

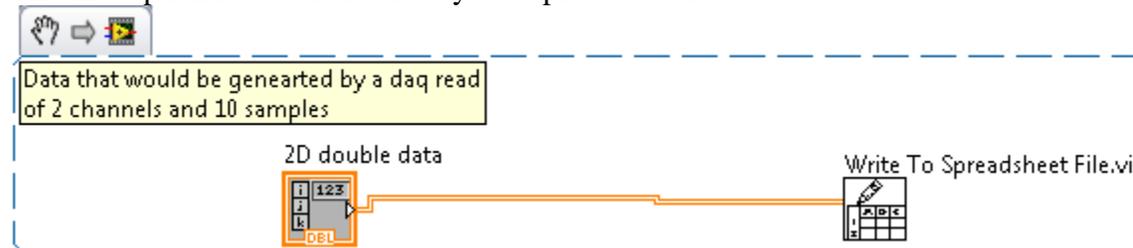
File Storage Techniques in LabVIEW

Starting with a set of data as if it were generated by a daq card reading two channels and 10 samples per channel, we end up with the following array:

		2D double data										
Channel	0	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00	0.00
Sample number	0	10.00	20.00	30.00	40.00	50.00	60.00	70.00	80.00	90.00	100.00	0.00

Note that the first radix is the channel increment, and the second radix is the sample number. We will use this data set for all the following examples.

The first option is to send it directly to a spreadsheet file.



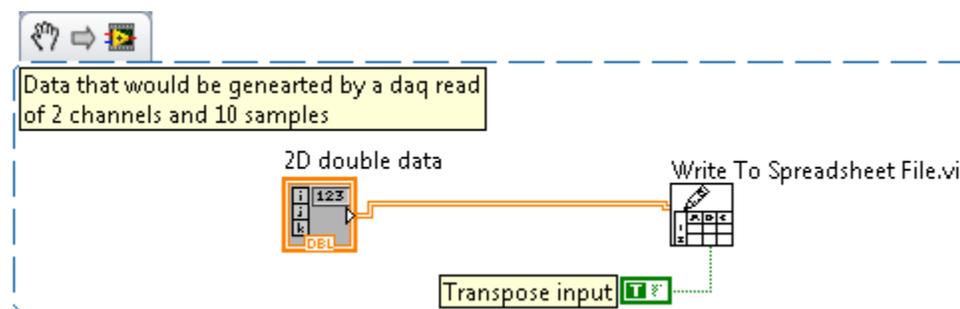
[spreadsheet save of 2D data.png](#)

The data is wired to the 2D array input and all the defaults are taken. This will ask for a file name when the program block is run, and create a file with data values, separated by tab characters, as follows:

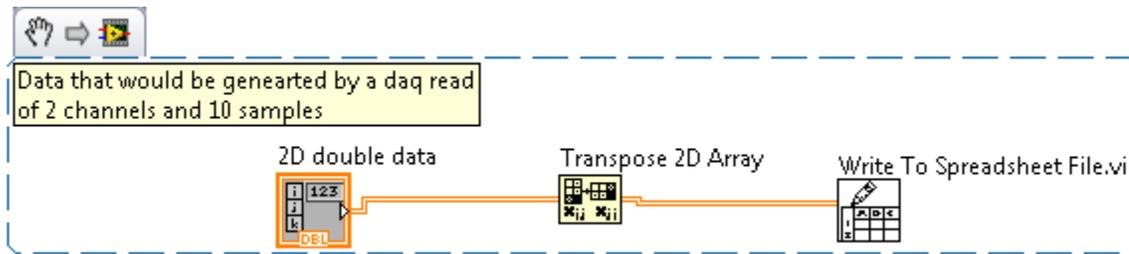
```
1.000 2.000 3.000 4.000 5.000 6.000 7.000 8.000 9.000 10.000
10.000 20.000 30.000 40.000 50.000 60.000 70.000 80.000 90.000 100.000
```

Note that each value is in the format x.yyy, with the y's being zeros. The default format for the write to spreadsheet VI is "%0.3f" which will generate a floating point number with 3 decimal places. If a number with higher decimal places is entered in the array, it would be truncated to three.

