CHAPTER 4

Loops and Conditional Statements

An application cannot be complete without using ‘for’ or ‘while’ loops or conditional if and else statements. This may cause you to wonder how such statements can be achieved with a graphical language like LabVIEW. This chapter will show you how and will illustrate a variety of interesting new features that cannot be encountered in text-based languages.

There are seven items on the **Structures** sub-palette within the **Functions** palette that can be popped up by right clicking in the diagram window: **Sequence**, **Case**, **For Loop**, **While**

Loop, **Formula Node**, **Global Variable**, and **Local Variable**. Start with the **For Loop**.

4.1 For Loop

This loop iterates the number of times specified; however, if the total number iterations is zero or negative, it will not run. Note that the **For Loop** is available in the diagram window and not in the front panel. This is because the **For Loop** is related to a functionality of instruments, and not to data display or input switches. You can put a **For Loop** in the diagram window by first selecting it from **Functions** >> **Structures**. When the **For Loop** is chosen, the cursor changes to  . Dragging it in the diagram will create a **For Loop**, then place commands that are meant to repeat in the loop. Consider the following pseudo code:

```
for i = 1 to 10
     display i;
     pause 1 second;
end;
```

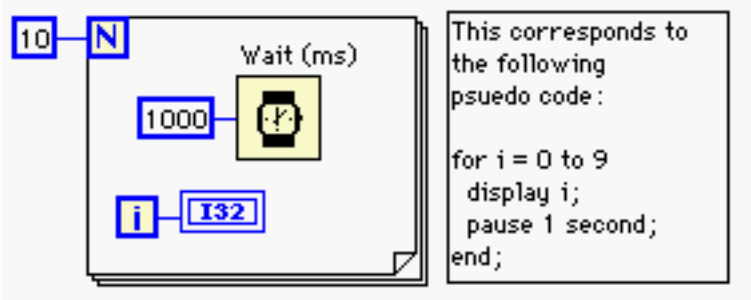

Figure 4.1 Conversion of conventional for loop into the **For Loop** of LabVIEW

The corresponding **For Loop** in LabVIEW is shown in Figure 4.1. The *count terminal*  is for the total number of iterations. The value entered must be a positive number in order for the **For Loop** to iterate. The *iteration terminal*  returns the current iteration index. In Figure 4.1, you have the **Numeric Constant** 10 wired to the count terminal. This will cause the **For Loop** to repeat 10 times and the index will vary from 0 (not 1) to 9. Like the C programming language, the indexing in LabVIEW always starts at zero. Now you will be introduced to an

important technique of creating input to LabVIEW VI terminals.

4.1.1 Creating inputs to terminals

The previous chapter discussed how to create sub VIs. It is true that almost everything in LabVIEW is a sub VI which is equivalent to saying that everything in C is a function. Also, each sub VI can call other sub VIs like each function can call other functions in C. When you call functions in C, you pass arguments in parentheses, whereas you pass arguments through wires in LabVIEW. When you pass arguments in C, you should match the *type* of the arguments such as float, integer, and so forth. If you want to force a certain data type to be a different type, you may use coercion or casting in C. LabVIEW also has coercion and casting features as well as data type conversion capability. In LabVIEW, you should create the correct type of variables or constants as input in order to avoid a type conflict. It is important to match the type because a type conflict causes an unnecessary use of memory.

When you created the diagram in Figure 4.1, you probably searched through sub-palettes looking for the **Numeric Constant** to wire 10 to the count terminal,  . It expects an I32 long integer type. Therefore, if you wire an input of a different type to it, a type conflict will result and cause LabVIEW to allocate a new memory address for type coercion. Recall one of the statements made previously:

If you are in doubt, right click on the object.

If you right click on the iteration input terminal, you will see three options available in the pop-up menu: **Create Constant**, **Create Control**, and **Create Indicator**. If you choose **Create Constant**, LabVIEW will automatically create a constant of correct type and even wire it for you! Of course, you can change the type of the constant manually. Right click on the constant 10 in Figure 4.1, and go to **Representation**. You will see the complete selection of available data types. Just for practice, change the data type to double precision DBL. After doing so, you will see that the color of the number 10 has changed from blue to orange. You will also see a gray dot on the iteration input terminal of the **For Loop**. This is called a *coercion dot* and it indicates the existence of a type conflict.

Figure 4.2 Coercion dot

Now, correctly wire an I32 data type to the iteration terminal  with the right-click method. The **Wait (ms)** can be found under the **Time & Dialog** sub-palette in the **Function** palette and its input type is an unsigned 32bit-integer (U32). To avoid a type conflict, right click on the input terminal of **Wait (ms)** and select **Create Constant** followed by entering 1000 for 1 second. This will provide us with enough time to see the increment of index. Go to the front panel and run the VI to view the increment of index from 0 to 9.

If you have more than one input to a VI and you want to automatically create inputs, select the Wire Tool and place it on the input of interest. (Its label will show up.) Now, bring up the pop-up menu by right-clicking and select an option from **Create Constant**, **Create Control**, or **Create Indicator**. This is an essential addition to the wiring technique discussed previously.

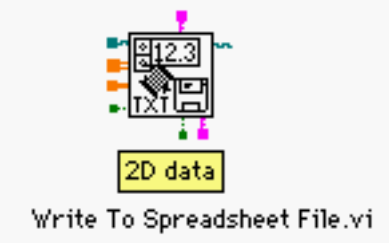

Figure 4.3 Creating inputs from the pop-up menu of the input terminals

An example of creating input terminals from the pop-up menu of the input terminals is shown in Figure 4.3. The VI used in the example is **Write To Spreadsheet File.vi** which can be found in **Functions** >> **File I/O**. Currently, the Wire Tool is placed on the second input **2D data** so that its label is being displayed. If you select **Create Control** or **Create Constant** from the pop-up menu, LabVIEW will automatically create a control or a constant of correct data type. Note that if a wire in Figure 4.3 has a square dot at the end of it, it means that the corresponding terminal expects a control or a constant as its input. If a wire does not have such a square dot at its end, the corresponding terminal expects an indicator as its input. Therefore, in the Figure 4.3,

the input **2D data** expects either a control or a constant as its input due to the square dot at the end of the wire.

