CHAPTER 6

Arrays and Clusters

You briefly encountered the use of arrays and clusters during the discussion about **Waveform Chart** and **Waveform Graph**. This chapter will explore them in detail along with examples.

6.1 Arrays

An array is a data type that has one or more elements of the *same* type such as an integer, single or double precision, or Boolean. As long as the type of each element is identical, the array structure can be used to contain multiple elements. Using mathematical terms, vectors and matrices are arrays where the former are 1-dimensional, and the latter, 2-dimensional. One advantage of the array data structure is that you do not have to carry all of the elements individually, as referring to the name of the array provides full access to each element through indexing.

Consider the following example to see an advantage of the array structure: When you create a sub VI, you need to create terminals for the objects in the front panel of the sub VI. (See Chapter 3 for the complete discussion of sub VIs.) A large number of objects can create a problem since a large number of terminals will make the wiring difficult. However, if the type of objects is identical, you may use a single terminal for an array, which consists of all of the objects of identical data type. For example, suppose that you have 10 numeric indicators, which will return 10 numeric values to another VI. Instead of having ten terminals, you can create an array indicator of those 10 numeric indicators, and assign a terminal to the array. If the data type of objects is different, you will need to consider a data type cluster, which is the topic in the next section.

If you want to create an array from multiple objects of identical data type, you will need a function that combines them into an array, and **Build Array** does that. In order to access each individual element or a portion of the array, use **Index Array** or **Array Subset**. Since the array data structure is important enough to be emphasized repeatedly, each function in the **Array** sub-palette will be discussed in Section 6.1.2. The next section discusses how to create an array.

6.1.1 Creating an array

During the discussion of arrays, lower case letters in boldface will be used to indicate a one dimensional (1-D) array (or a vector). Upper case letters in boldface represent a 2-D array (or a matrix). Therefore, **X** is a matrix and **x** is a column vector.

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1,M} \\ x_{21} & x_{22} & \cdots & x_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,M} \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

In C or C++, arrays are declared as follows:

```
int x[100], X[100][100];
```

The above expression declares a 1-D array (a vector) and a 2-D array (a matrix), and each element is accessed by its index. LabVIEW declares arrays in a different way, however, and there are two steps in creating (declaring) them:

Step 1 Place an *array shell* in either the front panel or the diagram window. If the array will be either a control or an indicator, place it in the front panel. If the array will be a constant, place it in the diagram window.

Step 2 Declare the type of the array by dropping a proper data type of a control, an indicator, or a constant in the array shell.

Arrays may be of any data type such as Boolean or numeric. They can also be either controls or indicators depending on the type of element in the array shell. Step 3 in Figure 6.1 shows an array of numeric control so that you can *enter* data instead of displaying it. Step 1 in Figure 6.1 is an empty array shell obtained from the **Array** sub-palette in the front panel pop-up menu. Step 2 shows how it will appear when you get a numeric **Digital Control** from the **Numeric** sub-palette in the front panel pop-up menu. When you place it in the empty array shell, it will show a dotted line to indicate that the element is locked in the position. Releasing your mouse button will complete the step, and you will have an array of numeric control as shown in Step 3.

The up and down arrows of the shell are to increase or decrease the start index of display. In other words, if you have 3 as the index of the shell, the first element has the index 3 making it the fourth element in the array since indices start at zero in LabVIEW. To increase the dimension of array, drag the index box down to show more indices. Figure 6.2 shows the steps in changing the dimension of an array.

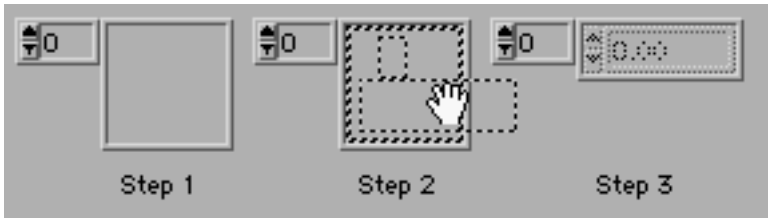


Figure 6.1 Steps to create an array. Step 3 shows an array of numeric control type.

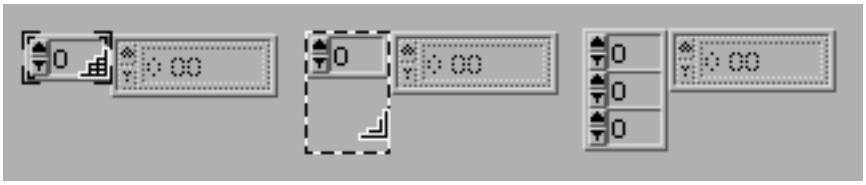


Figure 6.2 Increasing the dimension of an array from 1-D to 3-D

Now that you know how to create an array, you must learn how to manipulate it with actual data. I n LabVIEW, there are two ways to build an array: concatenation and indexing. In Chapter 4, you learned about indexing during the discussion about the **While Loop** and the **For Loop**.

