ARTICLE - Enhancing Liberty Basic Array Handling

(AKA: The memory Article)
by Dennis McKinney -
DennisMcK

originally in Liberty BASIC Newsletter #99, August, 2002

Liberty Basic is very easy to learn and use. Unfortunately this fact tends to make some users feel that it's too simple, and that a lot of things simply "can't be done" because the language doesn't support this or that. This isn't necessarily the case. For example, arrays of more that two dimensions, mixed type arrays with both string and numerical elements in them, and arrays of structures are not supported. Several times in the past couple of years I have had need for these kind of arrays, so finally I decided to try to find a solution. As it turns out, the solution is pretty simple.

Liberty Basic has a very powerful capability built right in, the ability to utilize the Windows API. This ability proved to be the answer and allows all three of these array types to be created with very little effort. It only takes four API calls and a little unusual usage of an array and a structure. Think about what a variable, array, or structure really is. Each one is a certain number of bytes in memory that are reserved, or allocated, to contain values. These bytes of memory are the same regardless of which language the programmer is using to fill in the values. The Windows API provides the means to allocate memory and Liberty Basic has the means to utilize this memory.

API Memory Functions

Three of the four APIs we will be calling are wrapped in Liberty Basic functions. These will be called:

```
'Function GlobalAlloc( dwBytes )
'Function GlobalLock( hMem )
'Function GlobalFree( hMem )
```

Each one will be explained as it appears in this example.

Array of Structs

Let's begin with an array of structures. As it turns out, all three types of arrays can be created as an array of structs. For this example we'll use an array 100 structures, and each structure will have 3 elements, 2 strings and one number. This array will not actually contain structures. The structures are going to be

stored in memory. The purpose of the array is to store the address of each structure that we store in memory.

```
'dimension the array for 100 addresses
    elements = 99
    dim structArray(elements)
'define one structure
    struct test,_
    a as char[20],_
    b as long,_
    c as char[20]
```

Notice that the string elements are typed as char[x], where x is the length of the string. Using a$ as ptr will not work.

Amount of Memory

When you are going to allocate memory the first thing you need to determine is how much memory you need. This is done by multiplying the number of elements in the array by the size of the structure.

```
'the size of the structure is
    sizeofTest = len(test.struct)
'the amount of memory needed is
    memBlockSize = (elements + 1)*sizeofTest
```

The memory API's are accessed thought kernel32.dll so,

```
open "kernel32.dll" for dll as #kernel
```

note: opening kernel32.dll is no longer necessary. Liberty BASIC recognizes #kernel32

Allocate Memory

Now we can allocate the memory needed

```
hSArray = GlobalAlloc( memBlockSize )
```

This function allocates the memory and returns a handle to the memory object. After this call hSArray will contain the handle for the memory. Code for checking for a valid handle (not null) isn't included here. The function takes one argument, the amount of memory being requested.

Ok, so we've created some memory, but where is it? This next function will return a pointer to the first byte of the memory block, which is the address of the start of the memory block. The function takes the handle of the memory object as its argument.

```
ptrSarray = GlobalLock( hSArray )
```

Let's review what has been done so far.
1. Dimension an array for pointers to memory addresses.
2. Defined the struct for our data.
3. Allocated the needed memory.
4. Determined where the memory is located within the heap.

Fill Structs with Data

The memory and structure array are ready to be used, so for example purposes we'll fill all of the structures with data and store them in memory. As each struct is filled it will be placed in the memory one after the other. To place a struct into memory we'll call the RtlMoveMemory API. This API needs to know three things:

1. Dest, where to put the data into memory.
2. Src, address of the memory block to copy, in this case the address of the test structure.
3. dwLen, the size, in bytes, of the block to copy, in this case the size of the test structure.

The destination is determined as an offset from the first byte of the allocated memory. If each structure we were copying were 10 bytes long, the first struct would be stored starting at ptrSarray, which is the first byte of the memory, the second struct would be stored starting at 11 bytes into the memory block, etc. Liberty Basic takes care of the second requirement (Src) by passing a pointer to our stuct. We have already determined the size of our structure (sizeofTest).

```
'for example purposes, fill the whole array of structures
     for i = 0 to 99
```

```
     'put some data into the struct
     test.a.struct = "Carol - " + str$( i)
     test.b.struct = i
     test.c.struct = "Andy - " + str$( i)
     'calculate the destination as an offset from the
     'first byte
     dest = ptrSarray+(i*sizeofTest)
     'put the structure into memory
     CallDll #kernel,"RtlMoveMemory", dest as long, _
     test as ptr, sizeofTest as long, ret as void
     'store the address so the data can be found when needed
     structArray(i) = dest
     next i
```

