Lab Assignment 1

Introduction

LabVIEW is a software package widely used in a variety of applications involving monitoring, measuring, testing and/or controlling of systems and processes. Additionally, LabVIEW is more than just a program, it is a graphical programming language; users create their own programs using a drag-and-drop style interface. The intention of the creators was to provide a relatively easy way for scientists and engineers to make their own programs to operate their equipment, run their tests, monitor their experiments, etc. In industry the uses of LabVIEW programs can be surprisingly diverse and powerful (for some examples on interesting projects using LabVIEW visit the company website, http://www.ni.com/labview/who_uses_labview.htm). It can be easy to get intimidated after looking at some of the top tier applications, but this is a scalable utility; LabVIEW programs are only as complex and complicated as you make them.

The goal of this introductory course in LabVIEW is not to turn anyone into an expert user, although that’s certainly an achievable goal for anyone in the class who puts their mind to it. Instead our goal is to introduce everyone to the software and establish a degree of comfort when working with it. Since this is a class focused primarily on taking measurements, this introduction to LabVIEW will be similarly structured; by the end of the term everyone should be able to create their own LabVIEW code to acquire readings and store them in a usable format.

Terminology

Before getting started there are a few important terms we should introduce and briefly define as they will be brought up repeatedly and they are integral to LabVIEW.

VI: Short for Virtual Instrument, this term is used to refer to LabVIEW programs. This is also the file format extension designating LabVIEW programs.

SubVI: VIs that are embedded inside of other VIs are sometimes referred to as subVIs. This is a broad category with a great deal of crossover; some subVIs have no use other than as a subcomponent of another VI while some VIs will occasionally be used as subVIs inside of other VIs.

Front Panel: The portion of a LabVIEW VI that the user interacts with while running the program. This is analogous to the face of a stereo or the front of an oscilloscope; here is where the user will see all the knobs, buttons, visual displays, etc.
**Block Diagram**: The portion of a LabVIEW VI that generally contains all of the functional programming. Continuing the stereo and oscilloscope analogies, this would be the circuit boards, vacuum tubes, wires and other guts usually contained within the case; this is where the majority of tinkering, programming and debugging will generally occur.

**Icon**: The visual representation of a VI’s components as seen on the Block Diagram. Most LabVIEW programming involves placing, configuring and connecting Icons.

**Controls**: Items on the Front Panel used to simulate input devices such as knobs and buttons. It is through these that the user generally manipulates all relevant parameters of a VI. Controls are generally seen on both the Front Panel and on the Block Diagram with a different appearance for each.

**Constants**: Sometimes a particular input either never changes or changes so infrequently that having the user repeatedly select its value is unnecessary. In those cases a Constant will usually suffice. Constants can either be numbers or strings and are useful for things such as messages displayed to the user, channel numbers, file names, etc.

**Indicators**: Displays on the Front Panel that provide the user with pertinent data, such as temperature, voltage, position, etc. Indicators can be anything from a light that turns on and off to continuously changing graphs to a text box showing a number. Indicators, like Controls, are usually visible on both the Front Panel as well as the Block Diagram, again with different appearances.

**Function**: This is an object on the Block Diagram representing one of the discrete sub processes of a VI, such as adding two inputs, acquiring a signal from a measurement device or saving a reading to a file. Functions are hidden from the user and do not appear on the Front Panel.

**Wire**: Graphical representation of the path of data flow. On the Block Diagram the Icons that interact are connected to each other using Wires. The color and style of line used to depict a Wire represents certain qualities about it; orange and blue Wires transmit numeric data, purple Wires transmit strings of text, bold Wires are 1-D arrays, and so on.

**Terminal**: The location on an Icon to which a Wire can be connected. The terminals on Icons are almost always use specific, meaning that Wires must be connected to the correct Terminals in order for a Function to give the desired output.

**Controls Palette**: This is where all of the objects usable on the Front Panel are found. It can always be brought up by right clicking anywhere on the Front Panel. It consists of all of the Controls and Indicators as well as other classes of Front Panel objects.
**Functions Palette:** This is where all of the Functions are found for use on the Block Diagram. Like the Controls Palette, it can always be brought up by right clicking anywhere on the Block Diagram.

**Structures:** In order to get LabVIEW to perform selected actions repeatedly, or only when certain conditions are met we use structures. These take the form of while loops, for loops, case structures and numerous other types, some of which are specific to LabVIEW.