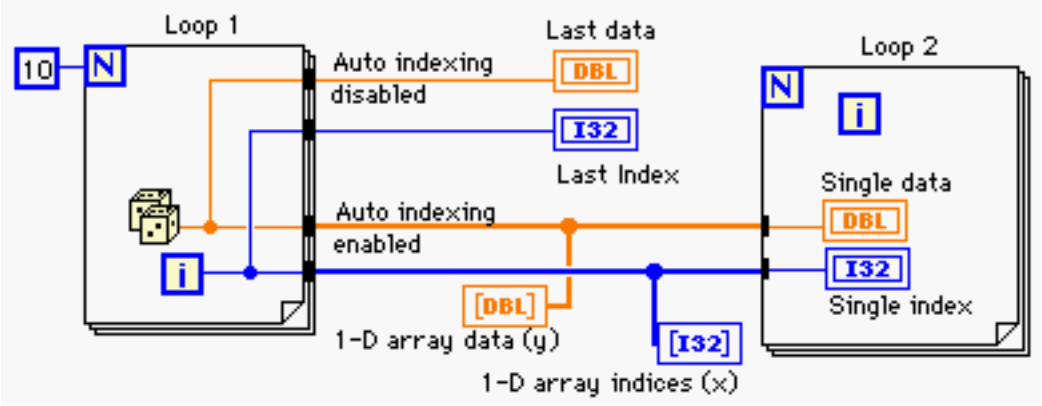


Figure 6.3 Indexing with the **For Loop**

For the sake of convenience, the indexing example is again shown in Figure 6.3. The thickness of the wire before and after the boundary of the two **For Loop**s shows the dimension of the data. In Loop 1, the upper wires leaving the **For Loop** have the same thickness since the indexing is disabled. Indexing can be disabled by right clicking on the black dot and selecting **Disable Indexing**; however, the lower wires leaving Loop 1 are thicker, indicating that the dimension of data has been increased. In this case, the data leaving Loop 1 at the bottom is a 1-D array constructed from 10 data elements. This indexing method is the fastest way to create an array and requires the least amount of overhead. In other words, in LabVIEW, creating arrays with the indexing feature is optimized.

Another way of creating an array is by concatenating data of the same type. This means that you manually combine multiple elements to create an array using **Build Array**. Consider the following example in Figure 6.4 and Figure 6.5. In Figure 6.4, you have two vectors, **x** and **y,** of numeric *control* type. This means that you can *enter* or *set* values. Their size is identical and is 3 by 1. As shown in Figure 6.4, you can build another vector **z** from **x** and **y**, or create a matrix **A**

with its columns or rows using **x** and **y**. (**A**' indicates the transpose of **A**.) In order to build another vector or matrix using **Build Array** as shown in Figure 6.4 and Figure 6.5, you need to understand the two options with the **Build Array**: element input and array input.

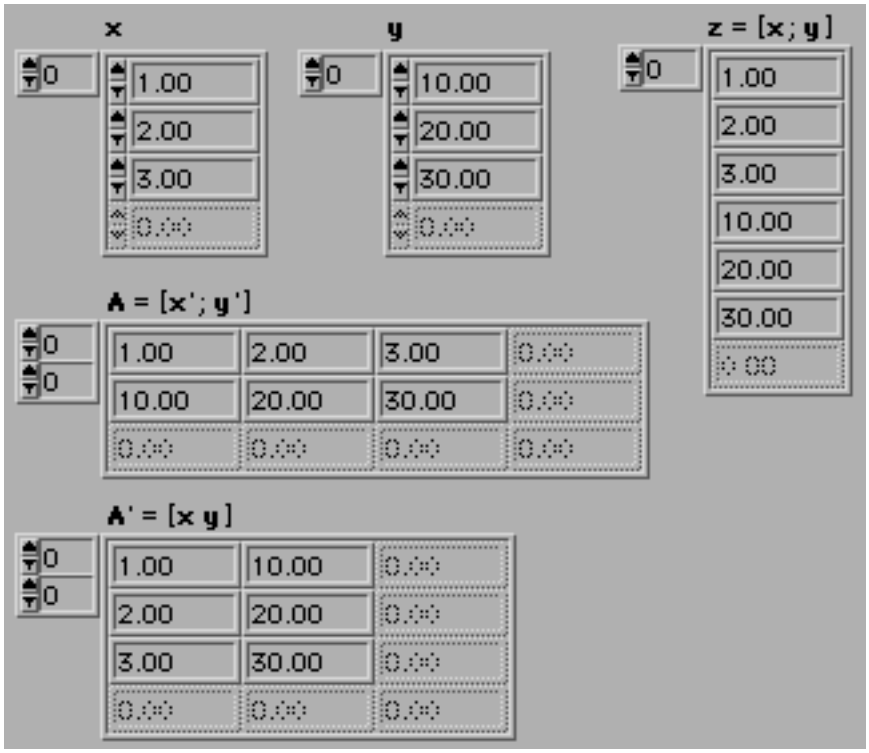


Figure 6.4 Building different arrays using **Build Array** (front panel)

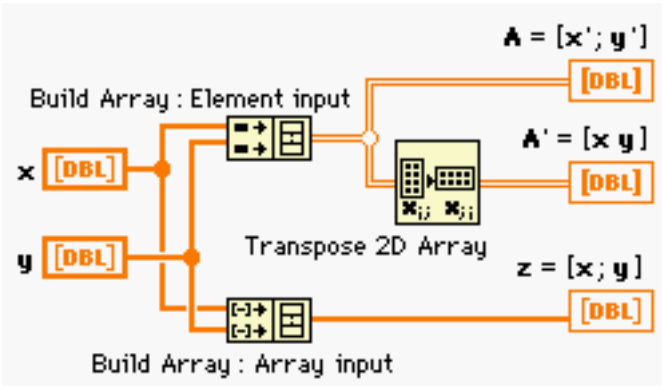


Figure 6.5 Building different arrays using **Build Array** (diagram window)

