

# Hidden Gems for Free

## Exposing undocumented memory access functions in Visual Basic 6

Michel Rutten

### Pop Quiz

*Do you...*

- *need flexible but efficient memory access in your Visual Basic programs?*
- *want to dereference pointer values in pure VB?*
- *want to cast instead of coerce?*
- *become tired of slashing with a block moving sledgehammer at a single DWord only because you have no better suited tool?*
- *get disappointed by VB's total lack of support for these kind of operations?*
- *sometimes find yourself longing back to GWBASIC, because at least it had `Peek` and `Poke`?*

If you have responded *Yes!* to any of these questions, then this article might relieve the pain a bit...

## Contents

[Pop Quiz](#)

[Contents](#)

[1. Introduction](#)

[2. The VB Runtime Library](#)

[2.1 Hidden Gems](#)

[2.2 GetMem](#)

[2.3 PutMem](#)

[3. Declarations](#)

[4. The Type Library](#)

[4.1 Basic declarations](#)

[4.2 Property syntax](#)

[4.3 Casting](#)

[4.4 Structures](#)

[4.5 Untyped assignment](#)

[4.6 VarPtr tricks](#)

[4.7 Goodies](#)

[5. License](#)

[6. Conclusion](#)

[Notes](#)

[Copyright](#)

## 1. Introduction

Technically savvy Visual Basic programmers for long have had to deal with VB's limited type system and lack of support for casting and pointer manipulation. Although type-safety generally is a very good thing, when you need to call the windows API or optimize a piece of code the strict type checking mechanism only tends to get in the way. Unfortunately, apart from a couple of unsupported low-level statements (i.e. `VarPtr`, `ObjPtr`, `StrPtr` and `AddressOf`) the language simply does not like untyped data and does a good job of discouraging you to use it. Most VB programmers eventually learn to live with this and unwillingly resort to the far from elegant but seemingly omnipresent abuse of an otherwise obscure API function exported by `kernel32.dll`, the discovery - and for that matter also first application in this context - of which should be contributed to [Hardcore Visual Basic](#) author and (classic) VB guru [Bruce McKinney](#), although the entry point is actually quite useful for VB programmers when used to do what it was meant for in the first place.

Readers who haven't got the slightest clue which particular function is being referred to are advised to at least *try* to get with the program first and return to this article only if they still have an healthy appetite for more delicatessen.

However if you are into VB gourmet then you are invited to stray from the path layed out for us mortals by the code gurus on mount Olympus (sorry Bruce, no hard feelings) and join in this copious feast of strange fruits and the likes. As most exotic food, it may take some time getting used to, so don't let that slight nausea at the third three finger salute turn you down, but hang in there and keep trying - carefully that is. Ultimately, you will be rewarded with both fast and elegant code, as well as a bit of insight into the intestines of your favourite programming environment.

## 2. The VB Runtime Library

### 2.1 Hidden gems

All Visual Basic programs require and reference an associated run-time library. For Visual Basic 6 this file is named **msvbvm60.dll**<sup>1</sup>. Usually it is located in the windows system folder. This file contains the compiled code and type library declarations for the built-in standard objects belonging to the **VBA** and **VBRUN** namespaces, as the Object Browser description clearly states on the status label when selecting them from the combobox. Apart from the regular VB objects, the runtime library also exports quite a lot of `stdcall` functions that are apparently necessary for executing VB code, as one can see by examining it's export table using the dependency viewer.

Because any VB application per definition requires this dll, any extra functionality we may get out of this library we get *for free*, i.e. without adding an extra external dependency to the project. Consequently, a scan through the list of exported functions might just reveal a couple of hidden gems waiting for us to get our dirty fingers on. It turns out that most of the exported functions are of little use to us, either because their names suggests that they implement a trivial VB command or because it is unclear what operations they perform and what arguments they expect. However there are a couple of members that clearly stand out because of their descriptive and provocative function names. They are exported as:

- |                  |                  |
|------------------|------------------|
| • <b>GetMem1</b> | • <b>PutMem1</b> |
| • <b>GetMem2</b> | • <b>PutMem2</b> |
| • <b>GetMem4</b> | • <b>PutMem4</b> |
| • <b>GetMem8</b> | • <b>PutMem8</b> |

Their names suggest that these functions read or write a value of the indicated size from or to memory, offering an interesting prospect at the very least. Do these functions indeed offer direct memory access? And if so, is it possible to call them from Visual Basic? Since there is no documentation available, the best way to make sure is by generating and inspecting an assembly listing. Mind you the EULA strictly forbids this... so get ready to go underground and let's take a peek<sup>2</sup>.