Since the data file is created in row format, and what you really need if you are going to import it into excel, is column format. There are two ways to resolve this, the first is to set the transpose bit on the write function, and the second, is to add an array transpose, located in the array pallet.



[transpose bit set.png](#)



[spreadsheet save with transpose.png](#)

The new data output now will look like:

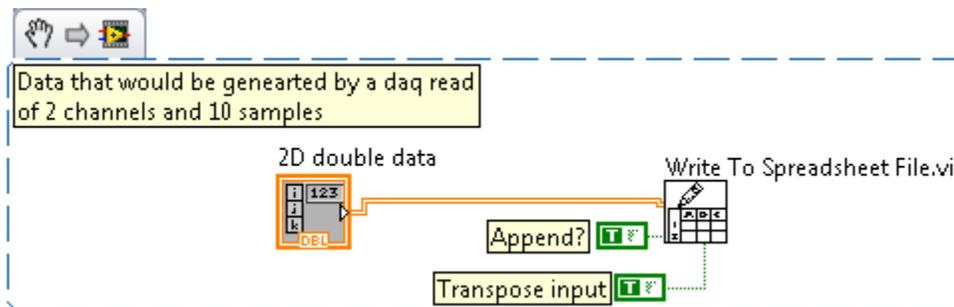
```

1.000  10.000
2.000  20.000
3.000  30.000
4.000  40.000
5.000  50.000
6.000  60.000
7.000  70.000
8.000  80.000
9.000  90.000
10.000 100.000

```

Note that it is now in column format with what was the first row, now in the first column. This gives you the first acquired channel in the first column and the second channel in the second column. This is a handy way to deal with the data in a spreadsheet. One problem with this format is that there are no labels or indicators to determine what is what in the file. We can address that in a moment.

Frequently you want the data to be saved in a file that already exists, adding to the data that is there. This function allows that by setting the "append" input to True.



[pngs/file_storage/transpose and append set.png](#)

Running this snippet against the file we created above gives the following output.

```

1.000  10.000
2.000  20.000

```

3.000 30.000
4.000 40.000
5.000 50.000
6.000 60.000
7.000 70.000
8.000 80.000
9.000 90.000
10.000 100.000
1.000 10.000
2.000 20.000
3.000 30.000
4.000 40.000
5.000 50.000
6.000 60.000
7.000 70.000
8.000 80.000
9.000 90.000
10.000 100.000

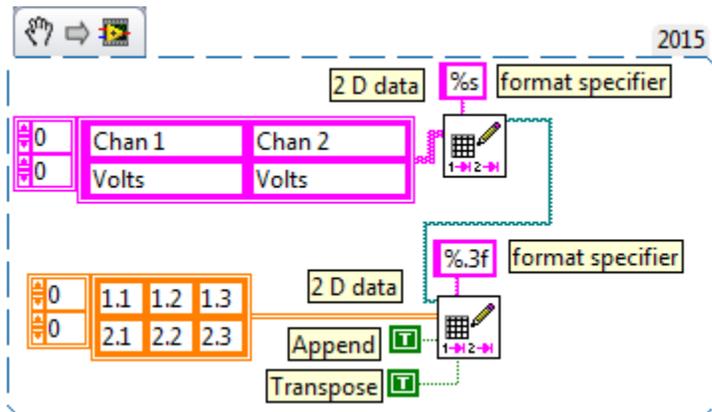
The real advantage to this method is that this VI is extremely efficient at doing the write function, making it a very fast way to save large amounts of data. As stated before, the largest drawback is that you have to know about the data, and remember it later. Because of this, a number of users prefer to deal with the files as raw text, which is what they really are.

Two methods exist to insert headings on the columns in the file. These are to use the spreadsheet write function in a separate write or to create a file using the text file functions. Both methods work equally well.