In Figure 6.5, two different types of input to the **Build Array** are shown. You may have realized that the shape of the input to the two **Build Array**s is different: one with two black square dots, and the other with two dashes [-]. In order to change the type of input to the **Build Array**, right click on the input to pop up its menu, and select either **Change to Array** or **Change**

to Element. With the two different types of input, the output of the **Build Array** will be either a new array with increased dimension or a new array of the same dimension. The thickness of the wire at the output of the two **Build Array**s shows the dimension of the output array. If you use the element input type, this will increase the dimension of the input array by one; therefore, using a 1-D array (vector) as its input will create a 2-D array (matrix) which is **A** in Figure 6.5. If you use the array input type, however, this will not increase the dimension, but create an output array of the same dimension, which is just another 1-D array (vector). The notation [**x y**] has been used to indicate two columns where each column is the vectors **x** and **y**, and [**x**;**y**], one column with **x** stacked on **y**. One last important reminder about the **Build Array** is that it treats 1-D array inputs as *row* vectors instead of column vectors. Therefore, the output of the **Build Array** with the element input type using 1-D arrays **x** and **y** as its input will be a matrix **A**, and its first row will be **x** and the second row, **y**. To transpose **A** to have **x** in the first column of **A**, and **y** in the second column, use **Transpose 2D Array**. It can be found in the **Functions** >> **Array** sub-palette.

6.1.2 Array functions

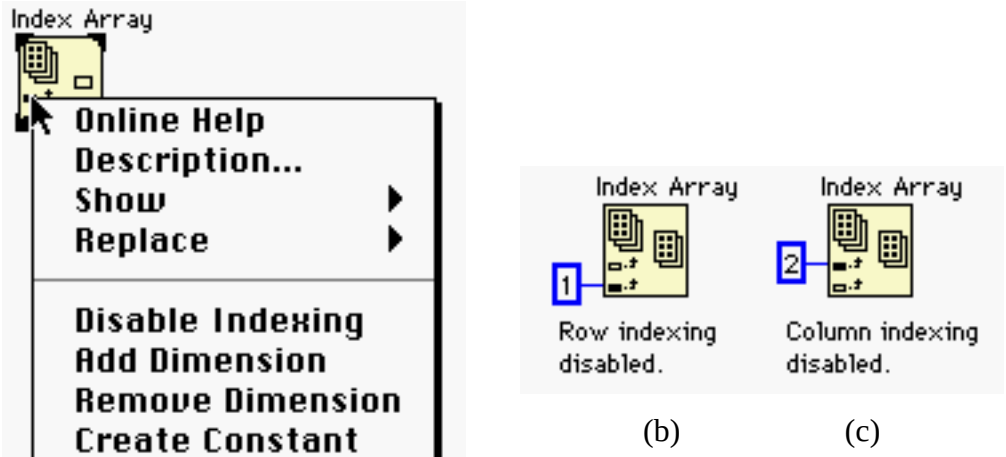
This section will examine each LabVIEW function in the **Array** sub-palette in the **Functions** palette.

 **Array Size** This function returns the dimension or the size of the input array. If the input is 1 dimensional, the output will be a scalar. If the input is 2 dimensional, the output will be a 1-D array whose first element corresponds to the length of row, and the second element to the length of column.

 **Index Array** This function provides access to each element in an array and has two input types: the array, and the index or indices. To increase the number of index input terminals, point the arrow cursor to the right bottom corner until the shape of the arrow cursor turns to a reversed L shape, and drag it up or down. The number of index input terminals should be the same as the dimension of the input array. This also allows you to index certain entire row or column elements. For example, consider a 2-D array **X** of size 3 by 4 as follows:

$$\mathbf{X} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

In order to index any entire column of **X**, you need to disable the indexing of row input. If you want to index any entire row of **X**, the indexing of column input needs to be disabled. You can disable or enable the indexing of row or column input by right clicking on it, and selecting **Disable Indexing** or **Enable Indexing** from the pop-up menu. If it is disabled, it will appear as a white square dot, and if it is enabled, its shape will be black square dot. (See Figure 6.6) The default setting is enabled indexing (black square dot). Note that since **X** is a 2-D array, you must have two index input terminals.



(a) (b) (c)

Figure 6.6 Disabling or enabling index input terminals of **Index Array** to index an entire row or column of a 2-D array input. The **Index Array** in (b) will return the second column of the 2-D input array since the index starts at zero. The output of the **Index Array** in (c) will be the third row of the 2-D input array.

If the 2-D array **X** is the input to the **Index Array** in (b) and (c) in Figure 6.6, (b) will return a 1-D array whose elements are 4, 5, and 6, and (c) will return a 1-D array whose elements are 3, 6, 9, and 12.

 **Replace Array Element** If you want to replace individual elements in an array, this function can be used. You specify the original array to be modified, the value for replacement, and its index as the input to this function. To increase the number of index input terminals, follow the method stated in the description about **Index Array**.

 **Array Subset** This function can extract a range of elements from the input array. The inputs are the original array, and the starting index and the length of the range to be extracted.

 **Reshape Array** This reshapes the input array. For example, if the input is a 1-D array of 6 elements, and if you reshape it with 2 for the input **Dimension Size**, it will truncate the original array to one of size 2. If you increase the number of **Dimension Size** input terminals to two, and specify 2 and 3 for their input values, it will return a 2-D array of size 2 by 3 from the original 1-D array of 6 elements. When you increase the dimension of the output array, the number of input array elements and output array elements must be the same. If the dimension of the input array and the output array is the same, the **Dimension Size** input value that is smaller than the length of the input array will truncate the input array, and the larger value will zero pad the input array. This can be useful in FFT data processing when zero padding is desired. (See Chapter 14 for more discussion about zero padding.)

 **Initialize Array** This function is to initialize an array. Dynamic increase in the size of array can result in an inefficient memory usage in LabVIEW. You can avoid this by allocating a block of memory to arrays and not changing their size dynamically. Therefore, if the number of array elements will vary, initialize an array that is big enough to hold all of the possible elements instead of changing the size of the array during the execution of VI applications.