## 2.2 GetMem

This is the assembly listing for the GetMem functions:

GetMem1	GetMem2	GetMem4	
mov eax, [esp+4]	mov eax, [esp+4]	mov eax, [esp+4]	; Get first argument
mov ecx, [esp+8]	mov ecx, [esp+8]	mov ecx, [esp+8]	; Get second argument
mov al, [eax]	mov ax, [eax]	mov eax, [eax]	; Dereference first
mov [ecx], al	mov [ecx], ax	mov [ecx], eax	; Store value in second
xor eax, eax	xor eax, eax	xor eax, eax	; Return (HRESULT) S_OK
ret 8	ret 8	ret 8	;

### GetMem8

---

```

mov eax, [esp+4] ; Get first argument
mov ecx, [esp+8] ; Get second argument
mov edx, [eax]   ; Dereference low dword of first argument
mov eax, [eax+4] ; Dereference high dword
mov [ecx], edx   ; Store low dword in second argument
mov [ecx+4], eax ; Store high dword
xor eax, eax     ; Return (HRESULT) S_OK
ret 8            ;

```

The assembly listings show that the `GetMemN` functions dereference the memory address given by the first argument and return the value stored at that address in the second argument for a variable size of either one, two, four or eight bytes. The fact that the `eax` register is always cleared indicates that VB should call these functions as `Subs`, that they never raise an error and that we should not decorate the return value parameter with the `retval` IDL attribute.

## 2.3 PutMem

This is the assembly listing for the PutMem functions:

PutMem1	PutMem2	PutMem4	
mov ecx, [esp+4]	mov ecx, [esp+4]	mov ecx, [esp+4]	; Get first argument
mov al, [esp+8]	mov ax, [esp+8]	mov eax, [esp+8]	; Get second argument
mov [ecx], al	mov [ecx], ax	mov [ecx], eax	; Store value in first
xor eax, eax	xor eax, eax	xor eax, eax	; Return (HRESULT) S_OK
ret 8	ret 8	ret 8	;

### PutMem8

---

```

mov eax, [esp+4] ; Get argument
mov ecx, [esp+8] ; Get second argument
mov [eax], ecx   ; Store value in low dword of first argument
mov ecx, [esp+C] ; Get third argument
mov [eax+4], ecx ; Store value in high dword of first argument
xor eax, eax     ; Return (HRESULT) S_OK
ret C            ;

```

The PutMem functions are the counterparts of the GetMem functions. They write the value of the second argument at the memory address given by the first argument for different variable sizes.

**Note:** the run-time library contains additional functions within this family so the above list is not exhaustive. The GetMem-family also includes GetMemObj and GetMemEvent, which expect and return an object variable and call AddrOf if the specified argument is not Nothing, GetMemNewObj, which creates an object instance, and GetMemVar and GetMemStr, which use OleAut32.dll to copy a variant or string variable. For each GetMemX function the library contains an associated PutMemX function. Furthermore it contains a set of SetMem functions for use with object variables. However due to the additional processing these functions are less suitable for creating usefull aliases.

### 3. Declarations

The easiest way to use a `stdcall` function from Visual Basic is to write a `Declare` statement for it. A set of straightforward declarations for this family of runtime library functions would be:

```
Declare Sub GetMem1 Lib "msvbvm60" (ByVal Addr As Long, RetVal As Byte)
Declare Sub GetMem2 Lib "msvbvm60" (ByVal Addr As Long, RetVal As Integer)
Declare Sub GetMem4 Lib "msvbvm60" (ByVal Addr As Long, RetVal As Long)
Declare Sub GetMem8 Lib "msvbvm60" (ByVal Addr As Long, RetVal As Currency)

Declare Sub PutMem1 Lib "msvbvm60" (ByVal Addr As Long, ByVal NewVal As Byte)
Declare Sub PutMem2 Lib "msvbvm60" (ByVal Addr As Long, ByVal NewVal As Integer)
Declare Sub PutMem4 Lib "msvbvm60" (ByVal Addr As Long, ByVal NewVal As Long)
Declare Sub PutMem8 Lib "msvbvm60" (ByVal Addr As Long, ByVal NewVal As Currency)
```

Alternatively one could use:

```
Declare Sub GetMem1 Lib "msvbvm60" (Ptr As Any, RetVal As Byte)
Declare Sub GetMem2 Lib "msvbvm60" (Ptr As Any, RetVal As Integer)
Declare Sub GetMem4 Lib "msvbvm60" (Ptr As Any, RetVal As Long)
Declare Sub GetMem8 Lib "msvbvm60" (Ptr As Any, RetVal As Currency)

Declare Sub PutMem1 Lib "msvbvm60" (Ptr As Any, ByVal NewVal As Byte)
Declare Sub PutMem2 Lib "msvbvm60" (Ptr As Any, ByVal NewVal As Integer)
Declare Sub PutMem4 Lib "msvbvm60" (Ptr As Any, ByVal NewVal As Long)
Declare Sub PutMem8 Lib "msvbvm60" (Ptr As Any, ByVal NewVal As Currency)
```

A quick test in the VB IDE debug window confirms that the `GetMem` and `PutMem` functions operate as expected. The `Declare` statements extend the language with some powerful memory access features. They allow reading from and writing to any memory location (given sufficient access permissions<sup>3</sup>) as if it were declared as a `Byte`, `Integer`, `Long` or `Currency` variable. In combination with the built-in `VarPtr` function, or a `ByRef` argument declaration as demonstrated in the second set of declarations, this opens up the possibility for type casting.