> To create and wire correct inputs to input terminals of a VI, select the Wire Tool and locate the input of interest by monitoring the label of input. Bring up the pop-up menu and select either **Create Constant**, **Create Control**, or **Create Indicator**. If a wire has a square dot as shown in Figure 4.3, the corresponding terminal expects a control or a constant as its input. If not, the corresponding terminal expects an indicator as its input. If you want to manually change the input data type, right click on the input, go to **Representation**, and select the preferred type.

4.1.2 Color of different data type

You have just seen that blue is an integer data type and orange is a float type. This color differentiation allows you to easily identify the data type in much the same way that you identify controls and indicators by the thickness of the borderline. The following is the complete list of colors and their corresponding data types: (Clusters are discussed in detail in Chapter 6 where arrays are studied.)

blue	integer
orange	float
green	Boolean
pink	string
brown/pink	cluster (blend of different types)

4.2 While Loop

The **While Loop** is a structure that repeats its body until a test condition fails. A **While Loop** can be placed in a diagram window in the same manner as the **For Loop**.

Consider the following pseudo code that will now be converted to LabVIEW G-language code:

```
x = 0;
While (x < 10)
    display x;
    pause 1 second;
```

```
        x = x + 1;
end;
```

In this example, the condition (x < 10) is tested *before* the loop starts. So, if x was initialized with 10, the **While Loop** body will never get executed. However, since LabVIEW tests the condition *after* starting the loop, the **While Loop** *always* executes at least once. (This is similar to the DO … WHILE statement.)

> The **While Loop** in LabVIEW tests the condition after starting the loop; therefore, the **While Loop** always executes at least once.

This is an important notion to recognize so you can avoid damaging data acquisition systems with **While Loop**. For example, suppose that your VI application acquires voltage signals from a system that generates high voltage if the system breaks down. In this instance, your application uses a **While Loop** to acquire a voltage signal as long as the signal remains under a certain threshold. The threshold exists in order to avoid any damage to the data acquisition system. Unfortunately, the problem here is that since the LabVIEW **While Loop** tests the condition *after* starting the loop, the first run will completely damage your system if it was already malfunctioning. After running the system after the first run of the **While Loop**, it stops! This can be avoided by testing the voltage independently without passing it to the main data acquisition system *before* starting the **While Loop**. Therefore, you should keep this important difference in mind whenever you attempt to use the **While Loop** in LabVIEW. On the other hand, the **For Loop** tests the condition if the input to the iteration input terminal is positive *before* starting the loop.

> The **For Loop** in LabVIEW tests the condition to the iteration input terminal before starting the loop.

Figure 4.4 illustrates a LabVIEW **While Loop** that corresponds to the pseudo code above, displaying **x** while incrementing it until **x** is greater than or equal to 10. The comparison LabVIEW function **Less?** can be found in the **Comparison** sub-palette within the **Functions** palette and it returns a true or false Boolean value. (You can also identify the output type by the green colored wire.) When you create the VI in Figure 4.4, remember to use the automatic input

creation feature of LabVIEW by right clicking on the input, and selecting either **Create Constant**, **Create Control**, or **Create Indicator**. In this example, selecting Create Indicator after right clicking on the index terminal created the indicator. Actually, the last value of **x**, which is 10, is the same in both the pseudo code and the diagram window. However, LabVIEW will display the last value whereas the pseudo code will display only up to 9 since the while loop in the pseudo code will exit when the test fails.

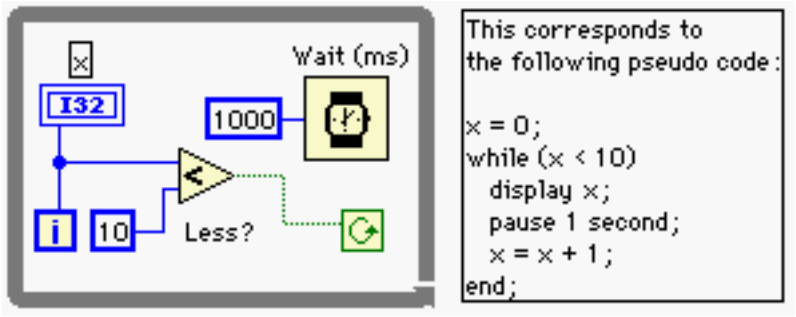

Figure 4.4 Conversion of conventional while loop into LabVIEW **While Loop**. The last value of **x** is the same in both, but LabVIEW will display the last value whereas the pseudo code will display only up to 9.

Figure 4.4 introduces a new terminal that is called *conditional terminal*. It expects a Boolean input since its color is green, and the **While Loop** will repeat until the input to the conditional terminal becomes false. The iteration terminal and **Wait (ms)** have been explained in the **For Loop** discussion section. After you complete the diagram shown in Figure 4.4, go to the front panel and run it to see the increment of **x**. This completes the discussion of the **While Loop**.

