OPC and .NET

Beta 2 edition

A Whitepaper by

http://www.viscomvisual.com/dotnet/

email to the VISCOM .NET team: dotnet@viscomvisual.com

Version 0.2 2001-07-09 13:29

Trademarks and Copyrights

Contents

General Provisions

The ideas, concepts, information, pictures, files and source code provided within this whitepaper and package can only be used under the following provisions to you ("the User"):

- The User must have a legal license for Microsoft .NET SDK Beta 2 and Microsoft Visual Studio.NET Beta 2.
- The User never understands this whitepaper and files as part of any standards or products like OPC or .NET
- The User accepts this whitepaper and files simply as an example for programming with .NET and OPC.
- The User never shares this whitepaper and files to any others, he simply passes links to our web site : http://www.viscomvisual.com/dotnet/.

source code and sample limitations

- the user keeps in mind that these are Technology Preview samples based on early beta technologies.
- this source code only shows basic ideas and concepts, but in no way production quality code.
- error and exception handling was left out on multiple areas where in fact required.
- memory leaks will show up.
- any new beta or release version of .NET will break some code.
- the user must have a working .NET development environment ,
  OPCDA 2.0 server+proxies and OPCEnum installed and running.
- Development and testing was done on a Windows 2000 SP2 system, any others may not work.
- OPC with DCOM to a remote machine is not yet implemented.
- the source code assumes some optional interfaces and features as most often provided by OPCDA servers.
- Multithreading / Apartment limitations to be defined...
- while samples show client side use of OPC interfaces, the presented concept could also work for servers, making it possible to write OPC servers in any .NET language.
- speed/performance comparisons are unrealistic at this beta stage.

Background

OPC is a widely used standard in industrial automation and uses the well established Microsoft Windows COM/DCOM technology as it's base.

Within the last few years, a huge base of products building on the OPC interfaces were released from a wide range of companies in different markets.

With the upcoming Microsoft .NET Framework, there are new concepts of communication between components and applications, named **Remoting and Reflection**.

Also in the work are new OPC standards based on XML, but these will provide solutions to somewhat different problems like internetworking and OS-independence.

In contrast, our focus with this whitepaper is the huge installed base of OPC servers.

The new .NET Framework will provide some interoperability layers and tools to reuse a large part of the existing COM/ActiveX components, but with some strong limitations.

We think it is an important job to make sure OPC as an excellent standard can immediately be used again with all the new .NET applications to come. So this whitepaper and samples should help any interested developers to learn how to keep working on proved solutions.

OPC standards are defined at 'two different layers' of COM/DCOM. First, as a collection of COM custom interfaces, and secondly as COM-automation compliant components.

So with this whitepaper we will elaborate the use of OPC at this two layers.

Also note the scope of this whitepaper and samples is at the primary OPC standard category, OPCDA (Data Access).