 **Build Array** This creates another array using input arrays. The dimension of the output array depends on the selection of input array mode: Element input and Array input. More details about those modes have been discussed in the previous section.

 **Rotate 1D Array** This rotates the elements in a 1-D array. In other words, this function shifts the elements of the input 1-D array by the amount of *n*, which is the number of places (bits) to be shifted. If *n* is positive, it rotates to the right, and to the left for negative values. Therefore, the right-most element will move to the left-most location for a positive *n*, and the left-most element, to the right-most location for a negative *n* during the shifting process.

 **Reverse 1D Array** This reverses the order of elements in a 1-D array.

 **Transpose 2D Array** This transposes 2-D input arrays. This may have to be used when displaying data using the **Waveform Graph**. For more information about the **Waveform Graph**, refer to Chapter 5 where displaying data is studied.

 **Search 1D Array** This has three inputs: a 1-D array, a value that is under search, and the search start index. This function will search for the value starting at the index in the 1-D array input, and returns the index of the value found. If it is –1, the value under search is not in the array, and if the value has more than one appearance in the array, the output is the index where it appears for the first time.

 **Sort 1D Array** This function sorts the input 1-D array elements in ascending order, but does not provide the indices of the elements.

 **Array Max & Min** This returns the maximum and the minimum values in the input array with their indices. If the input array is 1-D, the output indices will be a scalar. If the input is a 2-D array, the maximum and minimum indices are returned in a 1-D array: the first element is the row index, and the second, the column index. If there are duplicates of the value, the output indices correspond to the element at the first appearance.

 **Split 1D Array** This returns two portions of a 1-D input array where the index of the first portion is 0 to n – 1. The index of the second portion is *n* to the end where *n* is the value to the input terminal **index**.

Interpolate 1D Array The name of this function is somewhat misleading. This function takes a 1-D array and a fractional index as its input, and returns an estimated scalar value at that fractional index input. This value is computed by using only the two values in the immediate neighborhood of the fractional index. It is not computed by first fitting all of the elements in the array into an equation and then evaluating it at the fractional index. For example, if the input is [0 1 4 9 16] which is the output of the simple quadratic function $y = x^2$, the value at index 3.75 should be 14.06 since $y(x = 3.75) = 14.06$, but this function returns 14.25 which is the result of first order polynomial data fitting using two closest adjacent values 9 and 16 whose indices are 3 and 4. (9 + (3.75 – 3) x (16 – 9) = 14.25) Refer to Chapter 14 for the discussion about interpolation VIs.

Threshold 1D Array This function takes a 1-D array, a threshold value, and a starting index as its input. The name of this function is also misleading. The name suggests that this function would leave all of the values below the threshold which is connected to the input **threshold y**, and saturate the others above it. However, this function performs the opposite operation to the **Interpolate 1D Array**. If the input array is [0 1 4 9 16], the input **threshold y** is 14.25 which was the output of the **Interpolate 1D Array** in its example above, and the input **start index** is 0, the output **fractional index or x** returns 3.75. In order to obtain the correct fractional index in general, the **start index** input value must be less than the correct fractional index. If the elements in the input array monotonically increase, you may set it to zero, but if the **threshold y** value appears more than once in the input array, the function may return a wrong index unless the **start index** is chosen carefully.


Interleave 1D Array This interleaves each element from multiple input arrays alternatively. If you have **x** = [1 2 3 4] and **y** = [4 5 6] as two inputs to this function, the output array becomes **z** = [1 4 2 5 3 6]. If the size of any of the input arrays is bigger than the others, the output array will be constructed based on the minimal size, and all of the elements beyond the minimal size will be ignored. In this example, **x** is bigger than **y** by one element so that the last element of **x** is ignored in constructing the output array **z**.

 **Decimate 1D Array** This function performs the operation opposite to **Interleave 1D Array**. If the input array is **z** = [1 4 2 5 3 6], and you have two outputs **x** and **y**, this function will return **x** = [1 2 3], and **y** = [4 5 6]. The length of output arrays is identical. The valid input array length is *N* x *M* where *N* is the number of output arrays, and *M* is the length of each output array. If the length of the input array *P* is larger than *N* x *M*, the last *P* – *N* x *M* elements in the input array will be ignored.

 **Array Constant** This creates an empty array shell in the diagram window so that an array constant can be created by placing any type of constant in it.

 **Array to Cluster** This takes an array as input and converts it to a cluster. (Clusters are discussed in the following section.) You can also adjust the number of elements in the cluster beforehand. Just right click on the **Array to Cluster**, choose **Cluster Size…**, and enter a number to change it.

 **Cluster to Array** This reverses the process of the **Array to Cluster**. Note that arrays can only have the same type of data as their elements; therefore, if the input cluster, which can have different types of data, does not have the same data element type, this function will return the Broken Run Arrow. This function is useful if you have a cluster of Boolean elements. For example, suppose you have multiple Boolean buttons as elements in a cluster in the front panel to control different processes such as DATALOGGING, READ DATA, DISPLAY DATA, and so on. Though there are many different ways of doing so, users may monitor which button is pressed by first converting the cluster to an array of Boolean switches, and using **Search 1D Array** to find out if any one of the buttons is pressed. If none is pressed, the output from **Search 1D Array** will be negative one (–1). Based on such output, you may have **Case** structures to perform corresponding tasks.

