

--- ThunderVB Development Team ---
<http://www.thundervb.sf.net/>

Functions __vbaCopyBytes and __vbaCopyBytesZero

By Libor Blaheta

(Edited By Tai Chi Minh Ralph Eastwood)

Last Update 30-09-2005

Table of Contents

Abstract	3
Function __vbaCopyBytes	4
Function __vbaCopyBytesZero	6
Speed tests	8
Conclusion	9

Abstract

When you look at the functions that *msvbvm60.dll* exports, you will see many interesting names. Well known are the *VarPtr* and the set of *GetMem* and *PutMem* functions. However, if you search deeper, you will also spot functions with names *__vbaCopyBytes* and *__vbaCopyBytesZero*. If you disassemble them, you will notice these functions are quite short and their names do suggest that they could be useful.

I have extracted the code of *__vbaCopyBytes* and *__vbaCopyBytesZero* from a disassembler and then pasted their code to VB as *InlineAsm*. In this essay I will briefly explain how these functions work and what their purpose is.

Note: *free disassemblers available are - Borg, W32Dasm, HView or IDA*

Please note that this is my first attempt to write a technical article, so please be kind. I promise my next essay will be better and more interesting than this one. ☺

Function __vbaCopyBytes

The function `__vbaCopyBytes` copies a block of memory from one location to another. If you know API `CopyMemory`, this function does exactly the same. Now let's take a look at its code.

```
'Colored by ThunderIDE+ 1.0.0
Public Sub vbaCopyBytes(ByVal Length As Long, ByVal Destination As Long,
ByVal Source As Long)

'#asm'
'#asm'  mov ecx, [esp+4]           ;1 - ecx = 1.param
'#asm'  push esi                 ;2 - save esi
'#asm'  mov esi, [esp+16]        ;3 - esi = 3.param
'#asm'  push edi                 ;4 - save edi
'#asm'  mov edi, [esp+16]        ;5 - edi = 2.param
'#asm'  mov eax, ecx             ;6 - eax = ecx
'#asm'  mov edx, edi             ;7 - edx = edi
'#asm'  shr ecx, 2               ;8 - ecx = ecx/4
'#asm'  rep movsd                ;9 - perform dword-copy
'#asm'  mov ecx, eax             ;10 - ecx = eax
'#asm'  mov eax, edx             ;11 - eax = edx
'#asm'  and ecx, 3               ;12 - ecx = ecx mod 4
'#asm'  rep movsb                ;13 - perform byte copy
'#asm'  pop edi                 ;14 - restore edi
'#asm'  pop esi                 ;15 - restore esi
'#asm'  retn 12                 ;16 - return
'#asm'

End Sub
```

As you can see the function is pretty simple. The function has three parameters:

- *Length* – number of bytes to copy
- *Destination* – pointer to the destination memory
- *Source* – pointer to the source memory

Now, let's explain the code. The copying is based on the fast Asm instructions “*rep movsd*” and “*rep movsb*” (see the line 9. and 13.). These instructions perform byte/dword-copy from one memory location to another. Before you call these instructions you have to set the registers **EDI**, **ESI** and **ECX**. **EDI** should point to the destination memory; **ESI** to the source memory and **ECX** is number of bytes to copy.

Note: Notice that the function doesn't modify the Direction Flag.

So at first the function sets **EDI**, **ESI** and **ECX** to the passed values. Then **ECX** is divided by 4 (see the line 8.) Since we are going to use a dword-copy, “*rep movsd*” instruction is called. This instruction will perform dword-copy. This will be OK, if the number of bytes we want to copy has no remainder after dividing by 4. (eg. 20 or 36 bytes can be copied by “*rep movsd*” but not 21 or 37)

So you need to know the remainder. The function uses instruction “*and*” to get the remainder (see the line 12.) (Note: The arithmetic “*and*” operator acts like the mod operator when ‘modding’ by a multiple of 2). And then it performs byte-copy by calling the instruction “*rep*

movsb” (see the line 13.). This instruction will copy the rest of bytes. It’s clear the remainder can be only 0 or 1 or 2 or 3.

