COM in PureBasic
Part 1 — The Basics

by Timo "*freak*" Harter
purebasic@freak1.de

April 30, 2007

Contents

Chapter 1

Introduction

When i first started with PureBasic (and programming for windows in general), it was great to explore what was possible using calls to the Windows API. I could start playing around with the API commands one by one, not needing to get them all right at once, as PureBasic provided the perfect framework to still easily create a working program.

At one point though i always hit a wall: To do something more advanced, there was sometimes the accessing or creating of a COM interface required to do some task. Usually allready the description of the methods in question made me abandon the project, as they talked about, and required knowledge of a much larger concept which is COM. The few example codes that were around on that topic back then dealt with specific problems, and i was usually not able to make too much sense of them either (at least not in a way that i could write code to do some completly different task based on it) So i had to learn it all the hard way. Although there is much more code on this topic around on the PureBasic forums these days, most of it is written for a specific task as well, and it is not too easy to learn the general concepts from them.

This series of tutorials is intended to fill this gap. My hope is that it will give a good overview over the general stuff and underlying concepts and allow you to make your first steps in COM without the need to find it all out on your own.

1.1 About this Tutorial

What you need to know: This tutorial is not for the beginners with the PureBasic language. You should be familiar with it to a point where you can write more than just small test programs. Especially some knowledge on using pointers and working with memory buffers is absolutely required. Also at least a little prior experience with the Windows API will be helpfull. You need no prior knowledge about COM itself. I'll try to cover all that is needed there in these tutorials.

What you will learn here: This tutorial is intended to provide a basic overview about COM. What its general concepts are and how to do the basic stuff. After reading it, you should be able to better understand the COM related code others allready posted on the forum, as well as take your first steps in creating your own code for accessing COM stuff.

Later tutorials will deal with some larger fields of COM (like the Shell Namespace, the MSHTML models for the WebGadget or something similar), and are intended to deepen this knowledge with specific examples and explanations on a larger area of COM. Finally i hope to write a tutorial on how to create your own "components" to use with COM.

What you will not learn here: COM is a big field. I cannot teach you how to solve a specific problem or work with a specific interface. That would be of no use. The later tutorials will be a bit more specific than this one, but still quite general.

What i am trying to teach you is the stuff you need to know to be able to understand the COM related documentation, so you can solve the problem at hand yourself. After that, its up to you to read the given documentation carefully, and then simply try it. Believe me, it gets easier over time, as the basic concepts repeat themselves quite a lot and just need some getting used to.

1.2 COM – What is it anyway?

TODO

1.3 Basic design principles

I want to list here a few basic design principles of COM, and what significance they have for working with COM.

COM is a *binary* standard, not a *source* standard: COM specifies how to interact with components on a binary level. (This means it defines how to call methods etc in the compiled form, not relying on the specific source construct of a programming language, like C++ classes for example.) This way it is completly independent from programming languages. Any language that supports indirect function calls through pointers can create and interact with COM components. It does not matter what language the component is written in, communication with it is always the same way.

Access to components is done through interfaces, and *only* through that: Other than other object oriented frameworks, in COM your only access to a component are the interfaces that it provides. There is no way to directly access other data on the

component from the outside (which is possible with public variables in a Java class for example).

This completly hides the implementation from the caller, which allows to exchange the implementation behind an interface without the need to change anything on the client side, and it also allows COM to provide the possibility to access components in other processes and even on other computers as if they existed in the own application. (This is achieved by a concept called "marshalling", where COM creates a fake interface on your side to take your calls, then sending them over the network to where the real component is located. Both the client and the component can be written as if their respecitive counterpart is located in the own application, and COM does all the work inbetween.)

Interfaces are accessed through a GUID, not by name: A GUID is a 128bit identifier that uniquely identifies each interface (I explain more about GUIDs in the next chapter) By not using names to access an interface or method, naming conflicts cannot happen between different components.