Conclusion:
.NET is a first class citizen in the Automation World

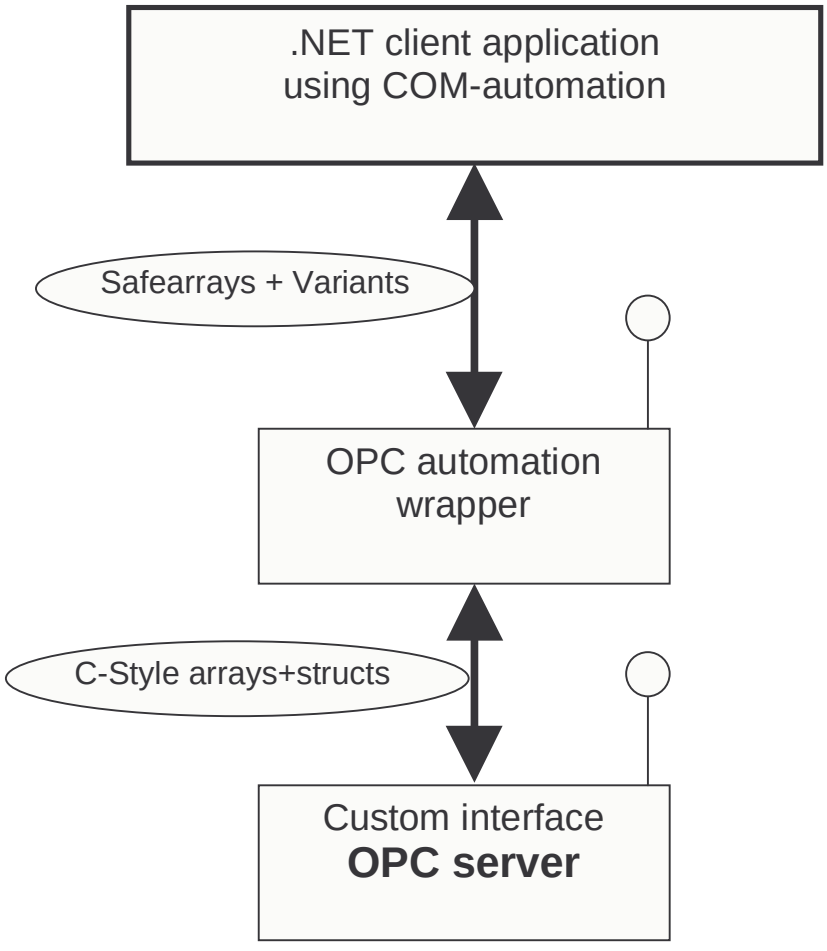
Interface levels

Automation Interface

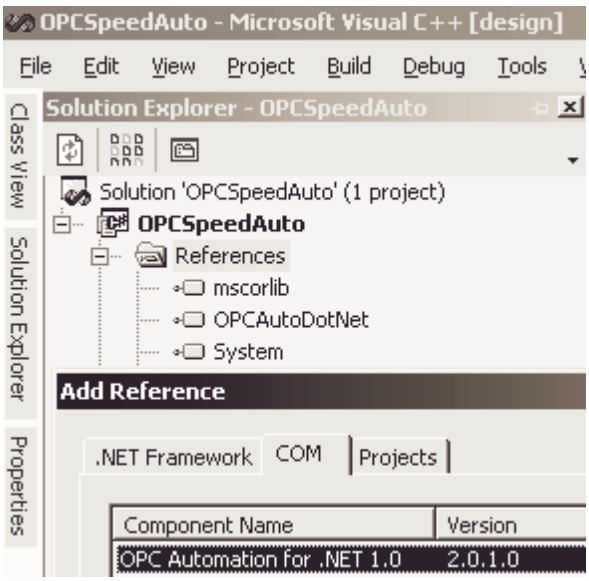
The OPC COM-automation level component can be built as a wrapper for any OPC-server at custom level. In this case it simply is a converter from custom data types and structs to COM-Variants and Safearrays, and to expose the automation look-and-feel as expected by clients like VisualBasic.

Problem
OPC defines the automation interface to use Safearrays as one-based (*Option Base 1*). But the new .NET runtime and its marshaler only supports (at the time of Beta 2) arrays with zero as the lower bound.

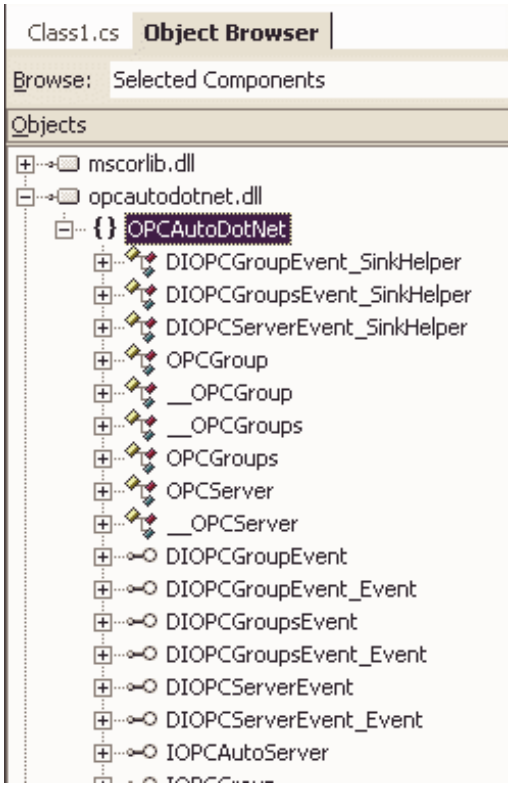
Besides that, the automation interfaces can be used from any .NET application. To prove this, we changed the source of the wrapper (available on request from VISCOM) to use zero based arrays.