There are numerous other terms and phrases with LabVIEW specific definitions, but these should be sufficient for the time being. Don’t worry if you don’t retain all of that information right away; the best way to learn it all is to use it.

This handout is structured from here on out with examples that you will be walked through from start to finish with commentary on what makes certain components work the way they do and occasionally notes that contain more in depth material that could be considered supplemental reading.

---

**Week 1: Controls, Constants, Indicators, Functions and Structures.**

Objectives: By the end of this lab exercise you should:

- Understand how to create new VIs and save them.
- Know how to find items on the Controls and Functions palettes and place them.
- Be able to wire functions together to create a working VI.
- Feel comfortable using While Loops and Case Structures in a VI.
- Grasp the difference between a graph and a chart in LabVIEW.

Probably the easiest way to become acquainted with the inner workings of a LabVIEW VI is to create a relatively simple one and watch it as it runs. For our first example, we’ll be creating a simple sentence generator.

**Example 1: Sentence Generator**

Start LabVIEW on your computer and select the Blank VI option from the welcome screen. Many people find working with LabVIEW easier when the Front Panel and Block Diagram are tiled, either side by side or one on top of the other; you can easily tile a selected VI by hitting Ctrl+T or selecting one of the Tile options from the View menu.
For the examples in this lab, all of the components you will need can be found under the *Modern* heading on the Controls Palette and the *Programming* heading on the Functions Palette.

The final version of the front panel and block diagram for this exercise are shown in Figures 1 and 2, respectively.

<table>
<thead>
<tr>
<th>Proper noun</th>
<th>Generated Sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>Berlin is the proud owner of 43 symponies.</td>
</tr>
</tbody>
</table>

**Figure 1.** Final version of the front panel for the sentence generator.

**Figure 2.** Final version of the block diagram for the sentence generator.

**Steps to create the Sentence Generator VI:**

1. Place a String Control (from the Controls Palette) on the Front Panel. Change the title on the String Control to Proper Noun by double clicking on the text. If you can remember to do it right away, you can change the titles of Front Panel items such as Controls and Indicators if you type immediately after placing them; as soon as you mouse click anywhere else the title field is deselected.

2. Now place another String Control and change its title to Objects.
3. The last item for the Front Panel is a String Indicator. Change its title to Generated Sentence and drag one of its borders to make its display larger. You can do this by clicking on the display field then clicking on one of the small black boxes that appears. If you don’t make the display field large enough, you will not see the entirety of the text output.

4. Now for the Block Diagram. Find and place a String Constant from the Functions Palette under the String category. Type into it “is the proud owner of” with a space both before and after.

5. In order to get a random number of proudly owned objects, open the Numeric menu on the Functions palette and select Random Number. Now place the Multiply function, also found in the Numeric menu. You may inadvertently notice that if you place the Multiply function so that one of its wiring nodes is very close to the Random Number wiring node, a wire automatically appears connecting them. This is an optional feature of LabVIEW that is on by default, though you can modify this in the Tools → Options → Block Diagram menu. In this case it is fairly harmless but oftentimes functions will get automatically wired in ways that can ruin their functionality, so it’s not a bad idea to be careful when placing items on the Block Diagram.

6. If it isn’t already, get the Random Number wired into the Multiply function. We’re not done yet, though, because Random Number only generates numbers between 0 and 1, so we’re going to multiply its output by 100. To do this we’ll use a Numeric Constant. You can either place one by opening the Numeric menu again, or you can right click on the free wiring node on the Multiply function and selecting Create → Constant. Give the Constant a value of 100.

7. Now place both a Space Constant (a special Constant found in the String menu) and another String Constant with a simple period inside.

8. In order to get a sentence from all of these separate components, we’ll use the Concatenate Function from the String Menu. Expand the Concatenate Function until you have 6 wiring nodes (you can do this by pulling the top or bottom of the function).

9. Once you have six wiring nodes on the Concatenate function, wire the outputs from your Controls and Constants into the inputs on Concatenate, being careful to get the order right with the first item in the sentence on the top and the last item on the
bottom. Be careful not to leave any wiring nodes unconnected; many functions, such as Concatenate, require each of their inputs to be wired before they will work. You will notice that if you wire the output from Multiply directly into Concatenate you will get a broken wire. That’s because Concatenate can only deal with String data (i.e. purple wires) and Multiply outputs Numeric data (in this case an orange wire). In order to get the VI to work you will need to get a conversion function from a submenu of the String Menu, for this example the Numeric to Decimal Integer String function should suffice. Now you can wire the output from Multiply into this new conversion function and then wire the output from the converter to Concatenate.