Interfaces have a fixed set of methods: This eliminates versioning conflicts. Once an interface is created and is assigned a GUID, this id represents this interface as it is. There cannot be a "newer version" of the same interface that has more or different methods, which would lead to a problem when somebody expects the old version of the interface and gets the new one or vice versa. Instead, to extend the features of a component, a new interface is created (it can contain the old one if that is desired) with a new GUID. Since components can export multiple interfaces (see the chapter on `IUnknown`) it is easily possible to both support the new and old functionality (through exporting both interfaces) The point is that if a client requests an interface, he will either get exactly what he requests, or a failure. He cannot get something that looks like what he wanted, but is in fact different (a different version etc)

1.4 Classes, Objects, Interfaces

TODO

1.5 Articles for further reading on COM

TODO

Chapter 2

What you need – PB language constructs for COM access

TODO

2.1 Interfaces in PureBasic

2.2 Handling GUID values

2.3 Where to find information

2.4 Useful tools to work with

2.5 A first example

Chapter 3

The heart of COM – The IUnknown interface

A basic principle of COM is that all interface extend the so called `IUnknown` interface. This means that you always have 3 methods available, no matter what interface you deal with. (In fact, you do not even need to know what interface you are dealing with. This is a very important point) These 3 methods are `QueryInterface()`, `AddRef()` and `Release()`. Their purpose is to manage the lifetime of the object, and to ask the object what other interfaces it supports.

3.1 Lifetime of an Object

Managing the lifetime of an object is a basic question, that any object oriented framework must address. Lets say you are the one who created the object, then you would also know when you do not need it any longer. That is quite simple. What happens now if you pass a pointer to your object to another function ? It may store your pointer for future use. Who knows when it will no longer be needed then ?

COM solves this problem by letting the object itself handle its lifetime. The object itself keeps a count of the so called "references" to it (anyone holding a reference to an interface can call its methods). `AddRef()` increases this count and `Release()` decreases it. They both take no arguments. Whenever the count reaches 0, the object must delete itself (ie free all resources it allocated).

When an object is first created, its reference count is 1 (the reference that the creator holds). Whenever the creator of the interface passes the pointer to somebody else (a function or method for example), the one who received the pointer can then call the `AddRef()` method to get a reference themselves. As long as you hold a reference to an object, you can be sure that it is still valid to call methods on its interfaces. Whenever the object/interface is no longer needed, somebody holding a reference calls `Release()` to drop his reference. The creator of the object must call it once as well, to drop the initial reference he got when creating the object.

You see that it now does not matter if the creator of the object calls `Release()` while somebody else still needs the object. The creator simply drops his reference to the object, decreasing its reference count, but the other references still exist, so the object is not freed. The object is freed when nobody is using it anymore (when the reference count drops to 0).

Handling the references you have to COM objects properly is important. If there is a bug (one `Release()` too much or less for example) two things can happen: Either the object is never freed, because the reference count does not reach 0, or it is freed too early, because `Release()` was called for for a reference that belongs to someone else. So getting this right is important.

You need to call `Release()` in all these cases:

- for each object you created
- for each `AddRef()` call
- for each successful `QueryInterface()` call (as it increases the reference count as well, see the next section)
- for all interfaces that you receive from another function or method call. (unless otherwise noted in the documentation)

  The convention here is that a function/method *returning* an interface pointer will increase its reference count, while when you pass an interface *to* a function/method. The receiving function will call `AddRef()` itself. This is just a convention, but most COM related functions/methods comply to it. Where this is not the case, it is usually well documented.

Note: Both `AddRef()` and `Release()` return the new reference count for the object after the call, but you can only really use this return value for debugging purposes. The remaining references belong to somebody else, so even though you may know that the reference count is not 0 yet (and thus the object still exists), it is not save to continue calling methods on the object, as you gave up all control of the object lifetime when you released any references you held. (the other holders can drop their references at any time, causing the object to be freed.)

3.2 Querying for other Interfaces

A typical object supports more than one interface. Different interfaces are used to support different types of actions. The `QueryInterface()` method has two important functions: It allows access to different interfaces on the same object, and it provides a save way to check which functionality an object supports.

Syntax:
```
QueryInterface(*IID, *Object)
```

`*IID` is a pointer to an IID that represents the interface you are requesting. `*Object` must be the pointer to a variable that will receive the new interface pointer on success. If successful, the method fills your object pointer with the pointer to the requested interface, increases the reference count of the object and returns `#S_OK`. (The reference count is increased because you now have a reference to the old interface and one to the new interface) If the interface is not supported, the method returns `#E_NOINTERFACE`. You must always be prepared for the fact that the interface you request is not available.

This way of accessing different functionality on one object has a number of advantages. It provides a save way to ask for a given functionality from an object, and especially a reliable way for the object to tell you if something is not supported. Lets say you try to query some "new" interface that was not present at the time the object you are working with was developed. This method tells you whether or not the interface is available without the risk of a crash that would result from just trying to call a function or method that is not there.

Since `IUnknown` is the base for every COM interface, you actually do not need to know anything about an interface pointer you receive (except that it is a COM interface), as you can request any interface through the `QueryInterface()` method and then you know whether the functionality you are looking for is actually present. Mostly you will know of course what interface you are receiving from a given function/method call and you do not need to query for it specifically, but this is still a very useful feature.

