Contents

[Reading PSCAD Memory arrays: 2](#_Toc199322839)

[Approach-1: 2](#_Toc199322840)

[Approach-2: 3](#_Toc199322841)

[Approach-3: 6](#_Toc199322842)

[Approach-4: 8](#_Toc199322843)

[Approach-5: 9](#_Toc199322844)

Making a DLL work with snapshot and multi-instancing is (mostly) the same for locally included C, and for DLLs, The key to both is to make your Code “Stateless”

Stateless design of DLL is simply done through ensuring that all of the variables that persist from one time step to another are stored in the Storage arrays of provided by PSCAD.

The important thing is that all the ‘data’ that is needed for your code to run, must be stored in the storage arrays.

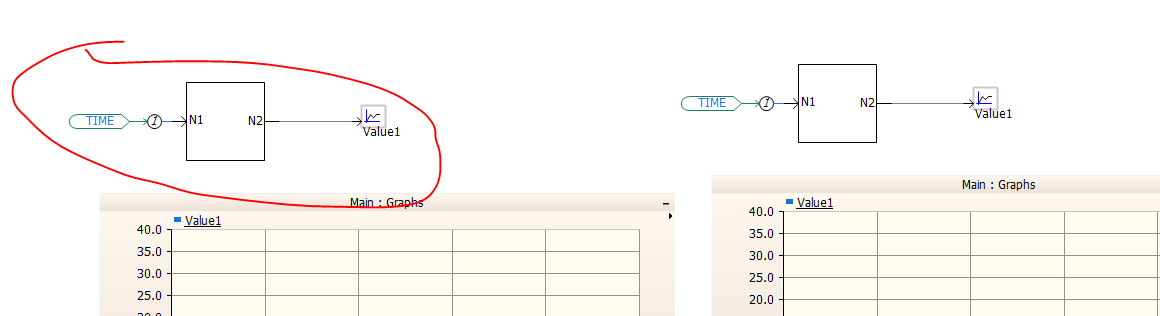
The following examples I show, show how to write the component stateless in ‘C/C++’ however all of the techniques shown here also apply to working with DLLs.

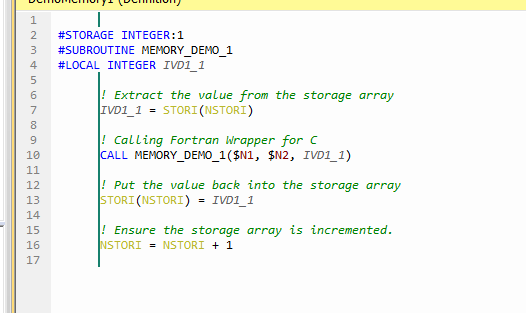
## Reading PSCAD Memory arrays:

### Approach-1:

The model Example\_1.zip shows this being done for 2 very simple cases.

The First way is to simply keep track of all variable in the storage arrays completely on the component side:





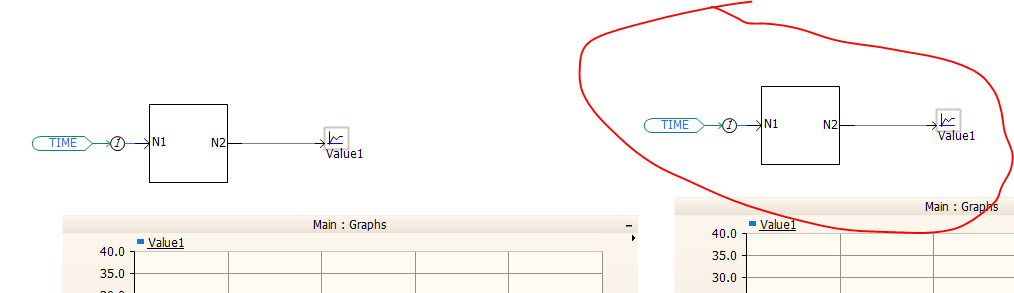
In here we ensure that all information required by the function MEMORY\_DEMO\_1 is entirely provided and passed in by the parameter. Then inside the component we simply store the value in the storage arrays.

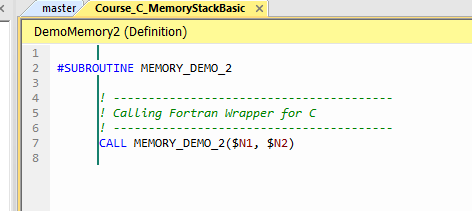
This is a good approach if you do not have very many variables being stored, and all of the variable are easily passed in via parameters.