*But wait! There's more...*

## 4. The Type Library

`Declare` statements in code modules have several limitations and disadvantages that can be circumvented by moving the declarations to a *type library*<sup>4</sup>:

- Because a typelib is language-independent it offers many more data types and expressions than VB itself, allowing declarations that can be used from but are impossible to specify in VB itself, e.g. function arguments that expect a Unicode string (`LPWSTR`) or a by value object interface (`IUnknown*`).
- Declarations in a typelib may be decorated with IDL attributes that offer meta-information to the compiler about the target, allowing finer control on the compiler-generated code (e.g. if `GetLastError` should be called) and the syntax that a declaration requires in VB (e.g. `Sub`, `Function` or `Property`).
- Each `Declare` statement in a code module increases the size of the compiled executable with at least 20 bytes. When using a type library, the executable will be smaller because the linker only includes the declarations that are actually used. This means that a type library can be continuously extended without unnecessarily bloating the executable.
- Function calls are faster when using a typelib instead of `Declare` statements because the external references are resolved at compile-time instead of run-time.
- Each declaration can be labeled with a description string that will be displayed in the Object Browser when the item is selected, including constants and members of enumerations and structures.
- Code modules with a lot of `Declare` statements can slow down or even destabilise the VB environment. A typelib puts less of a strain on the IDE, even when it contains a large number of declarations, because it is already in compiled form.

The type library accompanying this article is called [vbvm6lib.tlb](#), which stands for *Visual Basic 6 Virtual Machine Type Library*, which is also the description by which the library can be identified in the references dialog. The name of the type library is **VB6**, which may be used as a prefix for a type specifier to explicitly specify that a referenced type declaration belongs to this library, e.g. `VB6.DWord`. The module containing the function declarations is called **VBVM6**, the name of which may also be used as a function prefix to specify that a referenced function declaration belongs to this library, e.g. `VBVM6.MemLong`. The following section will offer a brief explanation of the declarations it contains.

It is beyond the scope of this article to explain all the intricacies of writing a type library. Suffice it to say that extensive documentation and many tutorials can be found both in the MSDN library and (elsewhere) on the internet<sup>5</sup>. To be able to use a type library in VB, you only have to know where to find the references dialog and the object browser. You do not need to understand the IDL (let alone



the assembly) source code, however it does help a great deal in understanding how to use these declarations.

## 4.1 Basic declarations

Let's first concentrate on the aliases for the GetMem and PutMem functions. This is a basic IDL declaration for the GetMem4 function:

```
[entry("GetMem4")]
HRESULT __stdcall GetMem4(
    [in] int Address,
    [out, retval] int * lpRetVal
);
```

Assuming that Address is a Long variable containing a valid memory location, and Value is a Long variable to receive the function return value, the next statement shows how to call the function:

```
' Fetch the Long at the given memory address
Call GetMem4(Address, Value)
```

The associated PutMem4 declaration would look like this:

```
[entry("PutMem4")]
HRESULT __stdcall PutMem4(
    [in] int Address,
    [in] int NewValue
);
```

Assuming that Address is a Long variable containing a valid memory location, and NewValue is a Long variable containing a value to store, this is how the function would be called:

```
' Change the Long at the given memory address
Call PutMem4(Address, NewValue)
```

## 4.2 Property syntax

One of the advantages of using IDL is that it allows us to add special attributes to a declaration. We can improve the former declarations by adding a `helpstring` that describe the member when it is selected in the Object Browser. Even better, we can change the above statements into an associated pair of property procedures by giving both functions the same name and adding the IDL attributes `propget` and `propput`. I decided to drop the `Get` and `Put` prefixes, keep the **Mem** term in the name as a common prefix and use the name of the associated data type as a suffix, in this case resulting in the name `MemLong` as the following example demonstrates:

```
[
    entry("GetMem4"), propget,
    helpstring("Accesses the Long value at the given address.")
]
HRESULT __stdcall MemLong(
    [in] int Address,
    [out, retval] int * lpRetVal
);

[
    entry("PutMem4"), propput
]
HRESULT __stdcall MemLong(
    [in] int Address,
    [in] int NewValue
);
```

The improved IDL declaration allows us to access the `GetMem`/`PutMem` function pair from VB as an indexed read/write property:

```
Dim Address As Long
Dim Value As Long
Dim NewValue As Long

' ...

' Read the current value of the Long integer
' stored at the given memory location
Value = MemLong(Address)

' ...

' Change the value of the Long integer
' at the given memory location
MemLong(Address) = NewValue
```

This syntax is quite self-descriptive. One would almost start to think that this property is some native but hidden VB statement, and in a sense it is since we do not rely on any external dependencies but instead only exploit some basic features built into the standard run-time library.

