CASE STRUCTURES, ARRAYS, WAVEFORMS 2

ELECTRICAL ENGINEERING 20N
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

SIMON HONG, HSIN-I LIU, JONATHAN KOTKER, AND BABAK AYAZIFAR

1 Introduction

In this lab, we will explore another LabVIEW construct – the **case structure** – that allows different behaviors based on the states of certain blocks and variables. We will also explore how arrays are implemented and manipulated in LabVIEW.

In the most basic sense, an **array** is simply a sequence of data points. What makes these sequences interesting is that they can represent signals. A signal usually contains information that describes a natural phenomenon. For instance, sound can be represented as varying pressure waves, which in turn can be approximated by a sequence of numbers describing this physical phenomena. These sequences, or arrays, are fundamental in describing and dealing with signals, as we shall see in this lab. We will also play around with graphical methods of displaying the signals that these arrays represent, with the help of **waveforms**.

1.1 Lab Goals

- Manipulate case structures.
- Use arrays in a LabVIEW application, in association with other structures.
- Generate waveforms and display them on the front panel using charts and graphs.

1.2 Checkoff Points

2 Case Structures

2.1 Terminology

Until now, the block diagrams of the VIs we generated were fairly static: data moved from left to right through a preset arrangement of blocks and functions. A **case structure** adds a layer of dynamism, allowing us to change the block diagram itself, based on the data flowing through it and various triggers.

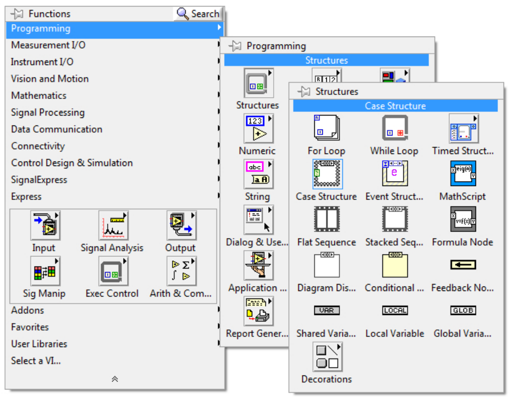
CASE STRUCTURE

2.2 Example VI

The following exercise will demonstrate how to use case structures in a VI.

1. Open LabVIEW.
2. Open a new VI by clicking on `Blank VI` on the LabVIEW `Getting Started` window.
3. Save the VI as `Case Structure.vi`.
4. Create the block diagram in Figure 2.
   - Right-click on the block diagram and place down a `Case Structure` from the `Programming → Structures` subpalette.

Figure 1 `Structures` Subpalette from the `Functions` Palette.

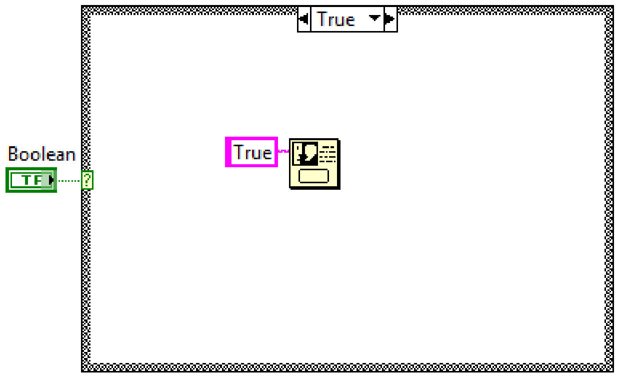


   - Place a `One Button Dialog` function inside both the `True` and `False` cases of the `Case Structure`. This can be found under `Programming → Dialog & User Interface`.
   - Create a `String Constant` from under `Programming → String` and wire it to the `message` input of the `One Button Dialog` function. Initialize the constant to `True` in the true case and a value of `False` in the false case.

   Click on the case selector label on top of the `Case Structure` to view all available cases.