The down sides to this, is if there are any more values that need to be stored between time-steps (or in the snapshot file) then you will need to edit out C file to include those changes, the Fortran wrapper file to include those changes (the additional parameter) and the component in PSCAD itself to include those changes. This can be a real annoyance if you are not exactly sure what is going to need to be stored and what isn’t.

### Approach-2:

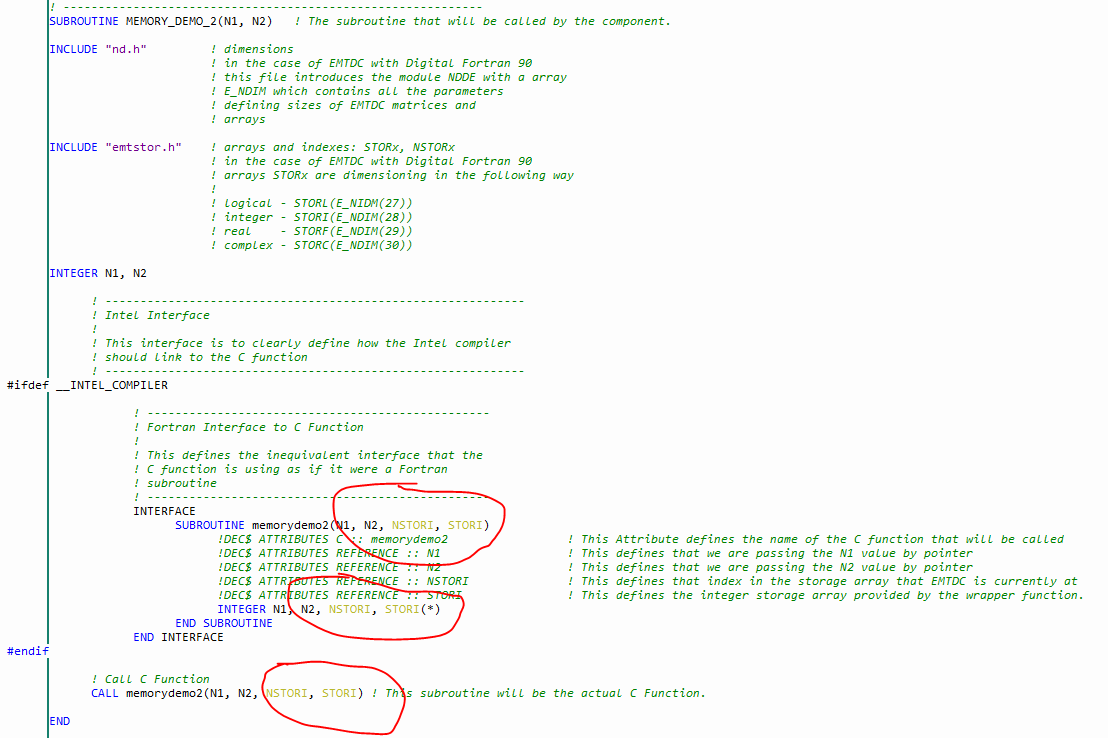
The second demo is to pass the storage array in its entirety into the C function, and pull the values out from there.





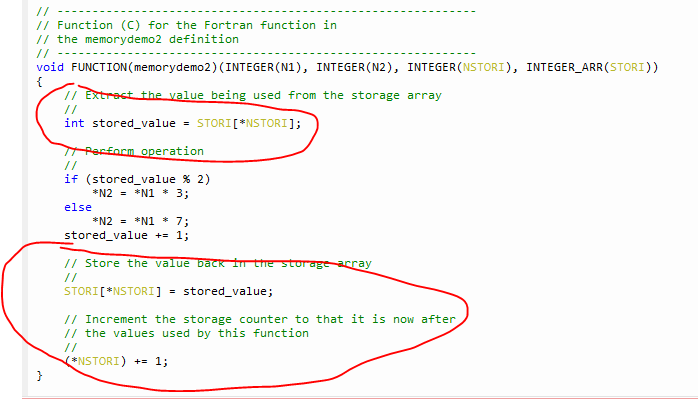
In this demo we do not pass any additional information other than the inputs and output from the component.

However when checking the Fortran wrapper file:…



You can see that we are passing in the entire storage array to the C function.

we can then pull and push values to the storage array directly in the C



This allows for simple management of the storage to be entirely performed within the C code.