After adding such a declaration to the library for each of the eight functions, the Object Browser will show the following properties:

```
' Access memory location as if it were a 64-bit Currency variable
Property Get MemCurr(ByVal Address As Long) As Currency
Property Let MemCurr(ByVal Address As Long, ByVal NewValue As Currency)

' Access memory location as if it were a 32-bit Long variable
Property Get MemLong(ByVal Address As Long) As Long
Property Let MemLong(ByVal Address As Long, ByVal NewValue As Long)

' Access memory location as if it were a 16-bit Integer variable
Property Get MemWord(ByVal Address As Long) As Integer
Property Let MemWord(ByVal Address As Long, ByVal NewValue As Integer)

' Access memory location as if it were a Byte variable
Property Get MemByte(ByVal Address As Long) As Byte
Property Let MemByte(ByVal Address As Long, ByVal NewValue As Byte)
```

### 4.3 Casting

The previous properties all expect an explicit memory address as the first argument. We can alter these declarations so they expect an `As Any` argument as the first parameter. That way we can specify any variable as the source and access it as if it were a `Byte`, `Integer`, `Long` or `Currency`, effectively casting its value. I decided to use the common prefix `As` to indicate a cast operation, and again indicate the data type as a suffix, in this case resulting in the name `AsLong`:

```
[
    entry("GetMem4"), propget,
    helpstring("Accesses the argument cast to a Long.")
]
HRESULT __stdcall AsLong(
    [in] void * Ptr,
    [out, retval] int * lpRetVal
);

[
    entry("PutMem4"), propput,
]
HRESULT __stdcall AsLong(
    [in] void * Ptr,
    [in] int NewValue
);
```

This demonstrates how to call this property:

```
Dim DW As VB6.DWord
Dim L As Long
Dim HiW As Integer, LoW As Integer

' Initialize the DWord structure
DW.HiWord = &HAABB
DW.LoWord = &HCCDD

' Cast it to a long
L = AsLong(DW)
Debug.Print Hex$(L)
' Outputs AABBCCDD

' Change the DWord structure as a Long integer
AsLong(DW) = &HABCD1234

' Display the new value of the structure members
HiW = DW.HiWord
LoW = DW.LoWord
Debug.Print Hex$(HiW), Hex$(LoW)
' Prints: ABCD 1234
```

Adding such an alias to the type library for each of the functions leads to the following set of properties:

```
' Access given variable as if it were a 64-bit Currency
Property Get AsCurr(Ptr As Any) As Currency
Property Let AsCurr(Ptr As Any, ByVal NewValue As Currency)

' Access given variable as if it were a 32-bit Long
Property Get AsLong(Ptr As Any) As Long
Property Let AsLong(Ptr As Any, ByVal NewValue As Long)

' Access given variable as if it were a 16-bit Integer
Property Get AsWord(Ptr As Any) As Integer
Property Let AsWord(Ptr As Any, ByVal NewValue As Integer)

' Access given variable as if it were a Byte
Property Get AsByte(Ptr As Any) As Byte
Property Let AsByte(Ptr As Any, ByVal NewValue As Byte)
```

## 4.4 Structures

By adding a series of structures (User Defined Types) to the type library, we can also make aliases that allows us to extract the individual DWords, Words or Bytes out of a larger sized value:

```
[helpstring("A 32-bit dword represented by two integers.")]
typedef struct DWord {
    [helpstring("Least significant 16-bit word.")] short LoWord;
    [helpstring("Most significant 16-bit word.")] short HiWord;
} DWord;

[
    entry("GetMem4"),
    helpstring("Returns the argument cast to a DWord structure.")
]
HRESULT __stdcall AsDWord(
    [in] void * Ptr,
    [out, retval] DWord * lpRetVal
);
```

### Example:

```
Dim L As Long
Dim DW As VBVM6.DWord
Dim HiW As Integer, LoW As Integer

' Initialize a Long integer value
L = &H12345678

' Cast it to a DWord structure
DW = AsDWord(L)

' Extract the individual member values
HiW = DW.HiWord ' Now contains &H1234
LoW = DW.LoWord ' Now contains &H5678

Debug.Print Hex$(HiW), Hex$(LoW)
' Prints: 1234 5678
```