10. The output from Concatenate goes directly into the Generated Sentence Indicator and your VI should be ready to run.

11. Enter something into the two String Controls and run the VI. If the run button shows a broken arrow then something is probably improperly wired. Go ahead and press the run button and LabVIEW will give a sometimes useful explanation of what is keeping your program from running. If you get totally stumped then ask for help; even experienced users often have trouble debugging programs.

12. Once you’ve gotten the VI to run you probably saw a sentence similar in nature to the one in the example. If the order of words or phrases is incorrect then there is probably an error in the wiring. If too many spaces appear then extra ones were entered either into the Control or in the String Constants.

13. This VI, being fairly low level and lacking any automated string manipulation other than Concatenate, will display exactly what is entered in the Controls. To see this in action, go to either Proper Noun or Objects and put in several extra spaces. Run the VI and notice the clunky output. LabVIEW is not, generally speaking, overly useful when it comes to text editing, but manipulating strings can be useful in many VIs.

14. Save the VI as Sentence Generator. It’s probably a good idea for every team member to have a copy of the VI to use as reference.

That concludes Example 1: Sentence Generator. Now that you have some experience with placing functions and getting them wired, we’ll move on to Example 2: Iterations to Average, which uses a few more Numeric functions and a While Loop Structure.

Notes: If you want to make your VI a bit more robust you can add something called the Trim Whitespace VI, which will remove all spaces, tabs, etc from the beginning and end of a string.
Trim Whitespace is in fact a subVI, and if you place it on the Block Diagram you can double click on it to see its Front Panel and Block Diagram (only the Front Panel will be visible immediately, but you can see the Block Diagram by pressing Ctrl+E). It’s not very complicated, as you can see from opening it, yet it saves space and unnecessary, repetitive programming.

This may seem like it solves the problem of having a user enter too many spaces, but remember, this subVI only trims the beginning and end of a string. What if the Proper Noun was Benjamin Franklin and the user put in several extra spaces after the first name? While it is certainly possible to make a VI that can eliminate all extra spaces within a string no matter where they occur, it’s not necessary for the purposes of this class.

It’s also worth noting that the Random Number function does not produce integers, it produces floating point numbers (numbers that have a component after the decimal point) but in the string output you only see whole numbers. That is because the VI uses the Number to Decimal String converter, which automatically rounds off the input to generate a whole number. For this application that makes sense, but if you were taking voltage measurements in the 0-1 range, rounding off to whole numbers would essentially ruin your data. Always be careful that any conversions you perform do not cause you to lose data.

Example 2: Random Number Average

This VI will keep a running tally of the average of a series of random numbers and automatically stop once the average falls within a limit selected by the user. It introduces a few important concepts in LabVIEW: Structures, Booleans, Delay and Shift Registers. The completed VI for this example should look something like that shown in Figures 3 and 4.

![Figure 3. Front panel of the finished Random Number Average VI.](image)
Steps to create the Random Number Average VI:

1. Start a new, blank VI. On the Front Panel place one Numeric Control, named Acceptable Error, and two Numeric Indicators, one named Iterations and the other named Average. You will also need a Stop Button, which can be found in the Boolean menu.

2. On the Block Diagram you will need to place a While Loop. You will find this in the Structures menu. Click in the upper left corner of the Block Diagram to start the area that will be contained by the While Loop and click again once you get most of the visible
Block Diagram inside the loop. Don’t worry if you end up needing more room later; Structures can be expanded and contracted to suit your needs.

3. You will see two items immediately inside the While Loop as soon as it’s placed, one is the iteration counter and the other is the stop input. The iteration counter keeps track of how many times the While Loop has executed, with the convention that the first iteration is number 0 rather than number 1. The stop input tells the VI when to stop running the While Loop. The default setting on the stop input is for it to stop as soon as it receives True; that can be changed simply by clicking on it, which changes its icon to a circle with an arrow.