You will need to pass in each storage array that you wish to pull/push values from along with their current indexes ( {STORL,NSTORL} {STORI, NSTORI}, {STORF,NSTORF}, {STORC,NSTORC} )

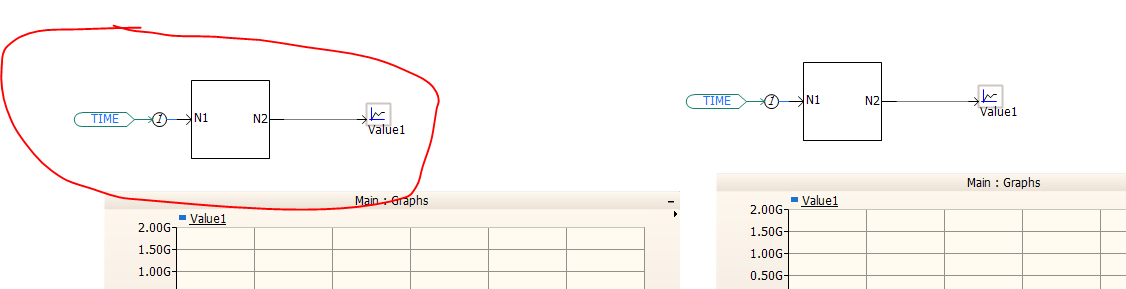
This works fine again if the storage requirements are fairly simple and only require storing Boolean/Integers/Doubles/Complex numbers.

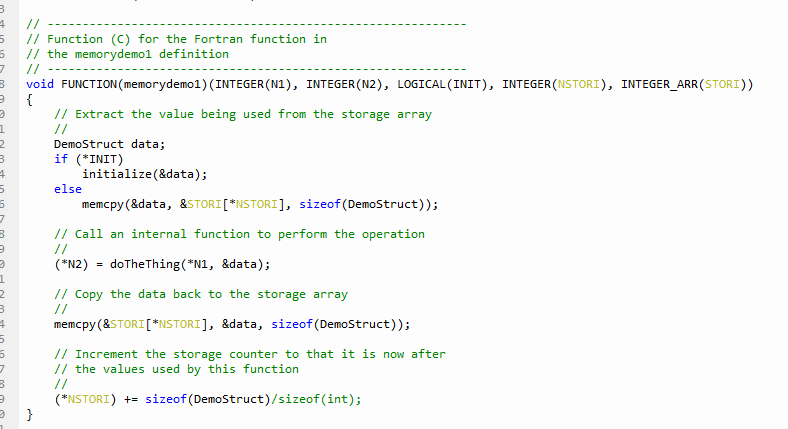
### Approach-3:

There are however some more advanced techniques for storing larger amounts of more complicated types of data.

The model Example\_2.zip shows 2 different ways to store large amounts of more complicated data:

In the first example here we pass only the integer storage array (STORI, NSTORI) no other storage array is required. If this is TIMEZERO or not is also required to be passed in for initialization purposes.





In this example an entire structure of multiple different types is saved to the Integer Storage array by using memcpy to move all of the memory in and out of the storage array at once. Due to the well defined binary states of Integers, there is no data loss regardless of the type of type being stored in it.

This is useful for larger amount of data, and very complicated data, as the memcpy simply scales with the size of the structure.

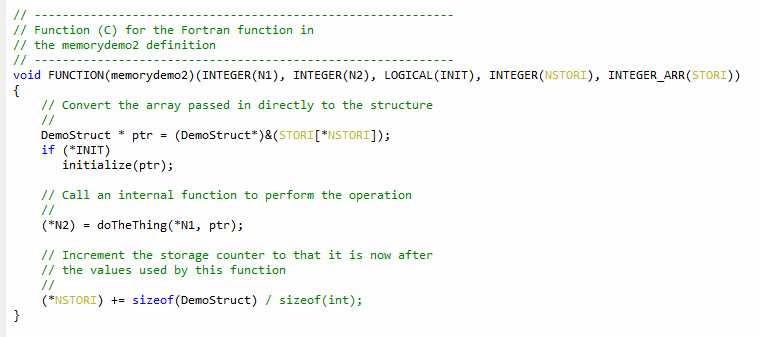
However if the structure gets very large then the time to copy the data may become very expensive (take a long time)

Another limitation of this is that the structure stored must be ‘Flat’ that means that it cannot have any internal pointers inside it. The structure is not allowed to have pointers

### Approach-4:

The second demo in this example shows how do we perform the same thing as the first, but keep the data in place, instead of coping it.





This is a much faster way of doing the same thing in demo 1 of this example, however this may also more ‘dangerous’ as if any overruns occur you may corrupt memory in the rest of the stack. You just need to be aware and not cause any buffer overruns in your code.

In simple form, after the below code, assign ptr values to the local variables and perform operations on it and then store the updated variables to ptr as below.