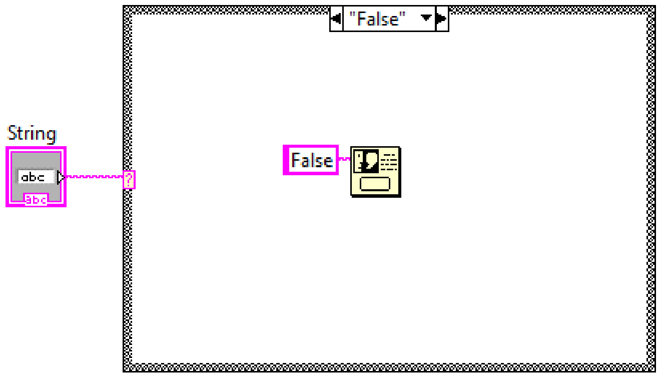
- Create a `Push Button` Boolean control (located under `Programming` → `Modern` on the front panel) and wire it to the `Case Selector` terminal of the `Case Structure`.

Figure 2 `Case Structure.vi` Block Diagram.



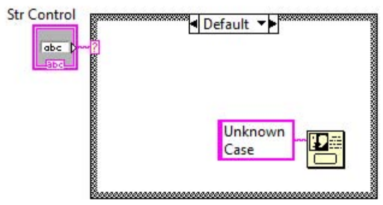
5. Run the program once with the Boolean button on and another time with the button off. The program should send a dialog box describing if the user has selected true or false. Also, run the program in both cases once again, but with execution highlighting enabled. Notice how, based on the value from the Boolean button, the case structure decides which of its cases to include in the block diagram.
6. Modify the `Case Structure` to use a `String Control` as the case selector.
   - Boolean controls are not the only type of controls that can be used with `Case Structures`.
   - Delete the `Push Button` Boolean control on the front panel and create a `String Control`, located under `Modern` → `String & Path`. Wire the `String Control` to the input of the `Case Structure`. Notice that the case types now have quotes.

Figure 3 `Case Structure.vi` with `String Control` Block Diagram.



   - Add a new case to the `Case Structure` called `Default` (without the quotes): this is to ensure that strings other than "`True`" and "`False`" will have a case as well. Cases can be added by right-clicking the border of the `Case Structure` and selecting `Add Case After`.
7. Type `True` in the text box on the front panel, run the VI, and confirm the contents of the dialog box.
8. Type `test` into the text box on the front panel, run the VI, and confirm the contents of the dialog box once more.

Figure 4 `Case Structure.vi` showing `Default Case`.

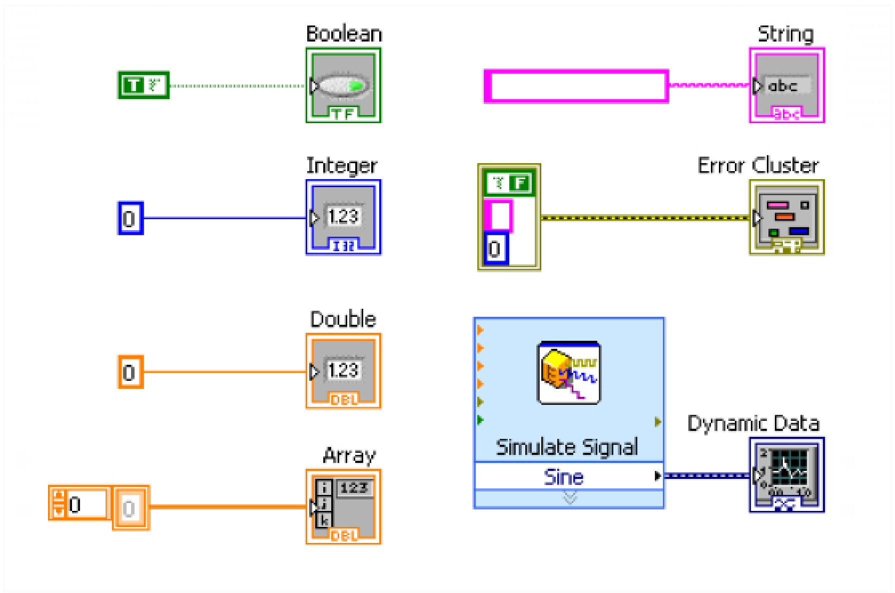


3 Arrays

3.1 Data Types

LabVIEW supports a variety of data types, a few of which you have already seen. The common ones are shown in Figure 5, and we can distinguish between them based on the color of the wires (and using `Context Help`, accessible through `Ctrl-H`).