4.3 Case Structure

The **Case** structure in LabVIEW is analogous to the If and Else statement in text-based programming languages. However, realization of nested If and Else statements in LabVIEW requires more attention since LabVIEW provides graphical conditional selection in a two dimensional diagram window. The nested If and Else statements can be realized by nested **Case** structure or multiple frame **Case** structure. You can also specify a range for each case. The example below shows a pseudo code and its conversion to a proper LabVIEW code with **Case**

structure. Consider the following If and Else pseudo code:

```
if (x < y)
     z = –1;
else
     z = 1;
end;
```

In this simple case, you have two options where only one of them will be executed based on the condition test; therefore, you need two frames for the **Case** structure as shown in Figure 4.5.

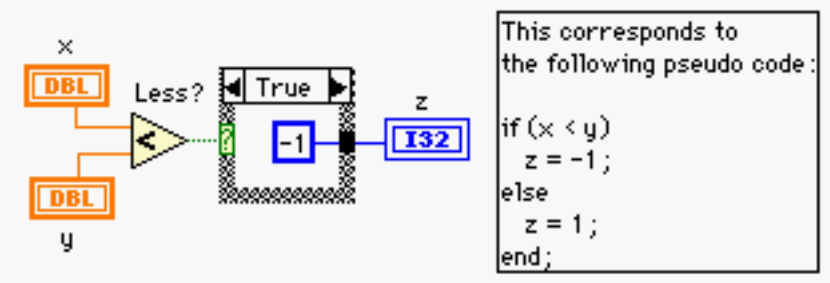

Figure 4.5 Conversion of conventional If/Else statement into LabVIEW **Case** structure

When you place a **Case** structure in the diagram window, it comes with True and False frames that you can toggle by clicking the arrow next to the label. When you finish the diagram shown in Figure 4.5, you will realize that the run button is broken ( : Broken Run Arrow). This indicates that the VI is not complete and LabVIEW cannot compile it. It is because the indicator **z** is expecting a value based on two conditions (either **x** is greater than **y** or not), so an output for the False case also must be defined as illustrated in Figure 4.6.

Figure 4.6 Completing **Case** structure by specifying an output for each case

If you go to the other False frame and wire a **Numeric Constant** 1 to the white square *tunnel*, it

will turn black, and now the VI is complete. It is important to specify an output (if any) for both cases in **Case** structure as most errors in the use of **Case** structure result from unspecified output.

> In **Case** structure, you must specify outputs for all of the frames if there is any output at all. Any missing output at the tunnel can be detected by the presence of a white tunnel. However, this rule does not apply to input (data *entering* the **Case** structure).

If you complete the False frame as in Figure 4.6, the Broken Run Arrow should become a **Run** button  indicating that the VI is ready to execute. Go to the front panel, enter some value in the controls **x** and **y**, and observe the value in the indicator **z**.

4.4 Case Structure with Multiple Frames

In the previous section, the **Case** structure had True and False frames with Boolean input to select either one. However, the **Case** structure will also accept numeric values and string inputs to allow the user to select from more than two frames. Examples will be used to illustrate each case.

4.4.1 Case structure with numeric input control

Figure 4.7 uses both G-language and text-based language to illustrate an example of the **Case** structure with more than two frames. You can specify the range by using two period symbols (..) in the label of the frame. (You can change the label using the Text Editor.) Also, you can specify a default frame from the pop-up menu of any frame. Note that you must have all of the frames cover all of the possible ranges. Failing to do so will result in the Broken Run Arrow.

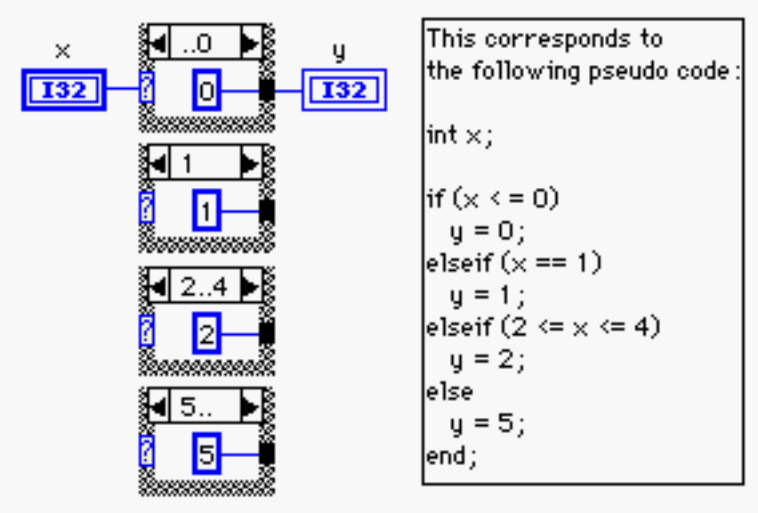

Figure 4.7 Example of the **Case** structure with more than two frames. Numeric value is used to select a frame. This figure shows a single **Case** structure with its frames shown together.

4.4.2 Case structure with string input control

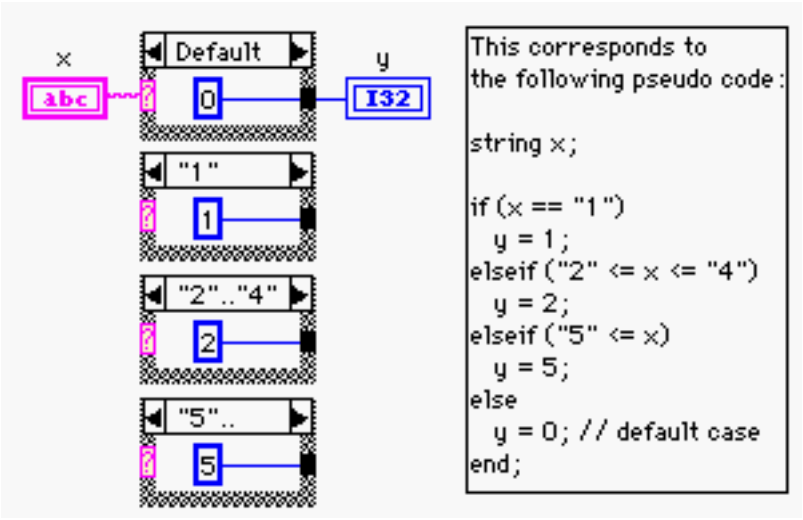

Figure 4.8 Example of the **Case** structure with more than two frames. String input is used to select a frame. This example performs the same task as Figure 4.7 does except that input values are now in string format instead of numeric format. This figure shows a single **Case** structure with its frames shown together.