Retrieving Data

Now that we've put 100 structures into memory, how do we get the data back from memory? Again, Liberty Basic provides the means. Simply point our structure to the address of the memory we want. Remember that the addresses were stored in the structArray() so we just do this: test.struct = structArray(i), where i is the element we want.

```
'for example, read all of the structures
     for i = 0 to 99
     test.struct = structArray(i)
     A$ = test.a.struct
     B = test.b.struct
     C$ = test.c.struct
     print A$ + " " + str$(B) + " " + C$
     next i
'To retrieve the third element from the 50th structure:
     test.struct = structArray(49)
     C$ = test.c.struct
     print C$
'To change the value of the third element of the 50th
     'structure:
     test.struct = structArray(49) 'get the structure
     test.c.struct = "Changed" 'change one or more values
     'save it
     dest = ptrSarray+(49*sizeofTest)
     CallDll #kernel,"RtlMoveMemory", dest as long, _
     test as ptr, sizeofTest as long, ret as void
 'just for example
     test.struct = structArray(49)
```

```
     C$ = test.c.struct
     print C$
```

Free Allocated Memory

The final and very important thing to do when your program ends is to free every memory that we allocated from the heap. This erases all the structs that we stored and frees the memory for use.

```
[quit]
     ret = GlobalFree(hSArray)
'call this for every memory block allocated.
    'Use the appropriate memory handle.
     close #kernel
     end
'**** Functions ****
     Function GlobalAlloc( dwBytes )
     'returns the handle of the newly allocated memory object.
     'the return value is NULL if fail.
     CallDll #kernel, "GlobalAlloc", _GMEM_MOVEABLE as long,_
     dwBytes as ulong, GlobalAlloc as long
     End Function
Function GlobalLock( hMem )
     'returns a pointer to the first byte of the memory block.
     'the return value is NULL if fail.
     CallDll #kernel, "GlobalLock", hMem as long, _
     GlobalLock as long
     End Function
Function GlobalFree( hMem )
     CallDll #kernel, "GlobalFree", hMem as long, _
     GlobalFree as long
     End Function
```

The complete example is contained in the demo below.

In closing I'll leave you with this. Gee, I wonder. Could this method be used with API calls that require arrays of structures?

Demo

```
    'dimension the array for 100 addresses
```

```
    elements = 99
    Dim structArray(elements)

    'define one structure
    struct test,_
    a As char[20],_
    b As long,_
    c As char[20]

    'the size of the structure is
    sizeofTest = Len(test.struct)

    'the amount of memory needed is
    memBlockSize = (elements+1)*sizeofTest

    Open "kernel32.dll" For DLL As #kernel

    hSArray = GlobalAlloc(memBlockSize)

    ptrSarray = GlobalLock( hSArray )

'for example purposes, fill the whole array of structures
    For i = 0 to 99
        'put some data into the struct
        test.a.struct = "Carol - " + Str$(i)
        test.b.struct = i
        test.c.struct = "Andy - " + Str$(i)
        'calculate the destination as an offset from the first byte
        dest = ptrSarray+(i*sizeofTest)
        'put the structure into memory
        CallDLL #kernel,"RtlMoveMemory", dest As long, test As ptr,
 sizeofTest As long, ret As void
        'store the address so the data can be found when needed
        structArray(i) = dest
    Next i

'for example, read all of the structures
    For i = 0 to 99
        test.struct = structArray(i)
        A$ = test.a.struct
        B = test.b.struct
        C$ = test.c.struct
        Print  A$ + "   " + Str$(B) + "   " + C$
    Next i

'To retrieve the third element from the 50th structure:
```

```
    test.struct = structArray(49)
    C$ = test.c.struct
    Print C$


'To change the value of the third element of the 50th structure:
    test.struct = structArray(49) 'get the structure
    test.c.struct = "Changed" 'change one or more values
    'save it
    dest = ptrSarray+(49*sizeofTest)
    CallDLL #kernel,"RtlMoveMemory", dest As long, test As ptr,
 sizeofTest As long, ret As void

    'just for example
    test.struct = structArray(49)
    C$ = test.c.struct
    Print C$

    input a$


[quit]
    ret = GlobalFree(hSArray)
'call this for every memory block allocated.
    'Use the appropriate memory handle.
    Close #kernel
    End

'**** Functions ****
Function GlobalAlloc( dwBytes )
'returns the handle of the newly allocated memory object.
'the return value is NULL if fail.
    CallDLL #kernel, "GlobalAlloc",  _GMEM_MOVEABLE As long,
 dwBytes As ulong, GlobalAlloc As long
End Function

Function GlobalLock( hMem )
'returns a pointer to the first byte of the memory block.
'the return value is NULL if fail.
    CallDLL #kernel, "GlobalLock", hMem As long, GlobalLock As long
End Function

Function GlobalFree( hMem )
    CallDLL #kernel, "GlobalFree", hMem As ulong, GlobalFree As long
End Function
```