6.2 Clusters

Clusters are very similar to arrays in that they can have multiple elements. However, clusters can have different data type of elements, whereas arrays cannot. You were introduced to **Cluster to**

Array in the previous section, and now you may ask why you would want to use the cluster type even though the type of each element is the same. The answer is that you can have more flexibility with clusters, even with the same type of data elements. For example, suppose you have an array of Boolean switches with labels START or STOP, etc., and want to have a different effect on each one. If you change the label of any one of the Boolean switches, the rest will all change identically since arrays *must* have exactly the same type (even the labels) of data element. However, if you have a cluster of Boolean switches, you can change the labels individually since each element is independent. This implies that you can have different settings such as color, label, mechanical action, or Attribute Node. This should justify the use of cluster even with the same type of data elements for some occasions. The drawback in using clusters instead of arrays is more overheads, so discretion must be exercised in choosing which one to use with identical data type elements.

6.2.1 Creating a cluster

The method of creating a cluster is similar to that of creating an array. First, you place an empty cluster shell, and insert data elements one by one. Since you are dealing with different types of data elements, the *order* of inserting data elements in the cluster shell is now important. For example, if you want to wire a cluster indicator to a cluster control, each element in the cluster indicator must match that in the cluster control including the creation order and data type. Figure 6.7 shows an example of creating a cluster whose elements are one numeric control and one Boolean LED control. The numeric control was created first in this example. Note that all of the elements *must* be either controls or indicators, but not both; therefore, it may be said that clusters are similar to arrays such that all of the elements must be of the same type (control or indicator).

Figure 6.7 Steps to create a cluster

To change the order of cluster elements, right click on the boundary of the cluster shell as shown in Figure 6.8. Select **Cluster Order…** to display the order as shown in Figure 6.9. Then, the cursor becomes a hand with the index finger pointing to the left. In this mode, you can change the order. After changing the order, click OK to exit the mode.

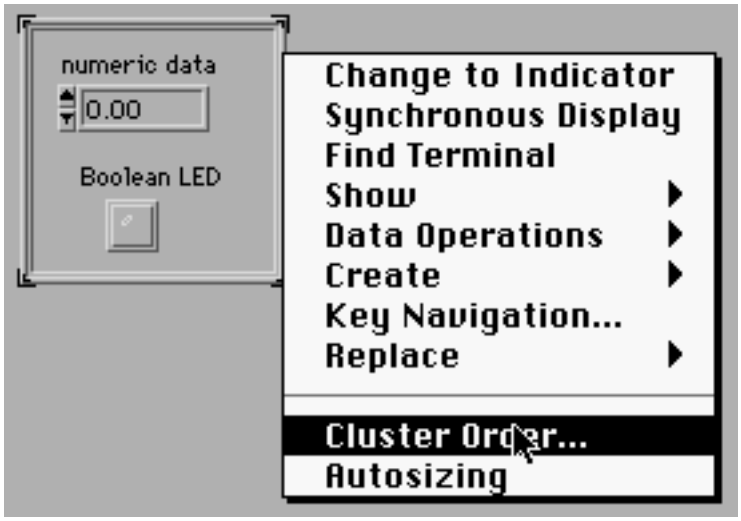


Figure 6.8 Displaying the cluster order

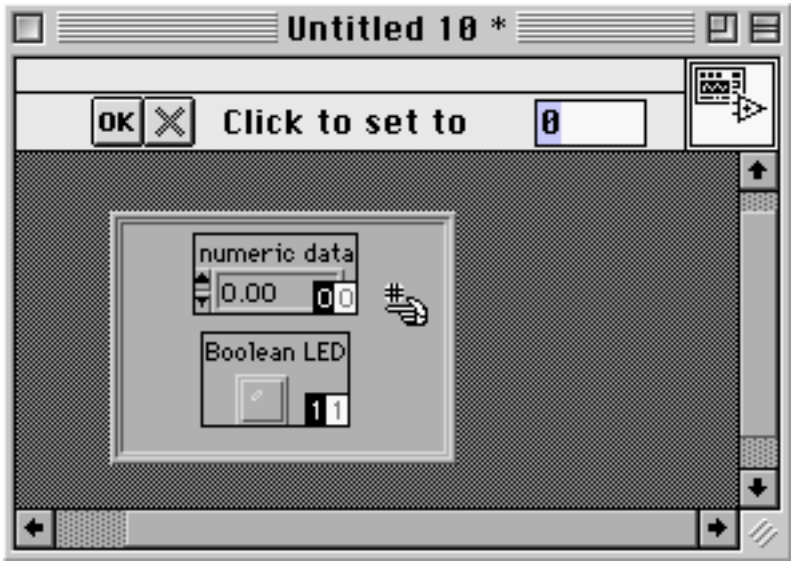


Figure 6.9 Changing the cluster order

6.2.2 Cluster functions

This section will discuss each LabVIEW function in the **Cluster** sub-palette in the **Functions** pop-up menu.


Unbundle This returns each element in the input cluster. When the input cluster is wired, the number of output terminals *must* match that of elements in the input cluster; otherwise, it will return the Broken Run Arrow indicating an error has occurred.


Bundle This function bundles up multiple input elements of different type and returns a cluster. The order of each input element is top first and bottom last. The middle input **Source Element** is optional and has two purposes: building a cluster and modifying a cluster. If you have a *template* cluster structure to build, you may wire it to the middle input. If you do so, the number and the order of input elements and their data type *must* match those of the elements in the template cluster.

The other purpose of the **Source Element** is to replace or modify any element in the cluster. The value to be replaced will be wired to the correct input terminal, and the cluster that needs to be updated will be wired to the **Source Element** terminal. Being similar to **Unbundle**, the number of input terminals *must* match the number of elements in the cluster which is wired to the **Source Element** terminal; otherwise, it will return the Broken Run Arrow.