Figure 5 Different data types (Graphic courtesy National Instruments).



For our purposes this semester, we will mainly use:

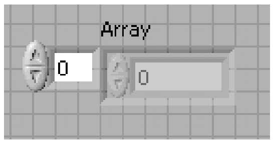
- `Boolean`: This data type handles only two values: `True` and `False`.
- `Integer`: This data type handles integers.
- `Double`: This data type handles double precision floating point numbers; in other words, this data type handles real numbers.
- `Array`: This data type handles arrays of values, whether they be `Booleans`, `Integers`, or `Doubles`. Wires that carry arrays are of the same color as that corresponding to the type of data contained in the array; as shown in Figure 5, for instance, the wire is carrying an array of `Doubles`, and so the wire is orange. However, *the wire is thicker*.
- `String`: This data type handles strings of characters.

3.2 Example VI

In the following exercise, we will build a VI that averages the elements in an array. The exercise will demonstrate how to use arrays and will illustrate some of the auto-indexing functions of loop structures.

1. Open LabVIEW.
2. Open a new VI by clicking on Blank VI on the LabVIEW `Getting Started` window.
3. Save the VI as `Array Average.vi`.
4. Create a 10-element array control on the front panel.
   - From the `Controls` palette, place an `Array` container from the `Modern` → `Array, Matrix & Cluster` subpalette.
   - Place a `Numeric Control`, located under `Modern` → `Numeric`, in the empty `Array` container. The front panel should now match the image shown in Figure 6.

Figure 6 Array Control on `Array Average.vi` Front Panel.



> You must insert an object in the array container before you use the array on the block diagram. Otherwise, the array terminal appears black with an empty bracket.

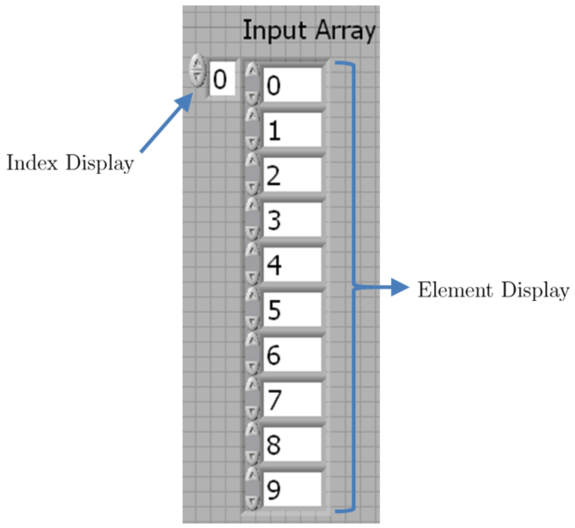
   - Expand the array to ten elements by dragging the bottom edge of the container and initialize them from `0` to `9`, as shown in Figure 7.

3.3 Index Display and Element Display

There are two components to an array control: the **element display** that shows the elements in the array, and the **index display** that shows the index of the first element in the element display. In the example shown in Figure 7, the first element shown in the element display is the element at position `0` in the array, because the index display shows `0`. Change the index display to see the corresponding changes in the element display.

This brings up an important point: when counting the elements in an array, we start at position 0, not at position 1. This is the same as how iterations of a loop structure (`For Loops` and `While Loops`) are counted.

Figure 7 Index Display and Element Display.