The **Case** structure can also take string variables as its input to select frames. An example is shown in Figure 4.8.This example performs the same task as that in Figure 4.7, but only their

frame labels are different. As the labels indicate, each frame is chosen by a string input instead of a numeric value. Note that you can still specify a range of numbers in string format using two period symbols (**..**). Also, note the difference in the label of the first frame such that you do not have the label **.."0"** for the first frame in Figure 4.8, whereas in Figure 4.7, you have two possible labels for the first frame. If you use the label **.."0"** for the first frame with string input control, LabVIEW will return the Broken Run Arrow. This is because the input is string type and the string range **.."0"** does not cover the rest of the range, thus violating the requirement in the coverage by labels. However, by labeling the first frame Default, the entire range is now covered and LabVIEW will be able to execute the VI.

4.5 Sequence Structure

In text-based programming languages, the top-to-bottom flow of execution is easily understood. However, LabVIEW must specify the order of execution in a two-dimensional diagram window. In LabVIEW, the execution order can be specified through wires or the **Sequence** structure. The first method is based on the simple mechanism such that if a control **x** is wired to an indicator **y**, **x** will be executed first while generating a value that will subsequently be passed to **y**. Therefore, the order of execution is **x** followed by **y** which makes the direction of data flow, from **x** to **y**.

Figure 4.9 Specifying execution order through wires. Control **x** will be executed prior to the execution of **y** and the **While Loop**.

Figure 4.9, illustrates an example of using wires in LabVIEW to specify the execution order. Note the wire from the control **x** to the **While Loop**. Even though the **While Loop** is not using the value coming through the wire from **x**, this is a legitimate way of programming in LabVIEW to specify the order of execution. You can also apply this rule to other elements in LabVIEW, including the **For Loop**, **Case** structure, and **Sequence** structure. Therefore, the order

of execution in Figure 4.9 is **x** first followed by the **While Loop** and **y**, since the order between **y** and the **While Loop** cannot be defined. (Actually, the *content* in the **While Loop** starts after the value in **x** is written to **y** since the value in **x** reaches the *boundary* of the **While Loop**, not its content. To see such a behavior, try the light bulb debugging tool  to see the signal flow. Therefore, the accurate order of execution is **x** -> **y** -> **While Loop**.) The second method (specifying the execution order using the **Sequence** structure) could be preferred by some users since the **Sequence** structure explicitly indicates the order of execution.

You can consider the **Sequence** structure as a collection of movie frames. When you play a movie, the player will project the images sequentially from frame 0, 1, 2, and so on. You can put a **Sequence** structure in the diagram window in the same way you create a **For Loop** or a **While Loop**. When you place it in the diagram window it will come with only one frame, but you can add more by right clicking on the border and selecting **Add Frame After** or **Add Frame Before**. If you add two more frames, for example, you will have frame numbers 0, 1, and 2 as shown in Figure 4.10.

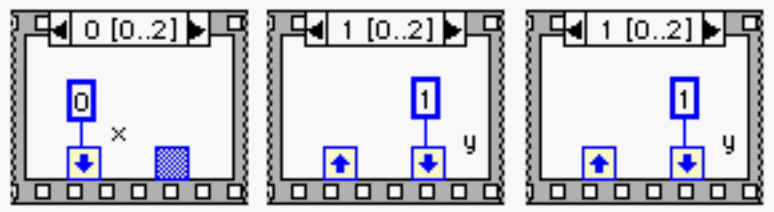

Figure 4.10 **Sequence Local** of **Sequence** structure. **x** is created in frame 0 so that it is available in all of the subsequent frames; however, **y** is created in frame 1 so that it is not available in frame 0, but is so in frame 2. Grey **Sequence Local** indicates its unavailability.

In Figure 4.10, **Sequence Local** is introduced, and is a channel through which parameters created in a certain frame can be made available to subsequent frames. Right clicking on the border of the frame and selecting **Add Sequence Local** will create it. As shown in the figure, the data you generate in a certain frame will not be available in the previous frames. (The corresponding **Sequence Local** is grayed out.) And, the order of execution is frame 0, frame 1, and so forth. Note that the **Sequence** structure should be used with discretion since manually specifying the execution order by the **Sequence** structure can degrade the speed performance of your VI.

4.6 Global Variable and Local Variable

GLOB LOCAL First, start with **Local Variable**s. In C/C++ code, everything you have in each function is local unless it is declared as global. Therefore, the default in C is local. In LabVIEW, the same is true because once you close your VI, all of the variables in that VI become unavailable until you open it again. Therefore, you may ask why LabVIEW has **Local Variable**s if everything is already local. This is because a **Local Variable** in LabVIEW has a slightly different meaning than conventional local variables mean in the text-based languages. Consider the following example.