Using the spreadsheet write requires a two-step process. The first is to write the headers using a %s format string (meaning string values). For this to work the headers must be in array format, with one element for each column header. In the following two examples I have created identical headers, each with two rows of label. If you only wanted a single row of label you would only need to create a 1D array instead of a 2D array as I have done here.

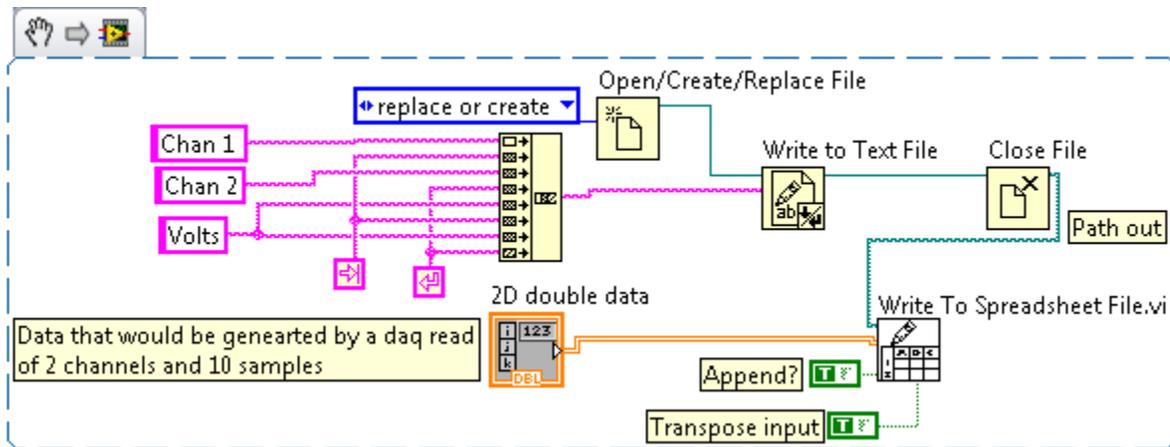
Example 1 output:

Chan 1	Chan 2
Volts	Volts
1.1	2.1
1.2	2.2
1.3	2.3



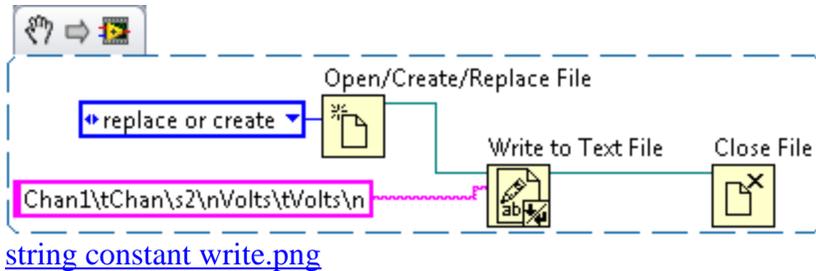
Write labels to spreadsheet.png

The second method is to create a text file, write the header contents to the file and then append the spreadsheet data to the file using the spreadsheet write. Note that in versions 2015 and later, the “Write to Spreadsheet File.vi” has been replaced with “Write Delimited Spreadsheet.vi”. For all practical purposes these function identically for writing numbers to the spreadsheet, though the newer write has more functionality that allows us to do what I have above with the labels.



[discrete string spreadsheet file save](#)

This example shows this as a discrete string build with a string concatenation, but it could be just as easily done with a single string constant as shown below. It is important to remember that “\t” is a tab character and the “\n” is a new line character. Also you must be sure that you have turned on the “\ codes display” option for the string constant, which is found in the right click menu.



The data file generated by either of these methods are the same, and shown below.