The type library contains the following structures:

Name	Size	Description
DByte	16 bits	Two Bytes representing one Word (Integer)
QByte	32 bits	Four Bytes representing one DWord (Long integer)
DWord	32 bits	Two Words representing one DWord
OByte	64 bits	Eight Bytes representing one QWord (Currency)
QWord	64 bits	Four Words representing one QWord
DLong	64 bits	Two Longs representing one QWord
DByteArray	16 bits	Array of two Bytes representing one Word
QByteArray	32 bits	Array of four Bytes representing one DWord
DWordArray	32 bits	Array of two Words representing one DWord
OByteArray	64 bits	Array of eight Bytes representing one QWord
QWordArray	64 bits	Array of four Words representing one QWord
DLongArray	64 bits	Array of two Longs representing one QWord

As the table shows there are two different structure declarations for each possible data size (1, 2, 4 or 8 bytes); one structure that represents the elements as named members of a record, and another that represents the elements as items of a fixed-size array. Depending on the situation, one may be more convenient than the other. Each structure is accompanied by an associated GetMem declaration that returns the value of a memory location in a variable of that type. Unfortunately Visual Basic does not allow passing UDTs by value, so there is no point in extending the type library with PutMem declarations using these structured types.

Keep in mind that, although the syntax might suggest that these functions return some kind of writable object referencing the original memory location, in fact they return a newly created structure containing a copy of the source argument. So do not be tempted to try a statement like:

```
' Won't work as expected
MemDWord(Address).LoWord = NewValue
```

For this reason the `propget` IDL attribute has not been assigned to these declarations, so VB identifies them as functions, not properties, and the associated helpstrings all indicate *Read Only*.



The following table lists the aliases that return a structure:

```
' Return the associated structure at given Address
Function MemDByte(ByVal Address As Long) As DByte
Function MemQByte(ByVal Address As Long) As QByte
Function MemDWord(ByVal Address As Long) As DWord
Function MemOByte(ByVal Address As Long) As OByte
Function MemQWord(ByVal Address As Long) As QWord
Function MemDLong(ByVal Address As Long) As DLong

' Cast the given argument to the associated structure
Function AsDByte(Ptr As Any) As DByte
Function AsQByte(Ptr As Any) As QByte
Function AsDWord(Ptr As Any) As DWord
Function AsOByte(Ptr As Any) As OByte
Function AsQWord(Ptr As Any) As QWord
Function AsDLong(Ptr As Any) As DLong

' Return the associated fixed-size array
' structure at given Address
Function MemDByteArr(ByVal Address As Long) As DByteArr
Function MemQByteArr(ByVal Address As Long) As QByteArr
Function MemDWordArr(ByVal Address As Long) As DWordArr
Function MemOByteArr(ByVal Address As Long) As OByteArr
Function MemQWordArr(ByVal Address As Long) As QWordArr
Function MemDLongArr(ByVal Address As Long) As DLongArr

' Cast the given argument to the associated
' fixed-size array array structure
Function AsDByteArr(Ptr As Any) As DByteArr
Function AsQByteArr(Ptr As Any) As QByteArr
Function AsDWordArr(Ptr As Any) As DWordArr
Function AsOByteArr(Ptr As Any) As OByteArr
Function AsQWordArr(Ptr As Any) As QWordArr
Function AsDLongArr(Ptr As Any) As DLongArr
```

## 4.5 Untyped Assignment

There may be cases that require a transfer of some value into a custom type, in which case the previous declarations cannot be used because of their predefined return value data type. Therefore, as a last resort, I implemented a series of aliases with both arguments declared untyped, all starting with the prefix 'Assign'. They are decorated so that VB will interpret them as regular VB Subs in which the target variable must be given as a ByRef argument, to allow variables of any type to receive the return value.

This is an example of such a declaration:

```
[
    entry("GetMem1"),
    helpstring("Assigns a byte from source to destination argument.")
]
HRESULT __stdcall AssignByte(
    [in] void * Source,
    [in, out] void * Destination
);
```

And this demonstrates how to use this function:

```
' Define a custom type
' This one fits in a DWord
Private Type CustomType
    SomeWord As Integer
    FirstByte As Byte
    SecondByte As Byte
End Type

' ...

Dim aType As CustomType
Dim aLong As Long

' Initialize a Long
aLong = -1&

' Assign the Long to the structure
Call AssignDWord(aLong, aType)

' Show the assignment result
Debug.Print Hex$(aType.SomeWord),
Debug.Print Hex$(aType.FirstByte),
Debug.Print Hex$(aType.SecondByte)

' Displays FFFF FF FF
```

Both the As Any parameter declarations and the illogical, reversed order of the arguments makes these functions extremely error-prone, though one could argue this holds for the module as a whole.