Suppose you have a VI with a lot of variables (controls and indicators) connected through complicated wires. Naturally, you will use a larger diagram window to see more objects on the screen. Suppose you have an object **x** at one corner of your screen and you need to access it at the other end, but between them are a multitude of wires and objects. Of course, you could wander through objects and wires to make your connection, but this is not a good way of programming. One solution would be to use a **Local Variable** of **x** which is a copy of **x** to access its value or even to update its value. In other words, you can read or write from or to **Local Variable**s**.** This implies that you are no longer limited to the rule that controls can only generate (write) values and indicators can only receive (display) values!

> **Local Variable**s of an object (either a control or an indicator) are identical copies of the original objects. Using **Local Variable**s in LabVIEW, you can read from indicators or write to controls. To create one, right click on the object, and choose **Create** >> **Local Variable**, or choose **Functions** >> **Structures** >> **Local Variable**, and select a corresponding object from its pop-up menu.

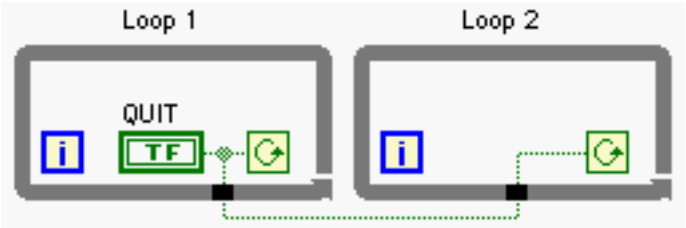

Figure 4.11 Controlling two **While Loops** without using **Local Variable**s

Consider a few examples where the use of **Local Variable**s is inevitable. In Figure 4.11, you are trying to stop two **While Loops** using the **QUIT** Boolean control. When you toggle the Boolean switch **QUIT** to generate False, what would happen to both **While Loop**s? There will be no problem in controlling Loop 1 because it will stop when **QUIT** generates False. However, Loop 2 will never stop because of the inherent characteristics of the **While Loop.** That is, once it starts, nothing goes in or comes out until the loop stops. Once Loop 2 starts, no other external value can get in the loop so that the conditional terminal in Loop 2 will keep receiving True forever! Let us look at an alternative as shown in Figure 4.12. If you use a **Local Variable** of **QUIT** inside Loop 2, you can stop the second loop when the first one stops because the **Local Variable** of **QUIT** will reflect the action of the original Boolean switch **QUIT**.

Figure 4.12 Controlling two **While Loop**s using **Local Variable**s

Another good example of **Local Variable** is incrementing a variable repeatedly. Suppose you want to perform the following:

```
k = k + 1;
```

Note that k on the left side is an indicator since it is receiving a value, but k on the right side is a control since it is generating a value. Note that you cannot use one object for both control and indicator without using a **Local Variable**. Therefore, beyond this point there is no such restriction that controls can only write and indicators can only display. In order to perform k = k + 1, you can use a **Local Variable** for either one of two k's. Changing the **Local Variable**s to indicators or controls is also easy. Right click on the **Local Variable** and choose either, **Change to Read Local,** or **Change to Write Local**. Now, look at the borderline; if it is thick, it is a control, and if it is thin, it is an indicator. The last example would be a **Sequence** structure with

multiple frames. If you want to have any parameters that you create in one of those frames available outside the **Sequence** structure *before* reaching the last frames, a **Local Variable** of them can be used.

As for **Global Variable**s, the concept is identical as in the conventional programming. For example, **Global Variable**s are those that can be accessed from different functions in C. In LabVIEW, **Global Variable**s are those that can be accessed from different VIs. Since they are defined outside VIs, they look just like a VI except for the fact that they do not have a diagram window. To create a **Global Variable**, choose **Global Variable** under the **Structures** sub-palette in the **Functions** palette. Place it in the diagram window and double click on the icon; this will bring up the front panel of the **Global Variable** without a diagram. Put any objects you would like to use as **Global Variable**s and save the front panel. The file name of the VI you just saved will be the name of **Global Variable** that can be used for the objects it contains. Since the use of **Global Variable**s can cause a great deal of confusion, readers should use them with discretion.

An example of the use of a **Global Variable** is as follows: Suppose you have multiple VIs running independently but they share the same data source. In other words, suppose you have the two VIs **APP1.vi** and **APP2.vi** running independently where **APP1.vi** acquires data and saves it to a **Global Variable** (update a **Global Variable**), and **APP2.vi** retrieves the data from the **Global Variable** for processing. As you can see, the timing must carefully be managed in order to process the correct data. If the update of a **Global Variable** occurs too fast for the second VI to keep up with, the second VI will miss the data acquired by the first VI.

4.7 Formula Node

Even though LabVIEW itself provides a full functionality for any possible arithmetic expression, there may be situations where you prefer to use equations by typing in the formula. The **Formula Node** is for such purposes. The most useful and common situations where the **Formula Node** is used would be the linearization process of thermocouple, Resistance

Temperature Detector (RTD), or strain gauge readings. In these processes, equations are nested versions of the same expression with different coefficients, and recursive calls cannot be used since LabVIEW does not support recursive calls. Figure 4.13 shows a simple example of how to use the **Formula Node**.

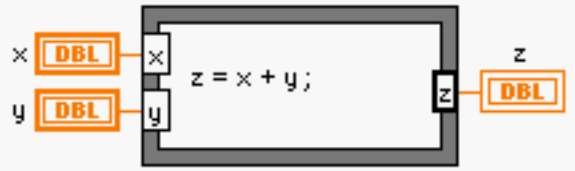

Figure 4.13 Example of **Formula Node**

First, put a **Formula Node** from **Functions** >> **Structures** in the diagram window. Second, right click on the boundary and select **Add Input** or **Add Output**. Conventionally, inputs are put on the left side and outputs on the right side. One thing to remember is that you have to label each input and output as shown in Figure 4.13. Then, use the Text Editor to enter the equation in the box. The variety of expressions is limited, but most trigonometric functions and logical expressions are supported. For the complete list of expressions that are supported by the **Formula Node**, you are advised to refer to the LabVIEW Users Manual.