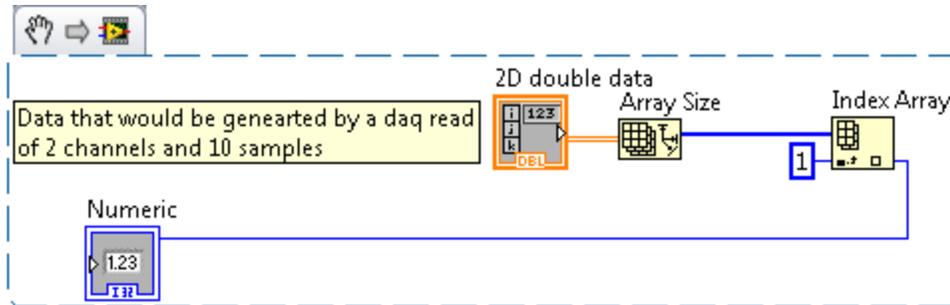
Chan1	Chan 2
Volts	Volts
1.000	10.000
2.000	20.000
3.000	30.000
4.000	40.000
5.000	50.000
6.000	60.000
7.000	70.000
8.000	80.000
9.000	90.000
10.000	100.000

This process works well for a large number of data points, but can also be used for a very small number of points. The only requirement is that there must be a full array, that being a data point in each row and column with no blanks. If you are using a single row of data you can use the 1D input to the write.

It is often advantageous to mix text and numbers in the same file, even more than just column headers. Dates, times and units labels mixed with the lines prevent the spreadsheet write method from being used. To accomplish this a different tactic must be used.

Most all files that are used for data storage are text files of some sort or another. There is nothing that prevents individual lines from being written as text for later use by other programs. This type of storage is typically done for either long term data or small amounts of data, since the process is very inefficient.

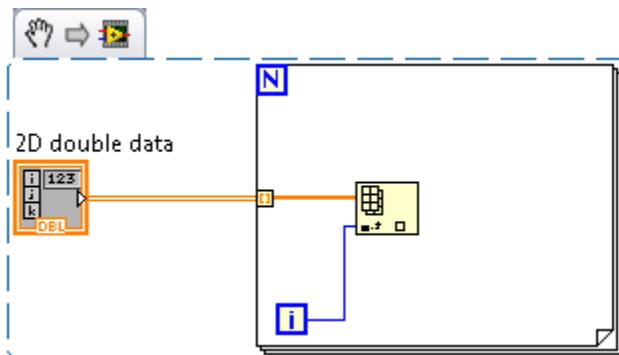
The first step in this method is to determine the size of the array. This is easily done using array size.



[read array size.png](#)

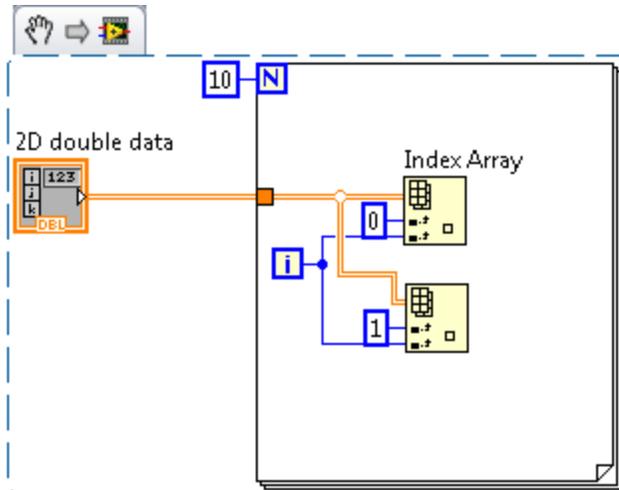
The output of the array size object is an array of integers, representing the size(s) of the array. In a 1D array, there is a single element that is the length of the array, however in a multi dimensional array, there is an array of sizes, each element of the integer array being one of the array dimensions. In the case of our sample, element 0 of the array is the number of channels and element 1 is the number of samples for each channel. The index array object allows us to break out the second element, which we will need to process the array.

In the following snippet I have dropped an index array object into a for loop and wired it to the 2D array. LabVIEW does something here that will bite you if you're not careful. Notice how the wire changes from a double wide line to a single heavy line and there is a small square with brackets inside where the line crosses into the for loop. LabVIEW is trying to be helpful, and realizing that it is an array coming in, it will automatically index the array.