## 4.6 VarPtr tricks

The `VarPtr` function simply returns the `DWord` value pushed onto the stack by the caller. The assembly code looks like this:

### VarPtr

---

```
mov eax, [esp+4] ; Dereference the first argument
ret 4           ; Return the result
```

The built-in `VarPtr`, `StrPtr` and `ObjPtr` functions are actually three different (hidden) aliases for the same function, as the original IDL declarations clearly show:

```
[entry("VarPtr"), hidden]
long _stdcall VarPtr([in] void * Ptr);

[entry("VarPtr"), hidden]
long _stdcall StrPtr([in] BSTR Ptr);

[entry("VarPtr"), hidden]
long _stdcall ObjPtr([in] IUnknown * Ptr);
```

In addition to these three aliases we can write some useful declarations for this function of our own.

The **ArrPtr** function obtains a pointer to the `SAFEARRAY` structure of any array argument except string arrays due to Visual Basic's implicit Unicode/ANSI conversion.

```
Function ArrPtr(Ptr() As Any) As Long
```

We have to declare a special **StrArrPtr** alias for string arrays using a `BSTR` parameter declaration, because a `SAFEARRAY(void) *` parameter declaration causes VB to pass in a pointer to a temporary array of ANSI converted strings. Upon returning from `ArrPtr`, this pointer will be invalid because the temporary array will already have been destroyed. By using a `SAFEARRAY(BSTR) *` parameter declaration, the original string array pointer itself is being passed in and consequently returned.

```
Function StrArrPtr(Ptr() As String) As Long
```

By aliasing the `VarPtr` function's return value, we can construct functions that return a copy of the source pointer cast to a reference variable type.

The **AsString** function casts a pointer to a Visual Basic (`BSTR`) String variable. The returned string will reference the characters at the given address, but also include the preceding `DWord` which will be interpreted as the string length, limiting it's usefulness. This function is the opposite of `VBA.StrPtr` and `VBVM6.StringPtr`.

```
Function AsString(ByVal BSTR As Long) As String
```



The **StringRef** function returns a second (stolen) reference to the Source string argument. The returned string will be pointing to the same characters as the given string. The caller is responsible for clearing the stolen reference without freeing the allocated memory.

```
Function StringRef(BSTR As String) As String
```

Example:

```
Dim sSource As String
Dim sStolen As String
' ...
sStolen = StringRef(sSource)
' sStolen and sSource now point
' to the same characters
' ...
' Clean up
StringPtr(sStolen) = 0&
```

The **AsObject** function casts a pointer to an `IUnknown` object interface pointer. No reference counting is performed. This function is the opposite of `VBA.ObjPtr` and `VBVM6.ObjectPtr`. It is perfectly suited for casting a stolen object reference back to an object interface pointer.

```
Function AsObject(ByVal Ptr As Long) As IUnknown
```

The **NoAddRef** function returns an uncounted (stolen) object reference. No reference counting is performed. The caller is responsible for keeping the object alive as long as the reference is in use and should clear the stolen reference without decreasing the reference count. Note that the argument has been declared `As Any` instead of `As IUnknown` to prevent VB from calling `AddRef` on the given object argument value.

```
Function NoAddRef(Object As Any) As IUnknown
```

Example:

```
Dim oSource As Object
Dim oNotCounter As Object
' ...
Set oNotCounted = NoAddRef(oSource)
' oNotCounted now contains an
' uncounted reference to oSource
' ...
' Clean up
ObjectPtr(oNotCounted) = 0&
```

## 4.7 Goodies

In addition to the previously mentioned generic declarations, the powerful IDL syntax allows us to write some very specific aliases that deal with the native VB reference types `Variant`, `String`, `Object` and (safe-)arrays.

The **`VariantType`** property sets or returns the (unmodified) `VARTYPE` member of the given `VARIANT` argument. It allows read/write access to the same value as returned by the built-in `VBA.VarType` function, but that function masks out the `VB_BYREF` flag and this property doesn't.

```
Property Get VariantType(V As Variant) As Integer
Property Let VariantType(V As Variant, ByVal NewValue As Integer)
```

The **`StringPtr`** property sets or returns the value of the `BSTR` argument's *string pointer*, the memory address of the string's first character. It allows read/write access to the same value as returned by the `VBA.StrPtr` function.

```
Property Get StringPtr(S As String) As Long
Property Let StringPtr(S As String, ByVal NewValue As Long)
```