4. From the Numeric menu, find and place the following functions: Random Number, Increment, Multiply, Add, Divide, Add, Within Range and Subtract. Don’t worry about wiring them for now.

5. From the Boolean menu, find and place the Or function.

6. Right click on the right border of the While loop and select Add Shift Register. Shift Registers are useful ways to get data from one iteration of a loop to the next. Until a data type is wired into it, the Shift Register is black, but its color will change to whatever kind of data it is given. Whatever input is received by the right Shift Register on a given iteration is outputted from the left Shift Register in the next iteration. You will see in the example that a Constant with a value of 0 is wired into the left shift register; this initializes the Shift Register so that the first output it gives is always zero.

7. With those items placed, you are ready to begin wiring. You can see in the example that many wires go over others, and things can start to look a bit messy. Sometimes keeping things spread out can help, other times having too many long wires can make it hard to see where each one goes. One thing to always avoid is placing objects over wires, which can give the appearance that things are connected when in fact they aren’t. When in doubt you can simply follow as exactly as possible the example, although that will not be possible for the homework and later projects. This example will not include wiring instructions for the entire VI, but a few things of importance will be pointed out. First, be sure to divide the sum of the random numbers by the number of iterations and not vice versa, which would result in the inverse of the running average. Second, be sure to wire the Within Range function correctly, the top node should be a higher value than the lower node or the VI will not work correctly (it will still run, but since no values would satisfy its criteria, it would never stop).
8. After everything is wired up, give the Acceptable Error a value of 10 and run the VI. If it doesn’t stop within thirty seconds, push the stop button and try again. If it still runs for more than thirty seconds, you may have to do some debugging; make sure all the wires are connected correctly and try again. If all else fails try using the Highlight Execution option to see the values get passed along the wires, sometimes this can illuminate otherwise mysterious problems. If everything worked then you can try running the VI a few more times. If your Acceptable Error is fairly large then the VI will generally run less than 10 or so iterations and they should all happen fairly quickly.

9. Now turn on the Task Manager for the computer you are working on and watch the CPU usage when the Acceptable Error is 0. This value in and of itself has no effect other than to keep the VI running since it would take an astounding number of iterations for the average to hit 50.0000. You can see that when the VI is running the CPU usage is rather high, probably in the 50% range if you have no other programs running. That is because LabVIEW is executing the VI as fast as the computer will let it. In order to slow down the execution of the while loop, and thus lower the VI’s impact on the CPU, open the Timing menu on the Functions palette and place a Time Delay inside the while loop. When the dialogue appears for the desired delay, simply accept the default value of 1 second. Now run the VI and watch the CPU usage. Hopefully you saw a drastic reduction in the amount of CPU being spent to calculate the average of a stream of random numbers.

10. Lower the Time Delay to 0.01 seconds by double clicking on it and changing its value, then give the Acceptable Error a value of 10 and run the VI a few times. You’ll notice that it doesn’t take very long for an answer to appear most of the time. Lower the Acceptable Error to 1 and run the VI again. Now it takes a bit longer and during that time you don’t know what’s going on within the VI because it only displays results when it’s completely finished; while it’s running there’s nothing to see. You’ll change that in the next step.

11. Add two Numeric Indicators to the Front Panel, one labeled Current Iteration, and the other labeled Current Average. Wire their functions so that they will receive, and thus display, the current average and current iteration number respectively. Give the Time Delay a value of 1 second and run the VI. Now you can see how many times the loop has been repeated and where the average value is as the VI is running.

12. Add a control for the Time Delay (the easiest way to do so is simply right clicking on the input wiring node and selecting Create → Control, which will automatically name the
Control) and give it a value of 1. Now make the Acceptable Error 0 and run the VI. While the VI is running, change the value of the Time Delay using the Control on the Front Panel; you can see the VI running faster and slower as you alter the delay. Now try to change the Acceptable Error so that the Current Average falls within that range. If you have your Block Diagram wired so that your Acceptable Error Control is outside of the While Loop, which is how it is done in the example, then you will be unable to stop the VI by changing the Acceptable Error. This highlights something to keep in mind while you program VIs of your own: anything outside of a While Loop is essentially inactive as the loop executes. The Time Delay, on the other hand, receives input from its corresponding Control every time the loop executes because it is inside the While Loop.

The modifications discussed in the last few steps are illustrated in Figures 5 and 6. Save your VI as Random Number Average but don’t close it yet; you will be using it in the next example.