5. Average the elements in the array and display it as the output of an indicator.

   - Right-click on the block diagram and place down a `For Loop` from the `Programming` → `Structures` subpalette.
   - Open the block diagram and wire the `Array` control to the left edge of the `For Loop` forming a **tunnel**, whereby the contents of the `For Loop` can communicate with the rest of the block diagram. TUNNEL

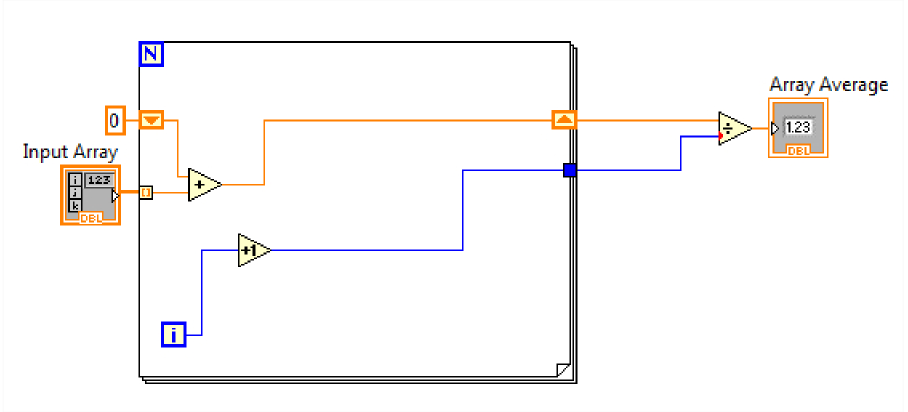
     Notice that when connecting an array to a `For Loop`, the tunnel has **indexing** enabled (▣). This will force the `For Loop` to process each element of the array at each iteration. Note that the count terminal `N` did not need to be specified since the input array was already predefined in size; therefore, the loop will run as many times as there are elements in the array. We will further explore the concept of a tunnel and of indexing a little bit later in the lab.

   - Right-click on the edge of the `For Loop` and add a shift register.
   - Create a numeric constant from the `Programming` → `Numeric` subpalette and change its data type to a double precision floating point number (DBL). Notice the change in color.
   - Initialize the shift register by wiring a constant (which should have value `0`) to the left square.
   - Place down an addition function in the loop. Wire the output of the addition to the right square, and *then* wire the indexed tunnel and the left square to its inputs. *The right square should be connected first, before the left square*: connecting the right square before the left square determines the kind of data carried by the shift register.
   - Using a division function *outside the loop*, find the average of the elements in the array and display it on a numeric indicator labeled `Array Average`. Note that the iteration index `i` can be used to find the total number of elements in the array (how and why?). The completed block diagram is illustrated in Figure 8.
   - Right-click on the blue outgoing tunnel and select `Disable Indexing`.

6. Run the VI and examine its behavior.

7. Run the VI again while viewing the block diagram, and with execution highlighted. Notice that the elements of the array are taken individually into the `For Loop`, and that the final iteration index leaves the `For Loop` after the `For Loop` completes its execution.

Figure 8 `Array Average.vi` Block Diagram.



8. Navigate to the front panel and verify the averaged output.

   Now, let's take a detour to further explore the idea of tunnels and indexing.

3.4 Tunnels and Indexing

Note that the iteration index is coming out through an **outgoing tunnel** with *no* indexing enabled (▪), as opposed to a tunnel with indexing enabled ([ ]). We needed the **incoming tunnel** with indexing enabled so that we can take the elements of the array one-by-one.

When an incoming tunnel has indexing *enabled*, the loop structure (either a `For Loop` or a `While Loop`) will take the elements of the incoming array one-by-one. If indexing is *disabled*, the loop structure will take the array in as a whole.

When an outgoing tunnel has indexing *enabled*, the loop structure *collects* the result of each iteration and puts it into an array. As a result, after a loop structure finishes running its iterations, the outgoing tunnel will produce an array where the ith position contains the result of the ith iteration. In other words, the output is being **indexed**. When an outgoing tunnel has indexing *disabled*, only the last value from the last iteration will pass through the tunnel.