Using the modified automation wrapper in .NET was simple:



Add a reference to the COM component **"OPC Automation for .NET 1.0"**. VisualStudio.NET generates the metadata-DLL, also including some helpers for events:



then the OPC-objects are useable as before in VB6.

Custom interface

To understand the issues with COM custom interfaces and the .NET framework, we must first analyze, why automation components can be used immediately.

VisualStudio.NET relies on the information found in a type-library for every imported COM component (e.g. the library generated by MIDL-compiler, named *.TLB).

The problem is now, type libraries can only contain automation compliant information. So if we compile a custom-interface IDL file, the generated TLB misses very important type descriptions, especially the method call parameter size (e.g. of arrays).

Solutions

At the time of .NET Beta 2, there's no tool (like TLBIMP) to import custom interface libraries. The workarounds are: writing a custom Marshaler in (managed-) C++, or the method we used, to write some marshaling helper classes in a managed language (here C#).

Managed marshaling code makes use of the framework services provided in the **System.Runtime.InteropServices** namespace, especially the **Marshal** class.

The other work we did was to rewrite the custom interfaces from the OPCDA IDL in C#. Important was to correctly define the non-integral parameters (arrays/pointers to arrays/structs...) as the .NET special type **IntPtr** . This is handled as a generic pointer to unmanaged memory. With this done, .NET has all the information (**metadata**) to execute calls to custom interfaces.

Next, the marshaling helper classes have to assemble and pack all input [in] parameters, call the interface and then disassemble/unpack all returned or output [out] data. To do all this, use of the **System.Runtime.InteropServices.Marshal** class member functions are required like:

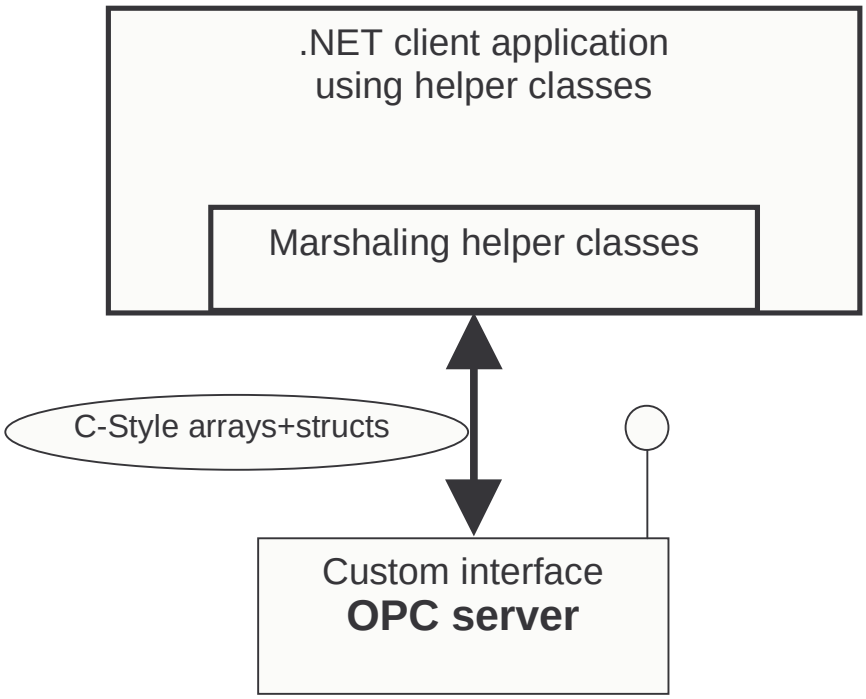
```
ReadInt32()          AllocCoTaskMem()    FreeCoTaskMem()
ReleaseComObject()   PtrToStringUni()    SizeOf()
GetObjectForNativeVariant()              DestroyStructure()
Copy()               StructureToPtr()    ThrowExceptionForHR()
```

As COM-calls return **HRESULT** result codes, it is sometimes useful to handle failed or especially the success-code S_FALSE within the helper classes. To do this, the interfaces had to be defined with the **[ComVisible(true), ComImport]** and methods with **[PreserveSig]** attribute, so we have a chance to handle HRESULT's ourself and not relying on the default .NET exception mapping.

The marshaling helper classes we built are also simple container/wrappers around the interfaces at OPC server- and group level.

Additionally, these classes provide the OPC callbacks in a more .NET like fashion, especially as expected in the form of events and delegates, and the parameters packed in an **EventArgs** derived parameter!

