

## **COM: A White Paper from PowerBASIC, Inc.**

Com Client support is new in PBWin 7.0 from [PowerBASIC, Inc.](#)

The COM acronym represents the "Component Object Model", which describes a method of building and accessing "Components". These are reusable chunks of code and associated data, which may be accessed by "COM-Aware" applications. One of the most frustrating things about this technology has been the ever-changing list of buzz-words used to describe it. We've evolved through OLE, VBX, and OCX, to the more current COM and ActiveX. Though nuances of difference abound, the important thing to remember is that COM and ActiveX describe a means of accessing code and data. COM+ refers to some extensions that are specific to Win2000. Throughout this discussion, we'll use the terms COM Object and ActiveX Object to describe components: reusable chunks of code and associated data.

First and foremost, some buzzword definitions are in order:

**ActiveX Client:** An application that uses the services offered by an ActiveX Server (Component). A client is also referred to as an ActiveX Controller because it controls the server: It chooses which services of the component will be utilized, when they will be utilized, and the purpose of each reference.

**ActiveX Container:** An application that can accept an ActiveX Control (with a visual user interface) or an ActiveX Document Object.

**ActiveX Control:** A special form of ActiveX component that offers a visual user interface to work like a Windows Control.

**ActiveX Component:** A COM Object, acting as a server, which can be used by other programs to provide needed functionality.

ActiveX Document Object: A document that can be embedded into an ActiveX Container.

ActiveX Server: A COM Object (Component) which can be used by other programs to provide needed functionality. In most cases, the terms ActiveX Component and ActiveX Server may be used interchangeably.

A COM Object may reside in a DLL, in which case it is known as an "In-Process" server because it shares address space with the client, or controlling application. It may reside in an EXE, in which case it is known as an "Out-of-Process" server because it does not share address space. A COM Object may even reside on another computer, even if it has a different architecture and/or operating system. In that case, communication between client and server, over a local network or even the Internet, is handled automatically by Windows, through a transparent process known as DCOM or Distributed COM.

COM Objects may be referenced from any Win32 PowerBASIC program using OLE Automation and the Dispatch interface to that object. An object could be described simplistically as a set of functions and some associated data. The internal implementation of the functions, and the variables they use, are hidden from the outside world by this concept of encapsulation. A COM Object is known as a server, because it publishes an interface of functions through which other programs may communicate. The client application, which is also known as the controller, communicates with the server by calling one or more of the published functions. It is never possible to reference object data directly, as it is only assigned or retrieved through a call to an object function.

A new class of variables is introduced. They are known generally as object variables, and may only contain a reference to a COM Object, and no other value of any kind. An object variable must be declared by its specific class, which could be

the generic "Dispatch" class, or one which you explicitly define with an INTERFACE structure. An object variable may only be used in specific situations, such as execution of an object method. It is never legal to reference them in normal numeric or string expressions, nor is it possible to even PRINT their value without the use of the special new functions like OBJPTR().

A Dispatch object is one that communicates through the IDispatch interface, which is the most flexible method, as it supports both named and optional parameters. While an object may support hundreds of functions (known as Methods or Properties) in Dispatch Objects, they are all accessed through a single entry point, called Invoke.

So how does the object tell one from another? By encoding every function (Method or Property), and every named parameter as a special numeric, long integer value known as a Dispatch ID, or DispID for short. When a Dispatch Client calls a method in a Dispatch Object, it passes one or more DispID's to tell the object precisely which Method and which parameters it expects to utilize. Luckily, PowerBASIC handles most all of these messy details for you, so all you really need to know are the names of the functions and params. Dispatch objects require that all parameters, return values, and assignment values be passed only as Variant Variables.

PowerBASIC can access Dispatch Objects through "Early Binding" or "Late Binding". Late binding, the simpler method, offers the most flexibility. You don't need to declare Methods, Properties, and parameters in advance, as everything is resolved at the time the program is executed. A Dispatch Object knows a lot about itself, including information about all of its member functions. However, for that very reason, the validity of these references is not checked at compile time. Virtually everything is done at run time.

To access a COM Object through late binding, you must first declare a variable to be of the generic object class "Dispatch".

Then, you use the SET statement to create an instance of the Dispatch Object in a specific server by referencing its unique ProgID. At this point, you can finally use the OBJECT statement to communicate with the COM server:

```
Dim OWord as Dispatch
Dim DocNum as Variant
Set OWord = Dispatch in "Word.Application.8"
Object Get OWord.Documents.Count to DocNum
Object Call OWord.Quit
Set OWord = Nothing
```

The above code snippet shows how you can access an active instance of Microsoft Word, to ascertain the number of documents which are currently open. When PowerBASIC executes the OBJECT GET statement, this is a general idea of what happens "Behind the Scenes"...

The Object variable OWord is checked to see that it references an instance of Microsoft Word.

The server is asked to verify that it supports a "Documents" collection of Document Objects.

The server responds affirmatively and returns a Dispatch ID of the value 6 for the client to use to identify it.

The server is instructed to execute function 6. It responds by creating a Document object, returning a reference to it.

The server is asked to verify that it supports a "Count" property within the Document object.

The server responds affirmatively and returns a Dispatch ID of the value 2 for the client to use to identify it.

The server is instructed to execute function 2. It responds by returning a Variant variable that contains that value.

The temporary Document object is destroyed.

An involved process? You bet it is, but it does offer functionality that couldn't readily be duplicated in any other way. However, did you notice that there are some ways this functionality could be optimized? One thing is obvious... Why not look up the values of the Dispatch ID's ahead of time? That certainly works, and it's known as "Dispatch Early Binding". It takes a little more work from you, but it offers a big boost in performance. All you have to do is determine the value of the needed DispID's, and tell PowerBASIC about it in an INTERFACE DISPATCH structure...

```
Interface Dispatch MSWord
  Member Get Documents<6>() as Document
End Interface
```

```
Interface Dispatch Document
  Member Get Count<2>()
End Interface
```

You can list every Method/Property in the interface, or just the ones you reference in your code. They can appear in any sequence. The only thing we're doing here is associating each Method/Property name, and each Named Parameter with the appropriate DispID value, and identifying any property which returns an object to be used in a nested object reference like "Document". You can look up the DispID's of your COM Servers using an Object Browser, or by reading the object documentation. If a Member offers named parameters, they're declared in a similar fashion...

```
Member Call Insert<4>(Line<5>)
```

Finally, you can even insert some additional information about each parameter, the internal type, and the Member return value for your own reference, even though the compiler doesn't use it. Namely, each parameter may be preceded with: Optional, In, Out, InOut)

```
Member Call Insert<4>(In Line as String<5>) as String
```

Put it all together, and you have...

```
Interface Dispatch MSWord
  Member Get Documents<6>() as Document
End Interface

Interface Dispatch Document
  Member Get Count<2>()
End Interface

Dim OWord as MSWord
Dim DocNum as Variant
Set OWord = MSWord in "Word.Application.8"
Object Get OWord.Documents.Count to DocNum
Object Call OWord.Quit
Set OWord = Nothing
```

The source code looks surprisingly similar, but in many cases, you'll find that the execution speed is remarkably faster.