Unbundle By Name This function provides access to each element in the input cluster. If the elements have labels, each element can be accessed (retrieved) individually by selecting its label. This is the major difference from **Unbundle**. Note that **Unbundle** must have the same number of output terminals as the number of elements in the input cluster. If you have a cluster with 100 elements and want to access just the 99th one, you still have to have 100 output terminals just to access one element with **Unbundle**! However, if you use **Unbundle By Name**, you can have just one output terminal with the label of the 99th element selected. Therefore, it is a good practice to label each element in the cluster even though it is not displayed. It is also recommended that you use **Unbundle** to access the elements when there are relatively few elements in the cluster; otherwise, use **Unbundle By Name**.

Bundle By Name This function is similar to **Bundle** since it also has two purposes and creates a cluster. However, this *must* have middle **Source Element** input wired. The first purpose is to simply create a cluster from a number of input elements of different data type. When you wire the template cluster to the **Source Element** input terminal, all of the labels of the input elements become available in the input terminals. Note that you do not need the same number of input terminals as the number of elements in the template cluster. In order to display all of the input labels, simply drag down the input terminals. Again, the data type and the order must match. The second purpose is to replace any cluster element. For example, suppose you have 100 elements in a cluster and you want to replace the 99th element. With the **Bundle**, you must have the same number of input terminals (100) just to access the 99th one, but **Bundle By Name** allows you to have only one input terminal with the label of the 99th element. Of course, you should wire the cluster that needs to be modified to the **Source Element** input terminal to make all of the 100 labels available.


Build Cluster Array This function builds an array whose elements are clusters that contain the input element wired to the input terminal of the function. For example, if you wire numeric constants 1, 2, and 3 to the function, the output will be [cluster(1) cluster(2) cluster(3)], where [ ] indicates an array, and cluster(*x*) indicates a cluster whose element is *x*. Therefore, if *x* is an array, this function will create an array of clusters where each cluster contains a single 1-D array **x**. For example, if three inputs to this function are **x** = [1 2 3], **y** = [4 5 6], and **z** = [7 8 9], the output will be [cluster(**x**) cluster(**y**) cluster(**z**)] as shown in Figure 6.10.

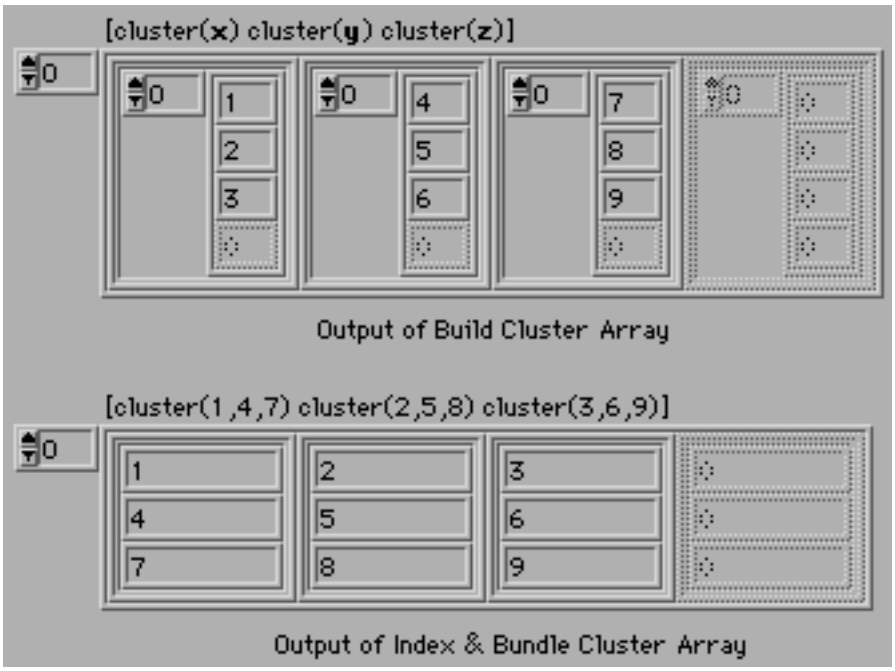


Figure 6.10 Outputs from **Build Cluster Array** and **Index & Bundle Cluster Array** with three inputs **x** = [1 2 3], **y** = [4 5 6], and **z** = [7 8 9]

Index & Bundle Cluster Array This function is similar to **Bundle Cluster Array**, and it creates an array of clusters, but the element of the clusters is different, and the input must be an array. Suppose you have **x** = [1 2 3], **y** = [4 5 6], and **z** = [7 8 9] for the input, then this function will return an array of clusters where each cluster has three elements as follows: cluster (1, 4, 7), cluster (2, 5, 8), and cluster (3, 6, 9). In other words, the output is [cluster (1,4,7) cluster (2,5,8) cluster (3,6,9)] as shown in Figure 6.10. Here, [ ] indicates an array, and cluster (*a*,*b*,*c*) indicates a cluster whose elements are *a*, *b*, and *c*.

Cluster Constant This constant is similar to **Array Constant** and provides an empty shell to create a cluster constant. You can place any type of constant in the shell to define its type.

6.3 Comparison of Functions

All of the functions for the array and cluster manipulation have been covered. Since their natures are somewhat similar to each other, they may cause some confusion. The following are a brief comparison and summary of the two types.

6.3.1 Build Array and Bundle (By Name)

These two functions are used to create an array or a cluster. You should be aware of the creation order of cluster elements. **Bundle** and **Bundle By Name** differ in terms of the number of input terminals required, and a cluster template can be wired to the input terminal **Source Element**.

6.3.2 Index Array and Unbundle (By Name)

These are to access each element in an array or a cluster. Be aware of the differences between **Unbundle** and **Unbundle By Name**.

6.3.3 Replace Array Element and Bundle (By Name)

These functions are to replace an element in an array or a cluster. As for the cluster, the **Source Element** input terminal *must* be wired, and such wiring is not optional. You must have the same number of input terminals as that of elements in the cluster, and the cluster must be wired to the **Source Element** input terminal of **Bundle.** However, this does not apply to **Bundle By Name**.