[for loop indexed array.png](#)

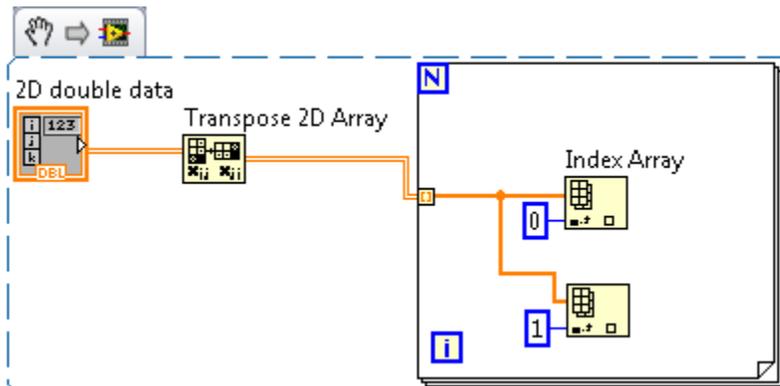
Automatic indexing is a feature that can be very useful, and can also be a real pain. In the above snippet, the loop count (the little N box) is set to the first dimension of the array, in this case 2. For each iteration of the loop, the line inside the box (orange 1D array line) will have the 10 elements associated with the first row of the array. Nothing else of the array is visible inside the for loop. The second iteration will have the second row of the array, and if there were more rows, this would continue until the array was out of rows. Since we need to see both channels at the same time, we have some choices. We can turn off the indexing, by right clicking on the box and selecting "Disable Indexing". This will turn the square solid orange and the entire array is now visible inside the for loop. Once this is done, we can then go through the array line by line and pull out each individual element as shown below.



[indexed by two objects.png](#)

Each pass of the array will pull out the two channels from the array with the index array objects. This is a simple and straightforward way to accomplish this task. It is easy to read at a later time and pretty obvious to anyone who is working with the program later just what is being done.

A slightly more elegant method makes use of auto indexing.