Here, we are only passing one value, as opposed to an entire array, outside the `For Loop`, and so we do not need a tunnel with indexing enabled. Later on, we will see an example where the outgoing tunnel needs to have indexing enabled.

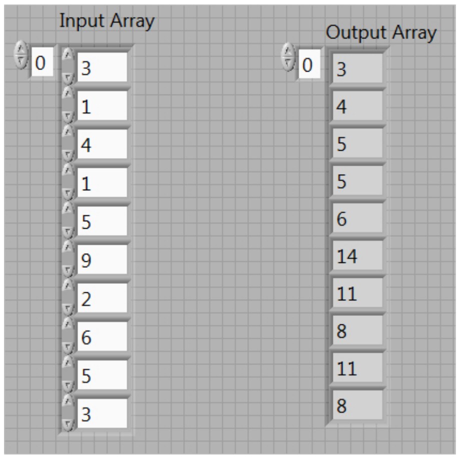
What would happen if we had enabled indexing on the blue outgoing tunnel in `Array Average.vi`?

Try re-enabling indexing on the outgoing blue tunnel and see what happens.

3.5 Self-Exercise

Now that we have learned how to use arrays with `For Loops`, implement the following VI which takes an array and sums each element with the previous element. An example of a test case is shown in Figure 9. Save this VI as `Previous Element Sum.vi`. You may find shift registers useful for this exercise.

Figure 9 `Previous Element Sum.vi` Front Panel.



4 Array Manipulation

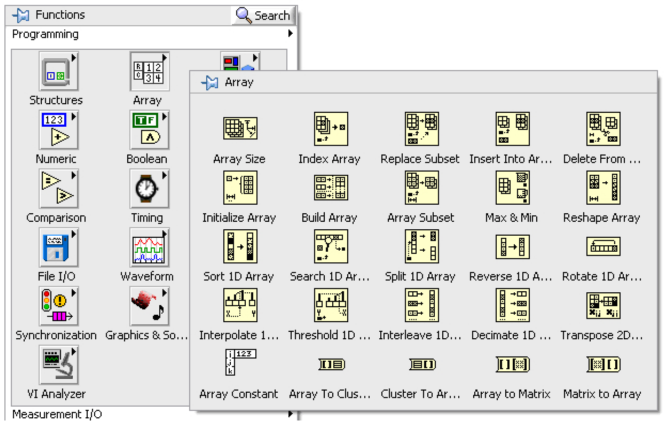
4.1 Example VI

The following exercise will introduce some of the common functions in the `Array` subpalette when performing array manipulations. This VI will initialize an array of 10 elements and have a control on the front panel to replace elements within the array.

1. Open a new VI called `Build Array.vi` in LabVIEW.
2. Initialize a 10-element array constant in the block diagram.
   - Using the `Initialize Array` function under the `Programming` → `Array` subpalette, feed the appropriate parameters to create a 10-element array filled with zeros.

     Use `Context Help` (`Ctrl-H`) and hover over different functions and structures to get a brief explanation of how they work.

Figure 10 `Array` Subpalette from the `Functions` Palette.



3. Create a `While Loop` with a `Stop` button to have this VI run continuously.

- Place down a `While Loop` on the block diagram.
- Right-click on the `Loop Condition` terminal of the `While Loop` and create a control. This will place a `Stop` button on the front panel.
- Place down a `Wait` function and have each iteration of the loop wait at least 100 ms. The `Wait` function can be found in the `Programming` → `Timing` subpalette.

  Although this function is not required in order for the VI to run, it is good practice to have `Wait` functions within `While Loops` as this will significantly reduce the amount of CPU resources taken when running this VI. In the absence of the `Wait` function, CPU resources will be devoted to the VI for the entire time that it runs, causing the rest of the computer to run slower.

4. Complete the block diagram for this VI to replace specific elements from a pre-initialized array.

   - Add a shift register and initialize it to the pre-initialized array from step 2.

     Yes, shift registers can carry whole arrays as well.