DemoStruct\* ptr = (DemoStruct\*)&(STORI[\*NSTORI - 1]);

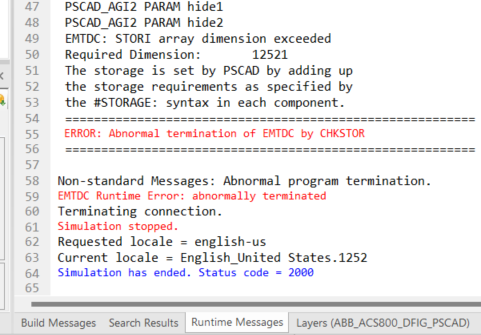
Valuei = ptr->iValue;

Valuei = Valuei + 1;

Ptr->iValue = Valuei;

(\*NSTORI) += sizeof(DemoStruct) / sizeof(int);

If you have lot more variables, which are to be stored, defined in structure, then there is a chance of getting the storage limit exceed error as below:



Then Adding storage\_interger to the component definition as below skips the above error message.

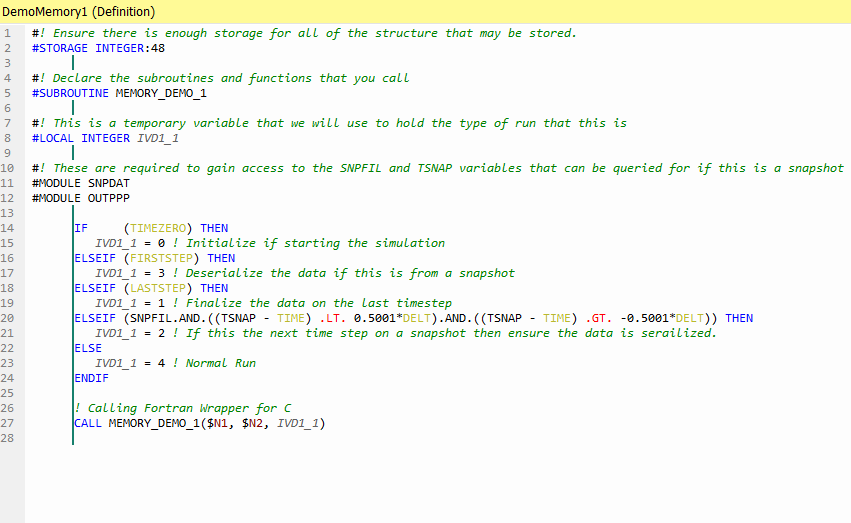


### Approach-5:

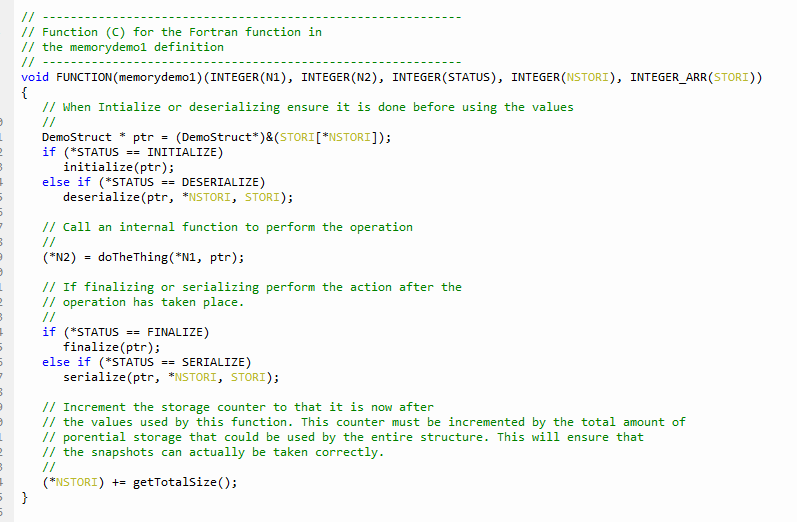
The model Example\_3.zip here shows how to deal with non-flat structure, (structure which are required to store pointers)

This demo is a lot more complex than the others ones as it requires you to take more into account

 In this demo, inside the component you need to discover if this time-step you are Initializing, Running as normal, Loading from a snapshot, or saving to a snapshot. Each of these options will trigger something different in the DLL



On the ‘C’ side we have this:



Where if we are initializing or reserializing we need to perform those actions first, and if we are finalizing or serializing then we perform that after.

The serialize function take the structure and convert it into a flat construct that can be written to the storage arrays

The deserialize function does the opposite reading from the storage array, and generating new pointers to the areas which are needed.

Regardless of if you are performing a serialize/desterilize operation, you need to keep the storage array amount reserved for performing the operations. So enough memory will need to be allocated in the integer storage array to store the entire contents of your DLL storage requirements