So we end up with the following simple 2-tier solution:



Conclusion:

Once this work was done, it was possible to use OPC custom interfaces in the .NET framework.

And now the magic happens:
every .NET CLR compliant language can use OPC directly,
C# and even VisualBasic.NET !

Classes

OpcServer

Wrapper class for the interfaces at server level. Sink for the Shutdown event of server connection-point. The AddGroup() and GetPublicGroup() member functions return a new instance of the OpcGroup class, see the next page.

class OpcServer
: IOPCShutdown

IOPCCommon
SetLocaleID() GetLocaleID()
QueryAvailableLocaleIDs()
SetClientName()

IOPCServer
Connect() *Disconnect()*
GetStatus() GetErrorString()
AddGroup()

IOPCServerPublicGroups
GetPublicGroup()

IOPCBrowseServerAddressSpace
QueryOrganization()
ChangeBrowsePosition()
BrowseOPCItemIDs() *Browse()*
GetItemID() BrowseAccessPaths()

IOPCItemProperties
QueryAvailableProperties()
GetItemProperties()
LookupItemIDs()

UCOMIConnectionPoint
ShutdownRequest()

OpcGroup

Wrapper class for the interfaces at group level. Sink for the Data-Callback events from group connection-point.

class OpcGroup
: IOPCDataCallback

Remove()

IOPCGroupStateMgt
SetName() GetStates()

IOPCPublicGroupStateMgt
MoveToPublic() *DeletePublic()*

IOPCItemMgt
AddItems () ValidateItems()
RemoveItems() SetActiveState()
SetClientHandles() SetDatatypes()
CreateAttrEnumerator()

IOPCSyncIO
Read () Write()

IOPCAsyncIO2
Read () Write() Refresh2 ()
Cancel2() Set/GetEnable()

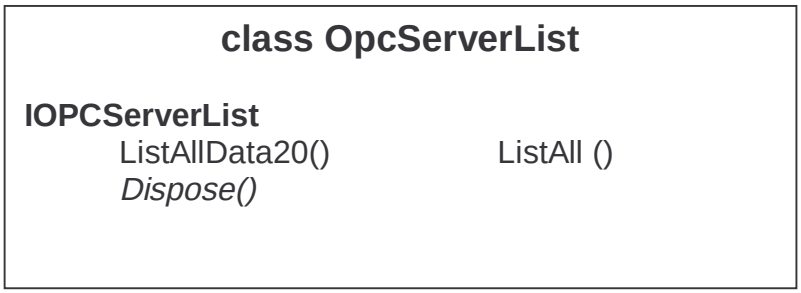
UCOMIConnectionPoint
OnDataChange()
OnReadComplete()
OnWriteComplete()
OnCancelComplete()

note: *italic* printed methods are in fact class members only, not interface members. *Remove()* is forwarded to the internal server interface as RemoveGroup().

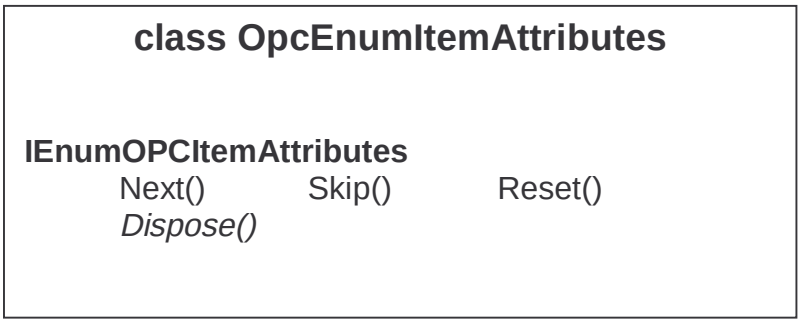
Helper classes

OpcServerList is a tiny wrapper around the interface IOPCServerList from OPCEnum (the enumerator for all installed OPC servers).

The server list is returned as a struct-array of type OpcServers[].