   - Place down a `Case Structure` and attach a `Boolean Control` (specifically, an `OK Button`) labeled `Insert` to its `Case Selector` terminal.
   - Place down a `Replace Array Subset` function from the `Programming` → `Array` subpalette and two numeric controls labeled `Element to Insert` and `Index` in the `True` case of the `Case Structure`. Wire the terminals of the `Replace Array Subset` function with the appropriate controls.

     Remember: the right square should be connected before the left square.

   - Draw the output of the `Replace Array Subset` block out of the `Case Structure`.
   - Create an array indicator at this array output of the `Case Structure`. Label it `Output Array`.

     Recall that the output of the `Replace Array Subset` block is an array. Thus, if an indicator is created at the array output of the `Case Structure` (right-click and select `Create` → `Indicator`), it will produce an indicator that can show the elements of the output array.

   - In the `False` case, simply feed the array through the `Case Structure` without any change.

     What have we done so far? We have taken an array from a previous iteration, and, if in this current iteration, the `Insert` button has been pressed, then element replacement is done. Otherwise, nothing happens to the array, and it gets pushed to the next iteration as is.

     The thickness of the wire is important. Notice how, in the block diagram of `Array Average.vi`, the wires inside the `For Loop` were thin. In that example, we were working with each of the values in the array one-by-one. Here, the wires inside the `While Loop` are thick, because we are using the entire array, and not each of the values of the array. Since we initialized the shift register with an array, the shift register can carry only the entire array, and not separate values in the array.

   - The completed block diagram is illustrated in Figure 11, while the completed front panel is illustrated in Figure 12.

5. Run the VI and verify its behavior.

5 Waveform Generation

5.1 Terminology

The **waveform** data type is another data type that we will use frequently. It represents the *sampled* version of a waveform. The values of any function at regularly spaced points are known as the **samples** of that

WAVEFORM

SAMPLES

Figure 11 `Build Array.vi` Block Diagram.

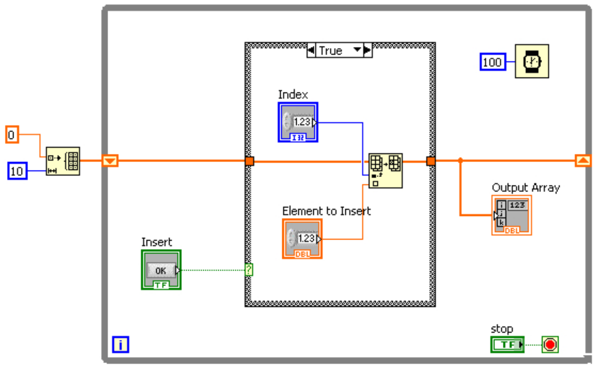
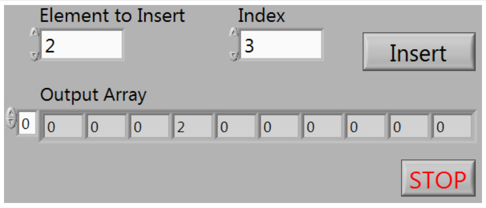


Figure 12 `Build Array.vi` Front Panel.



function. A large chunk of signal processing is devoted to determining how to take a continuous-time signal, obtain samples of the signal at various regularly spaced points, and then re-obtain the original signal from these samples. The length of the space between two samples is known as the **sampling period**. The waveform data type carries both the samples and the sampling period of a waveform.

SAMPLING PERIOD

5.2 Example VI

The following exercise will demonstrate how to build a ramping waveform and plot the data using a graph display. This VI will generate a ramping function from a specified minimum to a maximum across a user-defined time span and number of points.

1. Open a new VI called `Ramp Generation.vi` in LabVIEW.
2. Generate values for the ramping function.
   - Place down four numeric controls on the front panel, labeled `Max`, `Min`, `# Points`, and `Time Span`, as shown in Figure 17.
   - Using a `For Loop` and mathematical functions, generate an array of values for the ramping function based off of the control values. For reference, the graph of the ramping function is shown in Figure 13. The block diagram to generate this is shown in Figure 14.