6.3.4 Memory issue

Dynamically resizing arrays will cause an inefficient memory usage, and it can be prevented by first allocating a block of memory to the arrays. This may be done with **Initialize Array**. Clusters impose more overheads and can significantly slow down the process if they have complex structure elements. However, clusters provide excellent flexibility since elements are independent of each other. Note that all of the elements in a cluster must be either indicators or controls.

PROBLEMS

6.1 Generate two 1-D arrays **x** and **y** whose elements are as follows:

$$\mathbf{x} = [1\ 2\ 3\ 4\ 5],\ \mathbf{y} = [6\ 7\ 8]$$

Using **x** and **y**, perform the same task as shown in Figure 6.4. Save the VI as **P06_01.vi**. How are the results different from those shown in Figure 6.4?

6.2 Create a 2-D array **X** of size 3 by 4 (three rows and four columns) with random numbers between 0 and 1. You may use two nested **For Loop** to generate such a 2-D array quickly. Complete the VI that can select a different column, and display the selected column **x** until you quit the VI using a **While Loop**. Note that you will need to create a 2-D array only once before the **While Loop** initiates. The complete front panel is shown in Figure P6.2. Save the VI as **P06_02.vi**.

6.3 Expand the functionality of **P06_02.vi** such that the VI allows you to enter either column or row index, and displays the result in **x**. Its front panel is shown in Figure P6.3. Save the VI as **P06_03.vi**.

6.4 Build a VI which performs binary addition of two positive 1-byte long integers, and its complete front panel and diagram window are shown in Figure P6.4. The following are key functions used in this example:

Functions >> **Array** >> **Reverse 1D Array**
Functions >> **Boolean** >> **Boolean Array To Number**
Functions >> **Boolean** >> **Number To Boolean Array**
Functions >> **Numeric** >> **Conversion** >> **To Byte Integer**

Save the VI as **P06_04.vi**, and explain the functionality of each VI. Why do you need to use **To Byte Integer**?

6.5 Create a cluster control **BIO Original** with three different types of data: **Name** (String), **Age** (U8 integer), and **Marital Status** (Boolean). Their order is top the first, and the bottom the last in **BIO Original**. Complete the VI which updates the second element with the age in **New Age**, and displays the result in the cluster indicator **BIO Updated**. Use **Bundle** to complete the example, and save it as **P06_05.vi**. The complete front panel is shown in Figure P6.5. How many input terminals of **Bundle** will it take to run the VI successfully?

6.6 Repeat Problem 6.5 using **Bundle By Name** in lieu of **Bundle**, and save it as **P06_06.vi**. The front panel will be the same as Figure P6.5. How many input terminals of **Bundle By Name** are needed to run the VI successfully?

6.7 First, save **P06_06.vi** as **P06_07.vi**, and delete all of the labels of three controls in **BIO Original** in **P06_07.vi**. The **Run** button  will turn into  **List Errors** (Broken Run Arrow). Explain why, and suggest how the error should be fixed. Also, summarize your observation from Problem 6.5 through Problem 6.7.

6.8 One application where clusters are commonly used is in a database. Figure P6.8 shows a simple example with three entries, **Name**, **Age**, and **Marital Status** in each record. Note that an array of cluster is used for **Database Array**. Build a VI as shown in Figure P6.8,

and save it as **P06_08.vi**, and explore its functionality with different entry values. Can you *add* more record to **Database Array** using the diagram in Figure P6.8?

6.9 The database example in **P06_08.vi** is only to *update* the record in **Database Array**. In order to be able to *add* more record to **Database Array**, how would you modify **P06_08.vi**? Complete your suggestion, and save it as **P06_09.vi**.

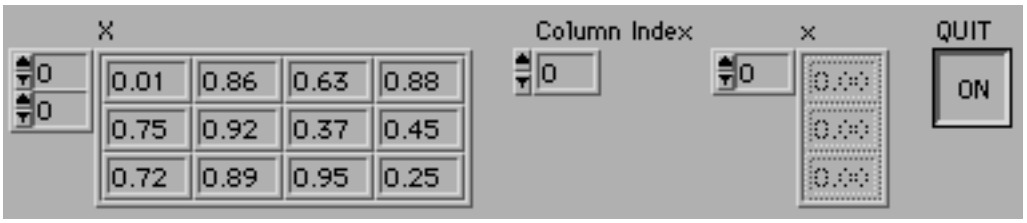


Figure P6.2

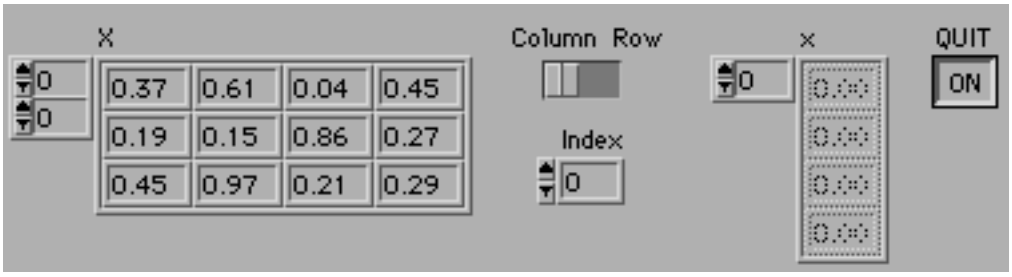


Figure P6.3

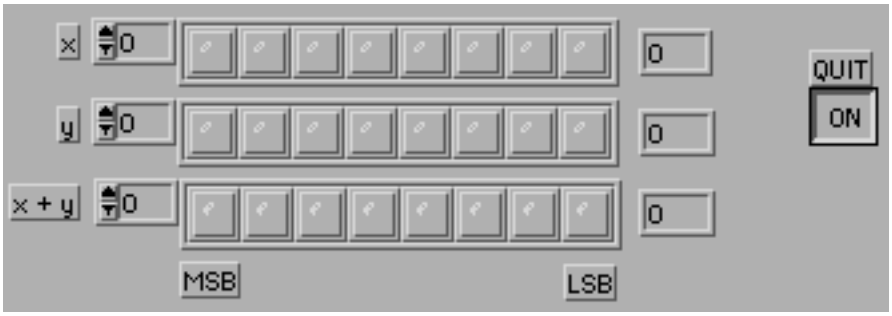


Figure P6.4 (a)

