Multiple Shift Registers

Initializing Shift Registers

Initializing a shift register resets the value the shift register passes to the first iteration of the loop when the VI runs. Initialize a shift register by wiring a control or constant to the shift register terminal on the left side of the loop, as shown in **Figure 9**.

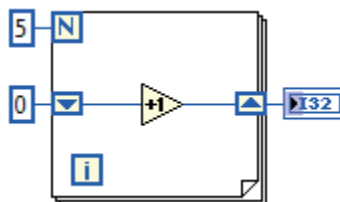


Figure 9. Initialized Shift Register

In **Figure 9**, the For Loop executes five times, incrementing the value the shift register carries by one each time. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator and the VI quits. Each time you run the VI, the shift register begins with a value of 0. If you do not initialize the shift register, the loop uses the value written to the shift register when the loop last executed or, if the loop has never executed, the default value for the data type. Use an uninitialized shift register to preserve state information between subsequent executions of a VI.

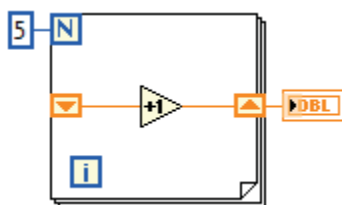


Figure 10. Uninitialized Shift Register

In **Figure 10**, the For Loop executes five times, incrementing the value the shift register carries by one each time. The first time you run the VI, the shift register begins with a value of 0, which is the default value for a 32-bit integer. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator, and the VI quits. The next time you run the VI, the shift register begins with a value of 5, which was the last value from the previous execution. After five iterations of the For Loop, the shift register passes the final value, 10, to the indicator. If you run the VI again, the shift register begins with a value of 10, and so on. Uninitialized shift registers retain the value of the previous iteration until you close the VI.

Stacked Shift Registers

With stacked shift registers, you can access data from previous loop iterations. The registers remember values from multiple previous iterations and carry those values to the next iterations. To create a stacked shift register, right-click the left shift register terminal and select **Add Element** from the shortcut menu.

Stacked shift registers can occur only on the left side of the loop because the right terminal transfers the data generated from only the current iteration to the next iteration.

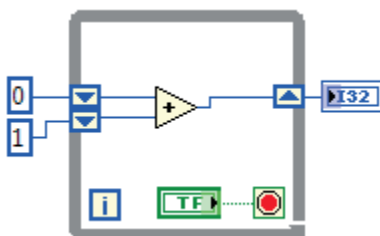


Figure 11. Using Stacked Shift Registers

If you add another element to the left terminal in the previous block diagram, values from the last two iterations carry over to the next iteration, with the most recent iteration value stored in the top shift register. The bottom terminal stores the data passed to it from the previous iteration.