The **`SAPtr`** property sets or returns the value of the array argument's *SAFEARRAY pointer*, the memory address of the `SAFEARRAY` structure describing the given array. The argument can be an array of any type except `String` due to the ANSI/Unicode conversion. Use the special `StrArrPtr` for string arrays.

```
Property Get SAPtr(Arr() As Any) As Long
Property Let SAPtr(Arr() As Any, ByVal NewValue As Long)
```

The **`StrSAPtr`** property sets or returns the value of the string array argument's `SAFEARRAY` pointer. It circumvents the ANSI/Unicode conversion that is performed when calling the `VBVM6.ArrPtr` property with a string array argument.

```
Property Get StrSAPtr(StrArr() As String) As Long
Property Let StrSAPtr(StrArr() As String, ByVal NewValue As Long)
```

The **ObjectPtr** property sets or returns the value of the given object argument's *object pointer*. It allows read/write access to the same value as returned by the `VBA.ObjectPtr` function. This function could be made slightly more type-safe by declaring the arguments as `IUnknown**` (i.e. `ByRef IUnknown`), but this would cause VB to `QueryInterface` the given object argument before and `Release` it after calling the function, unnecessarily slowing the function call down (a lot). However, because VB always supplies object pointers by reference regardless of how the argument is specified (be it either `ByVal` or `ByRef`, `As Object` or `As Any`), which is what this function expects in order to return a valid result, we can safely declare the argument as `void*` without introducing possible ambiguities. Actually this produces the same declarations as the `MemLong` aliases, but the property name allows for more readable code in this specific use case.

```
Property Get ObjectPtr(ByVal Obj As Any) As Long
Property Let ObjectPtr(ByVal Obj As Any, ByVal NewValue As Long)
```

The **VTablePtr** property sets or returns the value of the object argument's *VTable pointer*. It allows read/write access to the value at the memory location returned by the `VBVM6.ObjectPtr` and `VBA.ObjectPtr` functions, which contains the object interface or VTable pointer. This means that `VTablePtr(AnObject)` accesses the same value as `MemLong(ObjectPtr(AnObject))`. The `IUnknown*` argument causes VB to perform a pair of `AddRef` and `Release` calls on the given object argument before and after calling the function, causing a slight performance hit. However this declaration is required for the function to operate correctly. Note that it is impossible to declare such a by value object parameter in Visual Basic code, because object arguments of VB-generated functions always expect pointers to objects; a parameter declared as `ByVal AnObject As Object` still causes VB to supply a pointer to an object as the function argument, although it is a copy of the original object pointer (so the function cannot modify the original caller's value), but not the value of the pointer itself as defined by the `IUnknown*` idl type declaration this property uses (which VB *does* recognize as such).

```
Property Get VTablePtr(Obj As IUnknown) As Long
Property Let VTablePtr(Obj As IUnknown, ByVal NewValue As Long)
```

Regular Visual Basic objects store their reference count at the memory location immediately following the VTable pointer. The **GetUnkDesc** and **GetUnkDescFromPtr** function declarations alias the `GetMem8` run-time function to return this information for a given VB object in the form of the **VBUnkDesc** structure:

```
' Internal object descriptor pointed to by a standard
' Visual Basic (in-process) IUnknown object interface
' pointer, i.e. by object references stored in variables
' that have been declared "As IUnknown".
Type VBUnkDesc
    ' Pointer to the object interface's VTable structure.
    ' Applies to all COM object interface pointers.
    pVTable As Long
    ' Object reference count. Only applies to IUnknown
    ' interface pointers of objects with the default
    ' Visual Basic reference counting mechanism.
    RefCnt As Long
End Type

Property Get GetUnkDesc(Obj As IUnknown) As VBUnkDesc
Property Get GetUnkDescFromPtr(ByVal Ptr As Long) As VBUnkDesc
```

The descriptor structure is 8 bytes long, nicely fitting the `GetMem8` runtime library function. The `pVTable` member will always be valid for any COM object (per definition), but the `RefCnt` value is only valid if the variable that is being supplied as the function argument has actually been declared `As IUnknown`. Because of the `IUnknown*` parameter declaration, VB will `AddRef` and `Release` the given object before and after calling the function, so the returned reference count will always be one more than expected. A typical application of this alias would be such a function:

```
Function GetRefCnt(AnObject As stdole.IUnknown) As Long
    ' Since all COM objects implement IUnknown, the argument
    ' type declaration causes VB to simply call AddRef (not
    ' QueryInterface) on the given object argument value

    ' Protect against NULL objects (or GetUnkDesc will GPF!)
    If (AnObject Is Nothing) Then Exit Sub

    ' Subtract 2 to compensate for the AddRef calls that are
    ' caused by the IUnknown function arguments
    GetRefCnt = VBVM6.GetUnkDesc(AnObject).RefCnt - 2&
End Function
```

Assuming that the variable `pUnk` contains a valid `IUnknown` interface pointer, then the following holds:

```
GetUnkDesc(pUnk).VTable := VTablePtr(pUnk)
GetUnkDesc(pUnk).RefCnt := MemLong(ObjectPtr(pUnk) + 4)
```



## 5. License

The *Visual Basic 6 Virtual Machine Type Library* binary and source code are released under the *MIT license*<sup>\*</sup>, which is similar in effect to the BSD licence (this licence is *Open Source* certified and complies with the *Debian Free Software Guidelines*).