3.3 Rules for QueryInterface

There are some rules for `QueryInterface()` which all COM object must fulfill, and which are interesting from the caller's standpoint:

- A query for `IUnknown` must always be successful, and any query for this interface from one object must return the same pointer.

  Note that this is only true for the `IUnknown` interface. If you query twice for another interface, you are never guaranteed that the returned pointer will be equal. The fact that with `IUnknown` you are guaranteed that the returned pointer will be the same as long as you do the query on the same object allows you to compare if two interface pointers you have (of any kind) belong to the same object. Just query for `IUnknown` on both these interfaces and check if the result matches.

- The list of supported interfaces on any object must be *static*, not *dynamic*.

  This means if a query for a specific interface was successful once, it must succeed again as long as the object exists. There can be no condition where a query only works in certain conditions.

- Queries for interfaces must be *reflexive*, *symmetric* and *transitive*.

  This means in essence that you can query for every interface that a given object supports from any interface pointer to that object.

Chapter 4

Calling COM methods – Parameter and return types

4.1 Method return codes

A usual COM method returns a value of the `HRESULT` type. The corresponding PureBasic type is a `Long`. The predefined error codes for COM are all negative, so as a general rule, all negative return values can be interpreted as an error, while a return value of 0 or bigger indicates success.

The full list of possible error codes together with a short description is listed in the `WinError.h` include file which is included with the Platform SDK. PureBasic should know most of the constants defined there. The debugging functions of my COM Framework include a `GetCOMErrorConstant()` and `GetCOMErrorMessage()` function to help you identify error codes while developing.

Each COM method has its individual set of possible return values defined in its documentation, so make sure you look them up for the methods you use. For most of the methods though, the value `#S_OK` (equals 0) indicates success. Sometimes also the constant `#NOERROR` is used in documentation, which is the same value. There are some other return values that are quite common, so i will list them here:

`#S_OK` Operation successful

`#S_FALSE` Not an error code (value 1), but a common value to indicate a "false" status

`#E_NOTIMPL` The called method is not implemented in this interface

`#E_INVALIDARG` One of the arguments passed to the method is not accepted by the method

`#E_OUTOFMEMORY` Not enough memory to allocate needed data

`#E_FAIL` Operation failed (unspecified reason)

`#E_ABORT` Operation aborted

`#E_UNEXPECTED` Unexpected failure

Note that this interpretation of the return value (interpreting 0 as success and < 0 as failure) is different from how PB does it in its functions for example, so a common source of bugs is to not check for something like `#S_OK`, but instead simply for a nonzero result (which of course check for an error then and not for success). So keep this in mind.

4.2 Strings in COM

COM uses almost exclusively unicode to represent its strings. The most common string type is the `BSTR` type, although also other unicode types are sometimes used, and very rarely, also ansi strings. "Normal" unicode strings are usually marked as the type `LPWSTR`, `LPOLESTR` or similar. The can be handled directly as PB strings when you compile in unicode mode, and easily converted with `PokeS()` and `PeekS()` if you compile in ascii mode. The handling of the BSTR type will be discussed below.

Creating a BSTR string: A BSTR string is a string in the same format as PB's unicode strings, but allocated with a special API function `SysAllocString_()`. So if you compile your program in unicode mode, all you need to do is call that allocation function with a pointer to a PB string. If you do not compile in unicode mode, you first need to translate the string into unicode with the `PokeS()` function before turning it into a BSTR string.

The procedure below should do the conversion in any case:

```
1  Procedure.l MakeBSTR(String.s)
2    Protected Result = 0
3    CompilerIf #PB_Compiler_Unicode
4      Result = SysAllocString_(@String)
5    CompilerElse
6      Protected *Buffer = AllocateMemory(Len(String)*2 + 2)
7      If *Buffer
8        PokeS(*Buffer, String, -1, #PB_Unicode)
9        Result = SysAllocString_(*Buffer)
10       FreeMemory(*Buffer)
11     EndIf
12   CompilerEndIf
13
14   ProcedureReturn Result
15 EndProcedure
```

Note that the `PokeS()` part actually works in unicode mode as well, as the string will simply be poked unchanged, but the first solution is of course much faster in unicode mode.

Reading a BSTR string: Since the unicode representation of the BSTR type and PB are the same, they can directly be read with the `PeekS()` function. Just remember to use the `#PB_Unicode` flag when you do not compile in unicode mode to correctly read the string.