4.8 Auto-indexing and Shift Register

One of the unique features of LabVIEW is *auto-indexing* and *shift registers* with loops. It is natural to consider indexing elements and shift registers in **For Loop** or **While Loop** because you repeat the same operations. First, consider auto-indexing.

4.8.1 Auto-indexing

Suppose you are acquiring a single temperature reading in a loop (either a **For Loop** or a **While Loop**) at each iteration, and you want to collect each reading in an array. Or, you already have an array, and want to parse each element out at each iteration. LabVIEW allows you to do both by enabling or disabling the auto-indexing feature at the boundary of the loop. Figure 4.14. shows

the **For Loop**s, Loop 1 and Loop 2. In Loop 1, a single random number is generated at each iteration, and it repeats 10 times. The two lower wires leaving Loop 1 are using auto-indexing, and they are thicker indicating the increase of dimension to a 1-D array of ten elements from a single number. In Loop 2, a 1-D array enters and each element gets parsed out due to auto-indexing. This explains auto-indexing for the data entering the loop.

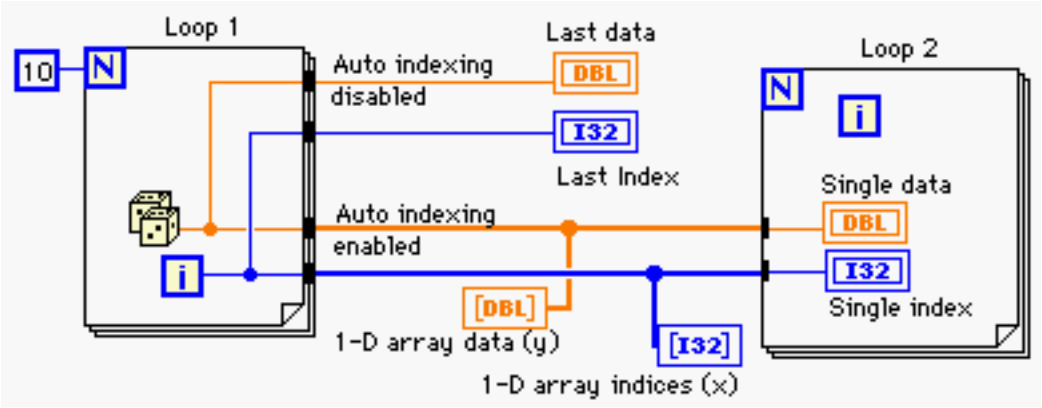

Figure 4.14 Auto-indexing with **For Loop**s

However, if you disable the indexing by right clicking on the tunnel and selecting **Disable Indexing**, the data leaving Loop 1 will be the *last* element generated, which will be the 10th element since the total number of iterations is 10 in Figure 4.14. The same thickness of the wires indicates that the dimension of the data has not been increased. Also, note that you do not have any input to the count terminal, N, of Loop 2, but it will still iterate 10 times. This is another unique feature of the **For Loop** of LabVIEW. However, this is true only if the indexing is enabled. If you disable the indexing, the **For Loop** will not be able to know how many elements there are in the input array. Therefore, you must provide an input to the count terminal to specify the total number of iterations. Auto-indexing is useful feature in array manipulation, and arrays will further be discussed in Chapter 6.

> If a 1-D array is wired to the border of the **For Loop**, it will automatically iterate *N* times, where *N* is the size of the array, and it will ignore any number connected to the count terminal. However, if the indexing is disabled, you must wire an input to the count terminal, N.

4.8.2 Shift register

While dealing with loops, you may have situations where the tracking of previous data elements is needed. LabVIEW provides shift registers to accomplish such tasks. You can create shift registers by right clicking on the border of either **For Loop** or **While Loop**, and selecting **Add Shift Register**. This will create a pair of shift registers at both the left and the right borders of the loop, with a down arrow on the left and an up arrow on the right. The left one can be dragged down to include more than one element. You can also increase the number of shift registers on the left side by right clicking on the left shift register and selecting **Add Element**. An example is shown in Figure 4.15.

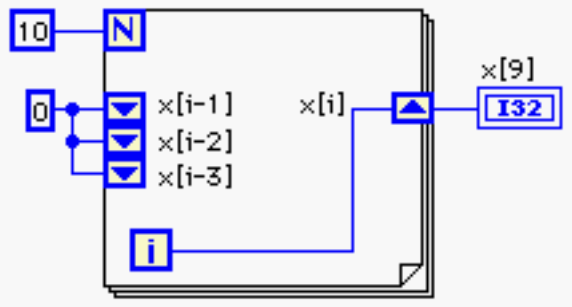

Figure 4.15 Shift register with 3 elements

Figure 4.15 shows a **For Loop** with a shift register which has three elements. $x[i]$ represents the data leaving at the ith iteration. At the next iteration, $x[i]$ will show up at the top element on the left side. At the following iteration, the top element will be pushed down to the second and the content in the second will be pushed down to the third, and this process will continue. Therefore, you will have a lag of 1 iteration between each element: the one leaving on the right side is the current one, and the three on the left are the previous values of 1 lag, 2 lags, and 3 lags. This feature can be used for moving average calculation for thermocouple signals or some noisy signals. If you average the three shift register elements on the left, it will become a moving average of three data elements. The moving average technique is studied in Chapter 14 again.