The license grants unrestricted rights to anyone (including companies) to copy, modify, and redistribute the *VBVM6Lib* type library and its source code (even for commercial purposes) as long as the original copyright and license terms are retained. However the library is not covered by any warranty and if it causes damage you're on your own, so do not use it if you're unhappy with that. In particular, note that the MIT licence is compatible with the GNU GPL. Therefore it is allowed to incorporate VBVM6Lib or pieces of it into a GPL program.

## 6. Conclusion

And there you have it: unrestricted memory access and type casting in pure Visual Basic. The *VBVM6Lib* type library offers the advanced VB programmer a complete range of assorted low-level functions in a reasonably readable syntax, considering the technical character of these operations. It relies completely on a couple of very basic functions that are built into the VB runtime library, so using these aliases does not introduce any external dependencies into the project whatsoever.

In view of the technical and unprotected character of these functions it must be said that this type library is not for the faint of heart. Any protection that Visual Basic traditionally offers is out of the way so it's up to the caller to ensure that these functions are called with valid arguments. Therefore when experimenting with these functions it is strongly advised to use an OS that has strong interprocess protection, otherwise the system probably has to be rebooted on each and every invalid call<sup>6</sup>. Mind you, this risk is not related to the Visual Basic programming language but inherent to direct and unprotected memory access. In any case caution is advised.

However assuming that you know what you are doing, this type library will allow you to pull the same low-level tricks in Visual Basic as in any given programming language. So no more excuses saying that you can't do that in VB anymore. Open the references dialog, select the *Visual Basic 6 Virtual Machine Type Library* and just do it.

Happy programming!

Michel Rutten

## Notes

**Note 1** Although this article and the associated type library target the Visual Basic 6 run-time library `msvbvm60.dll`, most of the described techniques could probably be ported to VB5 and the `msvbvm50.dll` run-time library, since it too contains the family of `GetMem/PutMem` functions. However VB5 might not accept some of the argument data type declarations in the the type library, so these aliases would have to be either modified or dropped. Obviously I haven't tested this, so you're on your own. [Back](#)

**Note 2** Yo Bill, chill out man, I'm not tryin' to rip you of so plz [back](#) off...

**Note 3** A bit of friendly advice: 32-bit operating systems normally prevent processes from accessing each other's (or the kernel's) memory. If you feel like peeking and poking around in there like them good 'ole GWBASIC times, I suggest you check out the `IsBadReadPtr` and `IsBadWritePtr` functions in `kernel32.dll` first. These functions can tell you if the OS permits reading from or writing to a given memory address or range. Otherwise that kernel will hit right back at ya with a nasty exception fault. Also keep in mind that none of the declarations tolerate being called with an argument value that equals 0, `Nothing` or `vbNullString`, so make sure you call them with valid arguments unless you like to take frequent coffee breaks. [Back](#)

**Note 4** A type library is a binary file or component within another file that contains type information about types and objects exposed by an ActiveX application in a form that is accessible to other applications at runtime. Such a type may be an alias, enumeration, structure or union. An object can be a module (defines a group of functions, typically a set of DLL entry points), interface, `IDispatch` interface (disinterface), or component object class (coclass). Using a type library, an application or browser can determine which interfaces an object supports, and invoke an object's interface methods. This can occur even if the object and client applications were written in different programming languages. You can create your own type libraries using the MIDL (or older `MkTypLib`) compiler.

Visual Basic creates type library information for the classes you create, provides type libraries for the objects it includes, and lets you access the type libraries provided by other applications. If you reference an external type library, VB uses it at design time to check the syntax of your code and at compile time to build the executable. But at run-time the type library is no longer required since the information has already been compiled into the binary. Therefore `typelibs` (usually) do not have to be distributed with the executable. [Back](#)

**Note 5** On the VB2TheMax website you can find an excellent [IDL For VBTutorial](#) by Philip Fucich. And in the MSDN library you may find a series of articles on the internals of VB and writing type libraries by Bruce McKinney, [Extending Visual Basic with C++ DLLs](#), the first of which is appropriately called *Stealing Code with Type Libraries*. [Back](#)

**Note 6** Guess how I know this... [Back](#)

---

Copyright © 2002 Michel Rutten, [m.rutten@bigfoot.com](mailto:m.rutten@bigfoot.com)

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