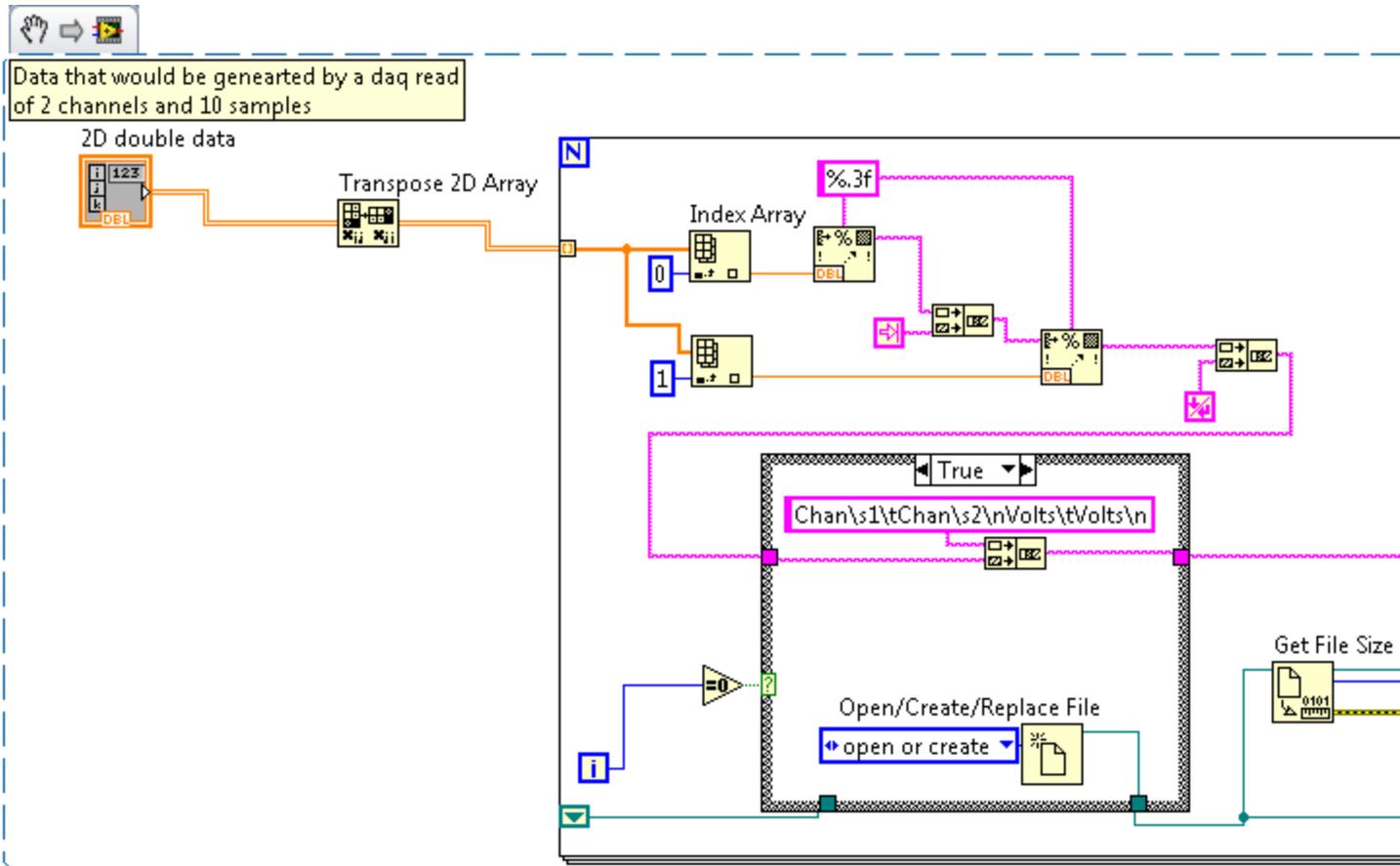


[auto_index_array.png](#)

In this example, the for loop is iterated ten times, based on the number of rows in the transposed array. For each row, the index array objects pull out the channel 1 and channel 2 data, which could then be fed to some other function. In this case we are going to convert it to a string and write it to a file. The entire program section is shown below.

While the technique is used here to write a single 2D array, it is easy to apply this technique to an application where the data is taken at long intervals and written to a text file so the data is not lost in the event of a computer hiccup or gurgling cringing death....

The data output is shown following the program snippet.



[save array to text file.png](#)

Chan 1	Chan 2
Volts	Volts
1.000	10.000
2.000	20.000
3.000	30.000
4.000	40.000
5.000	50.000
6.000	60.000
7.000	70.000
8.000	80.000
9.000	90.000
10.000	100.000

The file storage portion consists of three blocks, one which is run only on the first iteration, and the other three that are run every iteration. The initial function, "open/Create/Replace file" is used without any path input to cause the prompt for a file. This will then create the file if it is not

in existence, or open it if it is. This is located inside a case statement that executes only when the iteration counter is 0. In addition it appends the file header information to the string of data values. Once this has been done a value known as a reference number, which is how the computer refers to the file, is passed to a block which gets the file size. If the file is a new file, the size is 0. If not, it returns the total number of bytes in the file.

The third block in the string is a "set File Position". This block moves a pointer in the file to a particular position. In our case we have wired the output of the Get File Size to the input of the Set File Position so that the pointer is now pointing to the end of the file. The last block is a Write to Text File. This block begins writing the text string wired to its input at the location that the pointer is at, in our case, the end of the file. This set of blocks effectively appends the string to the existing file, even if that length is zero. The reference number is sent to a shift register so it is available for the next iteration of the loop.

In the next and all subsequent iterations of the for loop, the iteration counter is not equal to 0, so the FALSE case is executed, which does nothing but pass the string and reference number values back on out to the program.