Figure 13 Ramp function based on parameters provided.

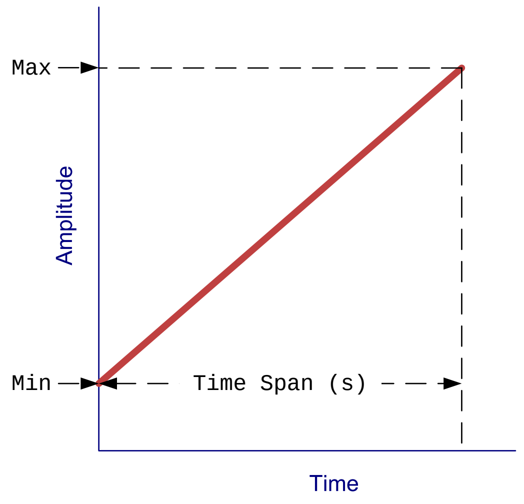
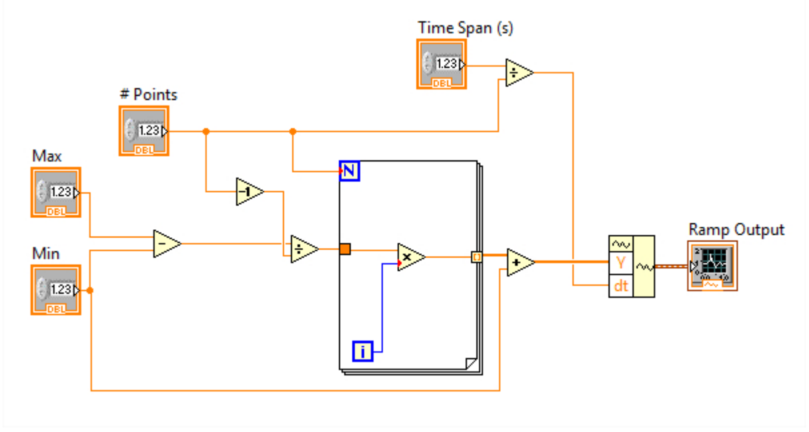


Figure 14 `Ramp Generation.vi` Block Diagram.



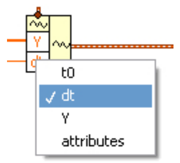
Notice several things about this diagram: it takes the difference between `Max` and `Min` and divides that interval by the number of data points (`# Points`) that the user requires. In the case of the image shown, this would mean that the user requires `5000` points to span the difference between `0` and `10`. In other words, the value of the ramp function at the i^{th} point is $\frac{10-0}{5000} \times i$. The `For Loop` allows us to traverse through the values of i from `0` to `5000`.

Notice also that the outgoing tunnel has indexing enabled; it is an **auto-indexing tunnel**. When an *incoming* tunnel has indexing enabled, it considers each of the elements of the array one-by-one; when an *outgoing* tunnel has indexing enabled, it collects the results of each iteration and makes an array from them. This can be neatly seen from the thickness of wire on either side of the outgoing tunnel: the wire on the left, containing just one scalar value, is thin; the wire on the right, containing the array of all these scalar values, is thick.

3. Build a waveform from the ramping function values and plot it using a graph.

   - Place down a `Build Waveform` function from the `Programming` → `Waveform` subpalette and wire the function values to the `Y` input and the *sampling period* to the `dt` input. The `Y` input tells the `Build Waveform` function where it can obtain its values to construct the resulting waveform; the `dt` input tells the function the distance between two adjacent samples. Note that you can expand the `Build Waveform` function by stretching the function block to display more inputs. You can change the inputs to the function by left-clicking on the input label, as shown in Figure 15.