**Notes:** In this VI you can clearly see that some wires that pass numbers along are blue while others are orange. Why is that? Simply put, the blue wires are integers and the orange wires are floating point numbers. If you mouse over a wire with the Context Help window open, it will tell you what kind of data that wire handles. In the next example you will see bold lined wires, which are arrays. Paying attention to what kind of wires you are working with can save you many headaches; many functions only accept certain kinds of wires and will not function when connected to the wrong kind.

The While Loop is a powerful programming tool, but it requires some care on the user’s part. A common error for beginning users of LabVIEW is putting While Loops inside of While Loops and ending up with a VI that never executes. LabVIEW structures always wait for all the components they contain to finish executing before they themselves execute. So if you place a While Loop inside of a structure, such as another While Loop, then so long as the inner While Loop continues to iterate the outer structure will sit and wait.

This is also a good time to point out the difference between Stop and Abort Execution. If you turn on Highlight Execution and run the VI you can quickly see how both operations work. Pushing the Stop button will cause your While Loop to finish up the current iteration and then halt the VI. Abort Execution, however, halts the VI immediately. Both of these have their uses, although Stop is generally the preferred method for most programmers because it can be used as a trigger for other operations, such as powering down equipment.
Figure 5. Front panel of the updated Random Number Average VI.

Figure 6. Block diagram for the updated Random Number Average VI.
Example 3: Random Number Average with Displays

In this example you will see the difference between two Indicators, Waveform Graphs and Waveform Charts. Seeing a visual representation of your data history can be very helpful; large spikes in data that should be fairly smooth can alert you to faulty equipment or incorrect experiment setup, periodic fluctuations can quickly point out a time dependent relationship, an asymptotic decay is much easier to see as a line than as a succession of numbers, etc. Being able to incorporate graphs and/or charts into your VIs will prove a valuable asset in your data gathering. Figures 7 and 8 show the final front panel and block diagram for this example.

![Waveform Chart](image1)

![Waveform Graph](image2)

Figure 7. Front panel for Random Number Average with Displays.
Steps to create the Random Number Average with Displays VI:

1. With Random Number Average open, place both a Waveform Chart and a Waveform Graph indicator on the Front Panel. To pass data to the Chart you will have to bundle it using the Bundle function from the Cluster & Variables menu. Wire the random number and the average into the Bundle and then pass its output to the Chart function. For the Graph you will use the Build Array function from the Array menu, stretch it so that it has

Figure 8. Block diagram for Random Number Average with Displays.
two inputs, wire the random number and the average number into it and pass its output to the Graph function. Try running the VI a few times using an acceptable error low enough to require roughly 10 iterations or more. Two things you can hopefully see from this are, firstly, that the chart display remembers all the data it has been given and, secondly, that the graph retains no data at all. While this may lead you to initially assume that the chart display is therefore more useful, keep in mind that you may not always want to see all that data that you have gathered with a given VI since it was last started up.

2. Now you will modify how the graph receives its data by giving it arrays of data points rather than a stream of numbers. If you are completely unfamiliar with arrays, you can refer to the notes at the end of this example, or you can simply plow through and learn by doing. From the Functions Palette, find and place two more Build Array functions and stretch them to have two inputs. Now add two shift registers to the while loop. Wire the Build Array functions so that two have their outputs go to one shift register apiece and both of them feed into the original Build Array (the one which outputs to the graph function). Now wire the first two so that one gets its top input from the random number X 100 while the other’s top input is the averaging operation. The other input for both should come from the output of their own respective shift registers. You’re VI should look similar to Figure 9 at this point, with the addition of the chart function.

3. Try running the VI a few more times. Hopefully your graph display looks significantly different from last time; you should be seeing the entire set of data points for each run of the VI. Notice that this is still different from the chart display, which retains data points between individual runs of the VI until its history buffer is full.

4. Now right click on the chart indicator, open its Properties and under Display Style choose Sweep. Change the History Length to 100 points and set the time delay to 0.1. Run the VI until the data reaches the right side of the display and notice the red line showing where new data is being written over old data. Go back to the Display Style and change it to Scope. Run the VI again until the data reaches the right side of the display and notice that instead of moving the display or overwriting the old data, the display simply starts over at the left side of a blank screen. These three styles of chart all have their own uses; it’s up to you to determine which best suits your particular application.
5. In order to save this VI as Random Number Average with Displays you should click on File and choose Save As... followed by the Open Additional Copy option. Name the copy Random Number Average with Displays.