Lastly, it is recommended that all of the elements of the shift registers on the left side be initialized with an initial value. In Figure 4.15, you have initialized three elements with zero in

order to avoid any possible confusion, especially when the starting value is different from the default value of the data type wired to the shift registers. LabVIEW will not return the Broken Run Arrow even if no initial value is assigned to the shift registers, and you will not be able to realize the mistake until the **VI** returns wrong values. Therefore, it is a good idea to initialize the shift registers manually.

PROBLEMS

4.1 Create a sub VI **P04_01.vi** as shown in Figure P4.1.
(a) How many times will the **While Loop** iterate?
(b) What does this VI do? Hint: Run the VI with the same value for **x** twice.

4.2 Create a VI **P04_02.vi** that calls the sub VI **P04_01.vi** three times simultaneously. The diagram window and the front panel are shown in Figure P4.2.
(a) Based on the answer in Problem 4.1 (b), what are the expected values in **y1**, **y2**, and **y3** at each iteration? (Note that the total number of iterations is 5.)
(b) Run the VI **P04_02.vi** with the **Highlight Execution** light bulb on in the diagram window, and observe the three output values at each **P04_01.vi**. Why do the values displayed in **y1**, **y2**, and **y3** not agree to the answers in part (a)?

4.3 Select the **Reentrant Execution** option for **P04_01.vi** and run **P04_02.vi** that you created in Problem 4.2 with the **Highlight Execution** light bulb on in the diagram window. Do the values of **y1**, **y2**, and **y3** match the answers in Problem 4.2 (a) now? Explain why or why not.

4.4 Create a VI **P04_04.vi** and place the VIs shown in Figure P4.4. Using the Text Editor, or **String Constant**, list the path of each VI in the diagram window. Then, create controls and indicators to each VI using the technique mentioned in section 4.1.1. After creating them, make sure to arrange the controls and the indicators nicely in the front panel.

4.5 An array is a data structure that contains one or multiple elements of the same data type. Using the **For Loop** and auto-indexing feature, create the following array **x** of 21 elements in **P04_05.vi**:

$$\mathbf{x}[k] = e^{-\frac{k^2}{10}},\ k = -10, -9, \ldots, 9, 10$$

You will need to use **Exponential** from **Functions** >> **Numeric** >> **Logarithmic**.

4.6 Create a VI that turns on a LED when a random number generated by **Functions** >> **Numeric** >> **Random Number (0-1)** is greater than 0.5, and save it as **P04_06.vi**. Use **Function** >> **Time & Dialog** >> **Wait (ms)** to iterate the **While Loop** at the rate of 500 msec. The complete diagram window is shown in Figure P4.6.

4.7 Figure P4.1 shows how a VI can have memory using the **While Loop** without any Boolean control wired to the conditional terminal. This is because the **While Loop** always executes at least once such as `do while` statement in C++. In other words, the **While Loop** in LabVIEW checks the condition *after* executing one iteration. Keeping that in mind, consider Figure P4.7, which is the incomplete diagram window of the pseudo code listed in the next page. Find out why the VI needs to be modified, correct the error, and save it as **P04_07.vi**. (Repeat executing the VI to see the incompleteness of it.) When the VI stops, `k` should be 10, and `sum` should be 55 every time you run it.

```
k = 0; sum = 0;
while (k < 10){
      k = k + 1;
      sum = sum + k;
      display k and sum for 1 second;
}
```

4.8 Realize the VI in Problem 4.7 using a shift register instead of the **Local Variable**, and save the VI as **P04_08.vi**.

4.9 Realize the following pseudo code using the **Case** structure with multiple frames instead of nested case frames. Save the VI as **P04_09.vi**.

```
if (k = 0)
      display a string “Case 0”;
else if (k = 1)
      display a string “Case 1”;
else if (k = 2)
      display a string “Case 2”;
else
      display a string “Default case”;
```

4.10 Suppose that a data sample is streaming into the computer, and you want to average a certain number of samples continuously. This averaging scheme is often said to be a *moving average*, and is used to smooth noisy input data such as temperature or strain gauge readings. Create a VI **P04_10.vi** that performs the moving average using a shift register with 4 samples: current sample and three previous samples. Use **Functions** >> **Numeric** >> **Random Number (0-1)** to generate a single data sample at each iteration in a **While Loop**. This is often said to be moving average of window size 4. See Figure P4.10 for details.