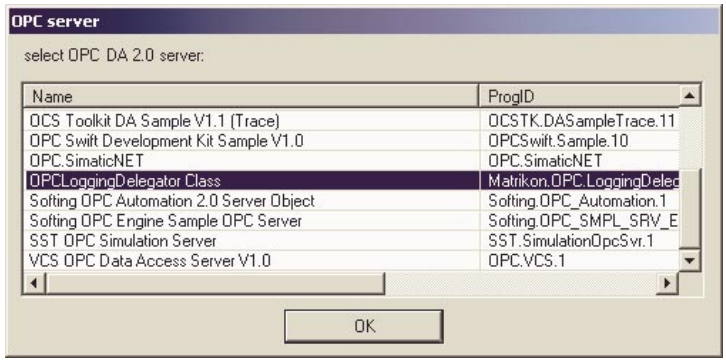
The call OpcGroup:CreateAttrEnumerator returns an instance of the OpcEnumItemAttributes class, used to enumerate item attributes. These are returned as a struct-array of type OPCItemAttributes[].



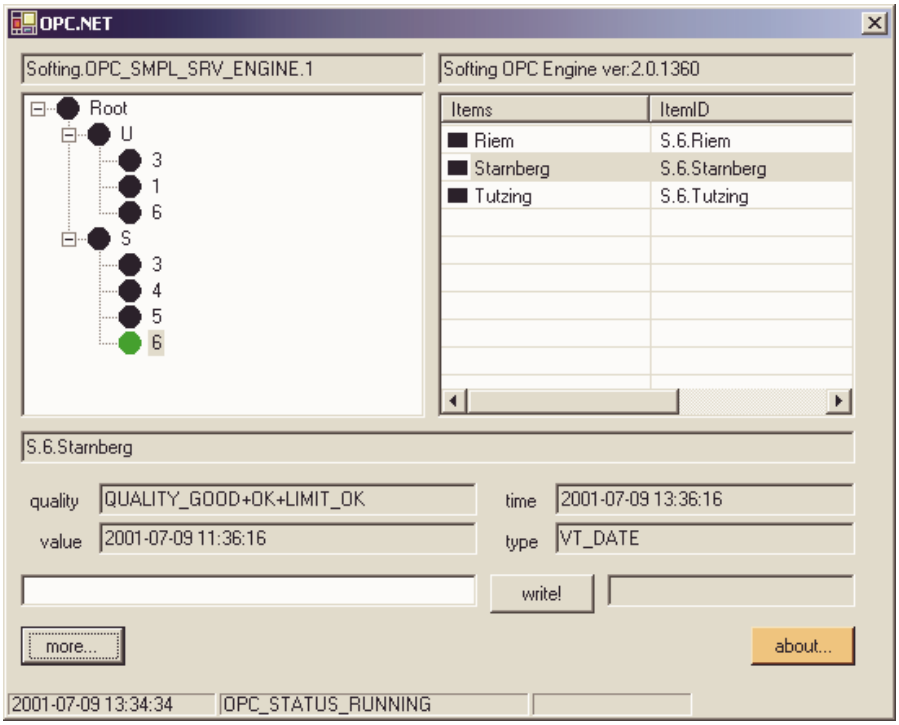
Sample client application

We built a first sample .NET WinForms application to show some functions at work:

starting **DirectOPCClient.exe** presents the first window which lets you select one from any installed OPCDA 2.0 servers:



If gracefully connected, you will see the main screen:



The sample application simply lets you browse the OPC server namespace, see the item values and even change them, if write permission is granted.

File listing

```
WhitepaperOPCdotNET.pdf          this acrobat/word document

DirectOPCClient \

     DirectOPCClient.sln         VisualStudio solution file
     DirectOPCClient.csproj      C# project file
     AssemblyInfo.cs             assembly info
     SelServer.cs                form for selecting server
     MainForm.cs                 main form
     PropsForm.cs                item properties form
     AboutForm.cs                about box form
     *.resx                      resource files

     \ bin \ Release

     DirectOPCClient.exe         sample app release build
     OPCdotNETLib.dll            library release build


OPCdotNETLib \

     OPCdotNETLib.sln            VisualStudio solution file
     OPCdotNETLib.csproj         C# project file
     AssemblyInfo.cs             assembly info
     OPC_Common.cs               OPC common interface def.
     OPC_Data.cs                 OPC-DATA 2.0 interface def.
     OPC_Data_Srv.cs             Server wrapper class impl.
     OPC_Data_Grp.cs             Group wrapper class impl.


CSSample \

     OPCCSharp.cs                simple C# console client
     …

VBSample \

     OPCBasic.vb                 simple VB.NET console client
     …
```