Note: As a BSTR string is specifically allocated with `SysAllocString_()`, it must also be freed with a function called `SysFreeString_()`. You need to do this both for BSTR strings received from a method call as well as for those you created to pass to a method. (The method will create its own copy if it needs to store the string.

PureBasic actually makes passing strings to a COM interface very easy, as it provides an automatic conversion for the above through the so called "Pseudotypes" (`p-bstr` and `p-unicode`). If an interface parameter is defined of this type, you can simply pass a normal PB string, and PureBasic will automatically do this conversion for you. This is supported almost everywhere where a string is passed to a method (You can look at an interface in the IDE's Structure Viewer, and if the parameter is defined as `p-bstr` or `p-unicode`, the automatic conversion is done.) You will still need to manually create the string if it is requested inside a structure like the `VARIANT` structure.

When a method returns a string (typically a BSTR one), it asks you to provide a pointer into which it will fill the string. Here no automatic conversion is supported. What you need to do here is pass a pointer to a long variable, which will be filled with the BSTR pointer. Then you can use `PeekS()` to read the string. Then do not forget `SysFreeString_()` of course.

Example:

```
1 If MyObject\SomeMethod(@bstr_sting) = #S_OK
2   Debug PeekS(bstr_string, -1, #PB_Unicode)
3   SysFreeString_(bstr_string)
4 EndIf
```

4.3 Passing Interface pointers

Often a method requires an "interface pointer" as a parameter. Either to access the given interface, or to return a newly created interface through this pointer. Lets say we have an interface variable called `MyInterface.ISomething`, what should be the parameter now? `MyInterface` or `@MyInterface` ?

The COM related documentation (mainly the Platform SDK) use C (or rarely C++) to show the required parameters. In C, an interface is always a pointer, so when the parameter definition looks like "`ISomething *argname`", you only need to pass `MyInterface`. If the documentation looks like "`ISomething **argname`", what is requested is a pointer to the interface pointer, so you have to pass `@MyInterface`.

In terms of functionality, `MyInterface` is the interface itself. Passing it to the method allows the called method to access the interface (call its method), but it cannot modify

your interface variable, as a copy is made when it is passed to the method. So if the method needs to modify your variable (to pass back a newly created interface through it for example), it will need a pointer to your interface variable, so by writing at that pointer location, it can actually change your variable. There you need to pass `@MyInterface`, which is the pointer to your variable. The latter one is the case for `QueryInterface()` for example, where the new interface is returned through the second argument. See the chapter on `IUnknown` for an example.

So the rule of thumb is: If the method only works with the interface, there is no "@", if the method needs to change the value (usually to return a new interface), the "@" is required.

4.4 The VARIANT data type

If you look up the definition of the `VARIANT` data type in the Platform SDK, it will look quite scary. It is a huge structure with a union inside containing many weird fields. That alone made me try to avoid using it for quite a while.

In truth, the concept is quite simple: `VARIANT` is a structure used in places where more than one type of variable may be possible as an argument. It has one member called "`vt`", which defines which field of the union to use (with the `#VT_...` set of constants), and then the union, of which only one field is used at a time.

Lets start with an example:

```
1 Define var.VARIANT
2 var\vt   = #VT_I4
3 var\lVal = 123
```

What we created here is a valid VARIANT structure containing a `#VT_I4` (4 byte integer, same as `Long` in PB) value of 123. The SDK tells you which union field to use for which vt type.

Here is a list of the most common used vt types, and which union field to use for them. This list should be enough for the common cases. In the rare cases where you need something else, look it up in the SDK. (see "VARIANT and VARIANTARG" in the index)

`#VT_EMPTY` Empty value. No field is in use.

`#VT_BOOL` A boolean value is in the `boolVal` field. Only `#VARIANT_FALSE` (0) or `#VARIANT_TRUE` ($FFFF) are valid.

`#VT_BSTR` A BSTR string is in the `bstrVal` field. Note that such a string must be freed with `SysFreeString_()` when the `VARIANT` is no longer needed.

`#VT_I1,#VT_I2,#VT_I4,#VT_I8` A signed integer of 1,2,4,8 bytes is in the `cVal, iVal, lVal, llVal` field respectively. These directly correspond to the `Byte, Word, Long, Quad` datatypes in PB.

`#VT_UI1,#VT_UI2,#VT_UI4,#VT_UI8` Same as the `#VT_Ix` types, except these are unsigned and the fields `bVal, uiVal, ulVal, ullVal` are used.