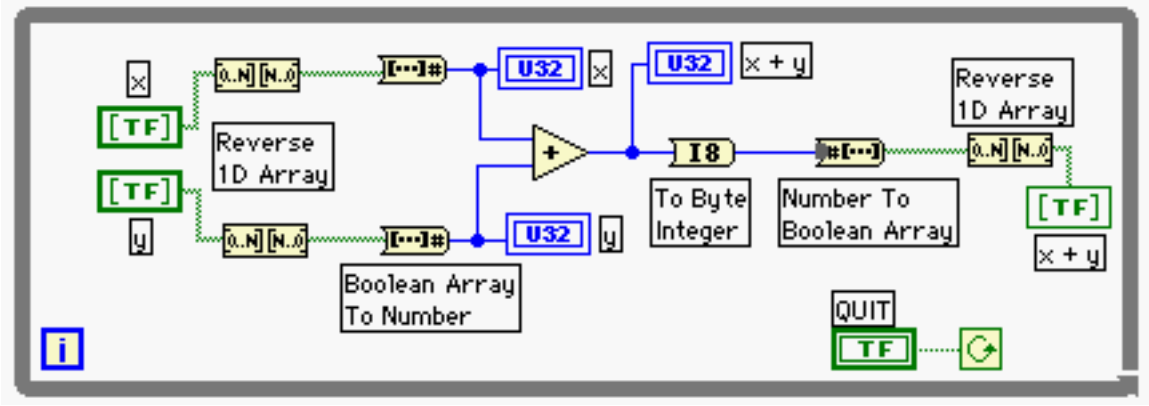


Figure P6.4 (b)

Figure P6.5

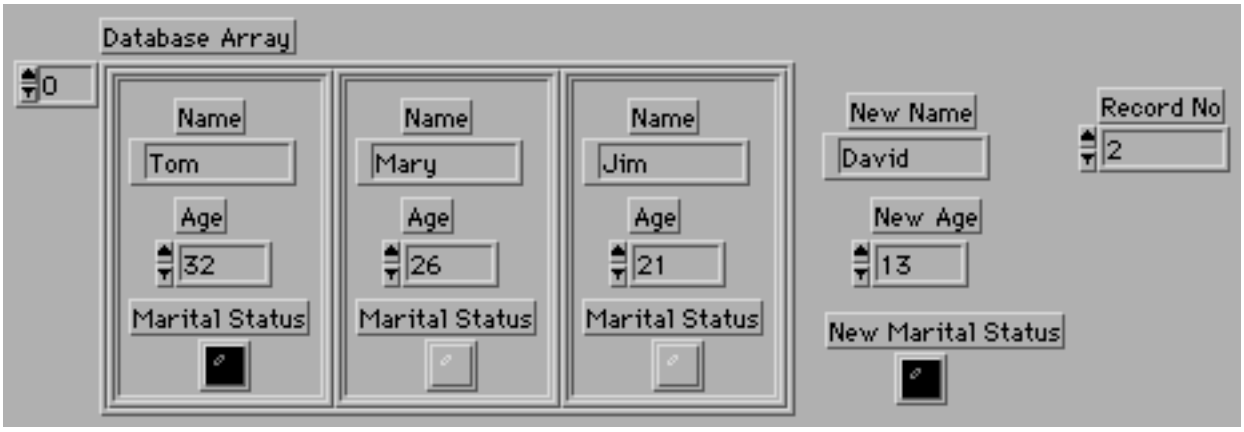


Figure P6.8 (a)

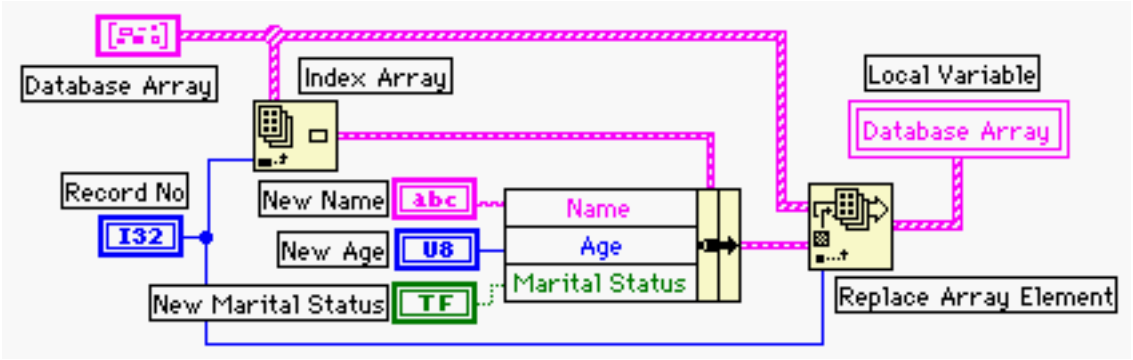


Figure P6.8 (b)