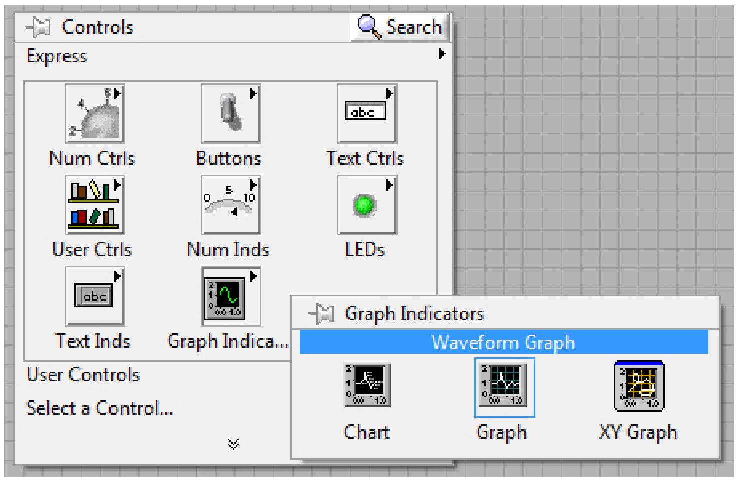
Figure 15 `Build Waveform` Function.



In the case of this exercise, the sampling period would be the amount of time between two adjacent samples, which is given by $\frac{\texttt{Time Span}}{\texttt{\# Points}}$. To derive this relationship, think of samples as the pickets of a picket fence, and the sampling period as the distance between two adjacent pickets.

- On the front panel, place down a `Waveform Graph`, located under `Express` → `Graph Indicators`.

Figure 16 `Graph Indicators` Subpalette from the `Controls` Palette



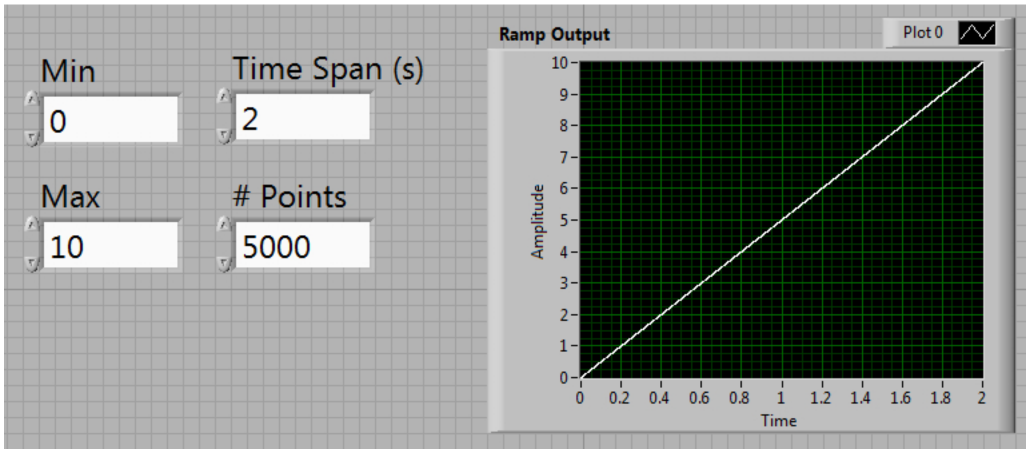
- The completed block diagram is illustrated in Figure 14, while the completed front panel is illustrated in Figure 17.

4. Run the VI and verify its behavior.

6 Acknowledgments

Special thanks go out to the teaching assistants (TAs) of the Spring 2009 semester (Vinay Raj Hampapur, Miklos Christine, Sarah Wodin-Schwartz), of the Fall 2009 semester (David Carlton, Judy Hoffman, Mark Landry, Howard Lei, Feng Pan, Changho Suh), and of the Spring 2010 semester (Xuan Fan, Brian Lambson, Kelvin So) for providing suggestions, ideas, and fixes to this lab guide. Portions of this lab guide were derived from material contained in the LabVIEW reference guides published by National Instruments.

Figure 17 `Ramp Generation.vi` Front Panel.



Min
0

Time Span (s)
2

Max
10

Points
5000