`#VT_INT, #VT_UINT` Signed and unsigned integer (4byte on 32bit platforms, 8byte on 64bit ones), stored in the `intVal, uintVal` fields.

`#VT_R4, #VT_R8` 4 and 8 byte float types. Fields are `fltVal` and `dblVal`. These correspond to the `Float` and `Double` types of PB.

`#VT_UNKNOWN, #VT_DISPATCH` The `VARIANT` contains an `IUnknown` or `IDispatch` pointer in the `punkVal` and `pdispVal` fields. You have to call the `Release()` method on these if you received such a `VARIANT`.

`#VT_BYREF` This is a special modifier that is combined with another type to indicate that the `VARIANT` only stores a pointer to the given value. The fields used then are the same as a above, only with a preceding "`p`" to indicate a pointer.

Note that `#VT_EMPTY` has the value 0, so since PB initializes all new variables/structures to 0, a newly created `VARIANT` structure in PB is allready of type `#VT_EMPTY`. (There is no need to initialize it to that like there is in other languages)

Creating a VARIANT of any such type is straight forward. What do we do though if a method fills in a `VARIANT` for us, and the contained value can be of any type ? Do we have to write code to handle every possibility then ? Fortunately no. COM provides a set of functions to deal with the `VARIANT` type in a simple matter.

There are actually quite a number of functions make various conversions and calculations directly on the variant type (for example, `VarMul_()`, `VarCmp_()` or `VarPow_()` to name a few) and also a number of functions to make any type of conversion between `VARIANT` contents such as `VarBstrFromBool_()` for example. I will only focus on these that can deal with any type of `VARIANT` and are useful to deal with `VARIANT` structures received from methods. If you ever need a specific conversion function, just look it up.

`VariantClear_(*variant)` :
This function takes the pointer to a `VARIANT` structure and frees whatever it contains. If the content is a BSTR, `SysFreeString_()` is called, if it is an interface pointer, `Release()` is called on it etc. This is useful for any `VARIANT` type that you have no code for to handle, or generally for any `VARIANT` structure that is no longer needed.

`VariantCopy_(*destination, *source)` :
This function copies the source `VARIANT` to the destination. It copies a BSTR string, calls `AddRef()` on interfaces etc, so that the copy is later independant from the source structure. This function also calls `VariantClear_()` on the destination before making the copy, freeing any old data in the destination.

`VariantChangeType_(*destination, *source, flags, vt)` :
This function converts the source `VARIANT` to the vt type given by the "vt" parameter and puts the result in the destination. Source and destination can be equal in which case the conversion is done in place. "flags" can modify the behaviour a bit, but i will not discuss them here. It is important to check the result of this function (it is of type

HRESULT), as the conversion may fail (you cannot convert a string to an `IDispatch` for example.)

Below is an example of how to use this function to handle a variety of variant types with little code:

```
1  Procedure.s VariantString(*var.VARIANT)
2    Protected Result.s = ""
3
4    ; do an in-place conversion to a BSTR
5    If VariantChangeType_(*var, *var, 0, #VT_BSTR) = #S_OK
6      Result = PeekS(*var\bstrVal, -1, #PB_Unicode)
7      VariantClear_(*var)
8    EndIf
9
10   ProcedureReturn Result
11 EndProcedure
```

Since a conversion to a string is almost always possible, this is an easy way to handle any type of variant that you receive from a method without much work. Note that this also frees the passed variant, as an in-place conversion is done and then `VariantClear_()` is called with it. If this is not desired, simply do the conversion to a new `VARIANT` structure.

In a similar way, you could change the type to `#VT_I4` to easily read it as a `Long`, no matter which numeric type (I1, I2, I4, UI1, ...) the `VARIANT` actually had.

Knowing how to create a `VARIANT` of a specific type, and with these functions to handle any type of them, working with `VARIANT`s is not that hard anymore. It is only important to keep in mind to always free the more complex types (BSTR, interfaces) when no longer needed to avoid memory leaks. The basic types (I1, I2, I4 etc) do not need a special free call of course.

Sidenote: Unlike other structures which are usually passed to a method or API function as a pointer, the `VARIANT` structure is often expected to be put fully on the stack. Since the structure is 16 bytes, that presented a problem to earlier versions of PureBasic, as this kind of thing was not supported. Since 4.0, there is the `p-variant` pseudotype, which handles that. The COM interfaces allready predefined in PB make use of that whenever needed, so you do not have to concern yourself with this problem. Just use the `VARIANT` structure like you would use any other variable in an argument to a functin call.