Function `__vbaCopyBytesZero`

The function `vbaCopyBytesZero` is interesting as well. It copies a block of memory from one location to another and then it fills the source block of memory with zeros. You can achieve the same functionality by calling `CopyMemory` and `ZeroMemory`. Again, let's study its code.

```
'Colored by ThunderIDE+ 1.0.0
Public Sub vbaCopyBytesZero(ByVal Length As Long, ByVal Destination As
Long, ByVal Source As Long)

'#asm'
'#asm'  push ebp                ;1 - save ebp
'#asm'  mov ebp, esp            ;2 - ebp = esp
'#asm'  mov ecx, [ebp+8]        ;3 - ecx = 1.param
'#asm'  push esi                ;4 - save edi
'#asm'  mov esi, [ebp+16]       ;5 - esi = 3.param
'#asm'  mov eax, ecx            ;6 - eax = ecx
'#asm'  push edi                ;7 - save edi
'#asm'  mov edi, [ebp+12]       ;8 - edi = 2.param
'#asm'  shr ecx, 2              ;9 - ecx = ecx/4
'#asm'  rep movsd               ;10 - perform dword-copy
'#asm'  mov ecx, eax            ;11 - ecx = eax
'#asm'  and ecx, 3              ;12 - ecx = ecx mod 4
'#asm'  rep movsb               ;13 - perform byte-copy
'#asm'  mov edi, [ebp+16]       ;14 - edi = 3.param
'#asm'  mov ecx, eax            ;15 - ecx = eax
'#asm'  mov edx, ecx            ;16 - edx = ecx
'#asm'  xor eax, eax            ;17 - eax = 0
'#asm'  shr ecx, 2              ;18 - ecx = ecx/4
'#asm'  rep stosd               ;19 - store eax to [edi]
'#asm'  mov ecx, edx            ;20 - ecx = edx
'#asm'  and ecx, 3              ;21 - ecx = ecx mod 4
'#asm'  rep stosb               ;22 - store eax to [edi]
'#asm'  mov eax, [ebp+12]       ;23 - eax = 2.param
'#asm'  pop edi                 ;24 - restore edi
'#asm'  pop esi                 ;25 - restore esi
'#asm'  pop ebp                 ;26 - restore ebp
'#asm'  retn 12                 ;27 - return
'#asm'

End Sub
```

As you can see the function above is a bit longer than the previous one. But still it's easy to understand. The function's parameters have same purpose as the function above.

- *Length* – number of bytes to copy
- *Destination* – pointer to the destination memory
- *Source* – pointer to the source memory

The code is quite similar to the code of `__vbaCopyBytes` so I won't dive into the details very much. First of all a dword-copy (see the line 10.) is performed and then a byte-copy (see the line 13.) is done for the remainder. We have already seen this; `vbaCopyBytes` uses the same method for copying.

The remaining code zeroes the source memory. Let's explore it.

The function uses the instructions “*rep stosd*” and “*rep stosb*”. Before you can call them you have to set the registers **EDI**, **ESI** and **EAX**. **EDI** should point to the destination memory; **ESI** to the source memory and **EAX** is a value that will be stored to location at **EDI**. In our case it’s 0.

The “*dividing by 4 trick*” is used once again. So, first of all “*rep stosd*” is called to zero the memory up to a multiple of 4 and then “*rep stosb*” is called for the remaining bytes.

Speed Tests

Maybe, you're wondering whether the function `__vbaCopyBytes` is faster than `CopyMemory` and `__vbaCopyBytesZero` than `CopyMemory` and `ZeroMemory`. I did several speed tests and I have to say both functions are faster than `CopyMemory` and `ZeroMemory`. Here, I'm not showing any speed-test results, however, along with this white paper is shipped a VB app that will perform speed tests. Try it on your PC to check what is faster.

And how `CopyMemory` and `ZeroMemory` work? That's a long story... ☺
You can disassemble `kernel32.dll` and study the code of `RtlMoveMemory`.

Note: *VB programmers know this API as `CopyMemory`.*

The code is not difficult to understand but it's longer than the code of `__vbaCopyBytes`. `RtlMoveMemory` uses same method for copying - at first "*rep movsd*" then "*rep movsb*" - as `vbaCopyMemory` does. But `RtlMoveMemory` has one advantage to `__vbaCopyBytes`. It can handle overlapping blocks of memory. `__vbaCopyBytes` doesn't support this and that's why it's slightly faster.

Also the function `RtlZeroMemory` uses similar method for zeroizing a memory as `__vbaCopyBytesZero` does. You can simulate the functionality of `__vbaCopyBytesZero` by calling `CopyMemory` and `ZeroMemory`. And guess what is faster?

Yes, `__vbaCopyBytesZero` is faster.

Also `__vbaCopyBytesZero` doesn't handle overlapping blocks of memory and calling single function is faster than calling two functions.

Conclusion

As you can see the dll library msvbvm60.dll (and msvbvm50.dll as well) is full of secrets. Well, some functions were published so many people know them, to name some VarPtr, GetMem, PutMem.

In msvbvm60.dll there're a lot of functions that have interesting names, what about EbGetErrorInfo, TipInvokeMethod or EbLoadRunTime.

What is their purpose and what do they do? Nobody knows... well actually, Microsoft knows but they won't tell it us. Only reverse engineering can uncover the secrets of VB Internals. So get a good disassembler and let's start...