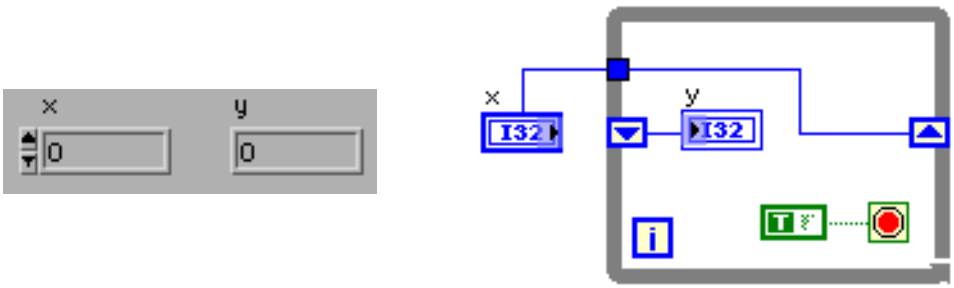

Figure P4.1

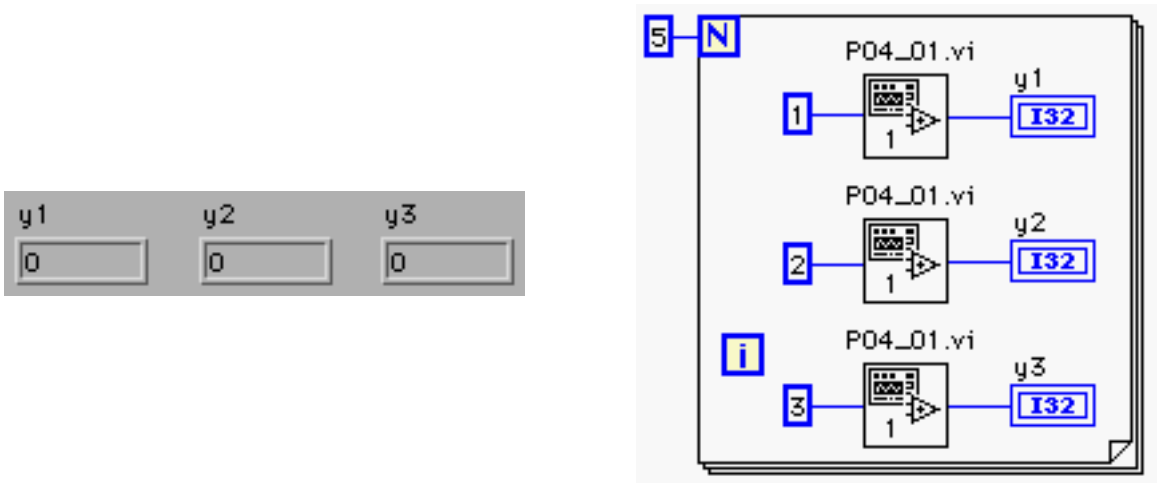

Figure P4.2

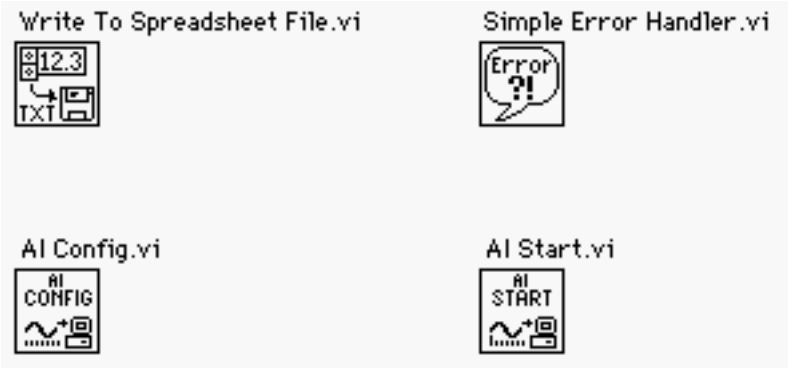

Figure P4.4

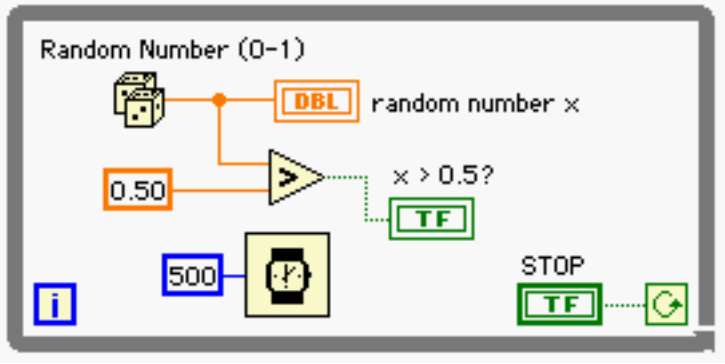

Figure P4.6

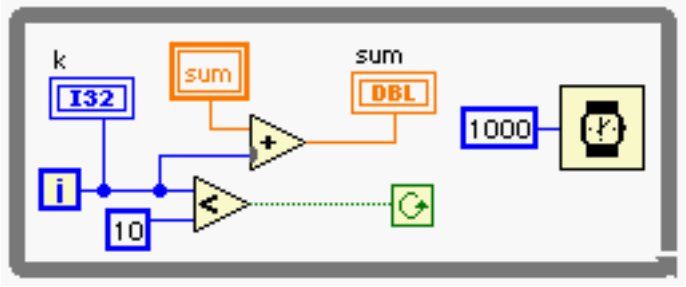

Figure P4.7

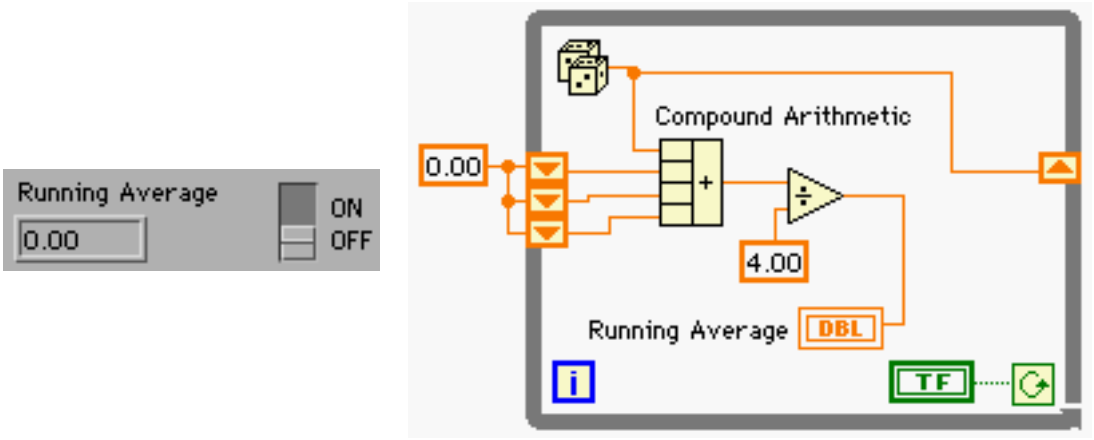

Figure P4.10