6. If you recall from earlier you monitored the CPU usage of the VI using the Task Manager and, if you had only LabVIEW running at the time, you probably saw roughly 50%
capacity when the Time Delay was 0. If you repeat that procedure with the VI that includes displays, you should see CPU usage closer to the 100% mark. This is important to keep in mind; LabVIEW will commandeer all the resources it can unless you put in measures to stop it from doing to. If your computer seems to be performing other tasks slowly while a VI is running in the background, perhaps you should check its Time Delay.

You can now close both VIs and, when prompted to select a save option, choose Don’t Save. Saving changes at this point would make Random Number Average the same as Random Number Average with Displays

Notes: If you have never been exposed to arrays before, then it is important to point out that the explanation given here will only be completely applicable to their usage in LabVIEW. Arrays are fairly ubiquitous in computer science and LabVIEW is no exception.

Arrays are ordered arrangements of elements. The elements in most arrays are either numbers or strings, although Boolean arrays are not uncommon. Arrays are referred to as having dimensions, such that a 1-D array consists of a single column or row of elements and a 2-D array (i.e. a matrix) would have both rows and columns. Arrays can have more dimensions, but 2-D arrays are enough for this course. The elements in an array are indexed by their position such that the element in position 0,0 is the upper left element.

LabVIEW makes extensive use of arrays and learning to use them effectively will greatly increase your programming capabilities.

Example 4: Light Selector

For this portion of the lab, you will learn the basics of the Case Structure. If you are at all familiar with programming, you can think of the Case Structure as a kind of If/Then command. Actually, it is similar to “If/then – else if/then – … – else if/then ... endif”. You will be working with a few Boolean operators as well, which work in states of True/False. A final version of this example is illustrated in figures 10 and 11.s
1. From the Structures menu, find and place a Case Structure. The default configuration for this structure includes only two cases, True and False. Clicking on the arrows to the side of the current case changes the display to either the case before or after. In order to get another case, right click on the case name and select Add Case After or Add Case Before. Now rename the cases so that you have a 1, 2 and 3 for the case names. You will probably see the names of the cases appear in quotations; this will not change until a numeric data wire has been connected to the structure.

2. Place three Push Buttons from the Boolean menu onto the Front Panel. On the Block Diagram, put a While Loop around the three buttons’ functions. Change the names of
the controls to Top, Middle and Bottom. From the Boolean functions palette, select the Compound Arithmetic function, expand it to accept three inputs and wire all three buttons into it, then pass its output to the loop condition. Now open the Boolean functions menu and place three Boolean to (0,1) functions to change a True or a False to a 1 or a 0. Multiply the middle output by 2 and the bottom output by 3. Add all of the outputs together using another Compound Arithmetic function, this time from the Numeric menu, and pass that sum along to the Case Selector on the case structure.

3. Now, back on the Front Panel, place three Round LEDs from the Boolean menu. These will light if they are given an input of True. Change their names to Top Light, Middle Light and Bottom Light. On the Block Diagram, place their functions to the right of the Case Structure.

4. Inside the Case Structure, you will need three Boolean constants for each of the three cases, as you can see in Figure 1. For the 1 case, the top constant should be True, the other two should be False. Cases 2 and 3 you should be able to configure on your own.

5. Once everything is wired, execute the VI without pushing any of the buttons. It may appear that your VI is doing nothing, but in fact it is waiting for one of the buttons on the Front Panel to be pushed. Try out the three buttons to make sure that the VI works as intended. After you have seen that the VI works as intended save it as Light Selector.

This VI may seem like a complicated way to get a light to turn on, but it showcases how to use case structures in ways beyond the simple True/False scenario. Case structures can be useful for making VIs perform different sets of actions depending on measurable conditions or for allowing a user to select between different configurations for a given VI.

**Notes:** Making sure that a switch has the proper mechanical action can often be the difference between a frustrating VI and a fully functional one. For this VI, any mechanical action could be made to work, but the Latch When Pressed option is arguably the best. Other times you may want a button that lets you select between two possible modes of operation, in which case the Switch When Pressed option might be better. It’s up to you to determine the best mechanical action for your own VIs.