CODE Consulting and Development, s.r.o.
Bolevecka naves 27, 323 00 Plzen, Czech Republic
e-mail: sales09@opclabs.com, Web: www.opclabs.com
tel. +420 603 214 412, fax +420 378 600 795

QuickOPC-Classic 5.12 Concepts

Contents

Introduction **7**

Installation **8**

Operating Systems 9
Prerequisites 10
Licensing 11
Related Products 11

Product Parts **12**

Assemblies 12
XML Comments 13
COM Components 13
Management Tools 13
Development Libraries 14
Demo Application 14
Simulation OPC Server 15
License Manager 16
Documentation and Help 16

Fundamentals **18**

Typical Usage 18
Thick-client .NET applications on LAN 18
Thick-client COM applications on LAN 19
Web applications (server side) 20
Referencing the Assemblies 21
Application Configuration File Changes (Rarely Needed) 22
Namespaces 23
Referencing the Components 24
Naming Conventions 25
Components and Objects 26
Computational Objects 26
User Interface Objects 28
Stateless Approach 30
Simultaneous Operations 30
Error Handling 31
Errors and Multiple-Element Operations 32
Helper Types 33
Dictionary Object 33
Time Periods 33

OPC Quality 34
Value, Timestamp and Quality (VTQ) 35
Result Objects 35
Variant Type (VarType) 35
Element Objects 36
Descriptor Objects 37
Parameter Objects 37

OPC Data Access Tasks 38
Obtaining Information 38
Reading from OPC Items 38
Getting OPC Property Values 39
Modifying Information 40
Writing to OPC Items 40
Browsing for Information 40
Browsing for OPC Servers 41
Browsing for OPC Nodes (Branches and Leaves) 41
Browsing for OPC Access Paths 42
Browsing for OPC Properties 42
Subscribing for Information 43
Subscribing to OPC Items 43
Changing Existing Subscription 44
Unsubscribing from OPC Items 45
Item Changed Event 45
Using Callback Methods Instead of Event Handlers 46
Setting Parameters 47
OPC Common Dialogs 49
Computer Browser Dialog 49
OPC Server Dialog 50
OPC-DA Item Dialog 50
OPC-DA Property Dialog 51
OPC User Objects 52
Computer Browser Dialog 52
OPC Server Browse Dialog 53
OPC-DA Item Browse Dialog 54
OPC-DA Item Select Dialog 55

OPC Alarms and Events Tasks 57
Obtaining Information 57
Getting Condition State 57
Modifying Information 57

Acknowledging a Condition 58
Browsing for Information 58
Browsing for OPC Servers 58
Browsing for OPC Nodes (Areas and Sources) 58
Querying for OPC Event Categories 59
Subscribing for Information 60
Subscribing to OPC Events 60
Changing Existing Subscription 61
Unsubscribing from OPC Events 61
Refreshing Condition States 61
Notification Event 62
Using Callback Methods Instead of Event Handlers 63
Setting Parameters 64

EasyOPC.NET Extensions 66
Usage 66
Data Access Extensions 66
OPC Properties 66
Type-safe Access 66
Well-known Properties 67
Alternate Access Methods 67
OPC Items 67
Type-safe Access 67
Software Toolbox Extender Replacement 68

Application Deployment 69
Deployment Elements 69
Assemblies 69
Development Libraries and COM Components 69
Management Tools 70
Prerequisites 70
Licensing 72
Deployment Methods 72
Manual Deployment 72
Automated Deployment 73

Advanced Topics 74
OPC Specifications 74
OPC-UA (Universal Architecture) 74
OPC Interoperability 76
Event Logging 77
EasyOPC Options Utility 77

COM Registration and Server Types 77
Asynchronous Operations 78
Multiple Notifications in One Call 80
Internal Optimizations 81
Failure Recovery 82
Timeout Handling 83
Data Types 85
Multithreading and Synchronization 87
64-bit Platforms 89
32-bit and 64-bit Code 89
OPC on 64-bit Systems 89
Version Isolation 90

Additional Resources 91

Introduction

Are you having difficulties incorporating the OPC data into your solution? Need to do it quickly and in quality? If so, QuickOPC comes to the rescue.

QuickOPC is a radically new approach to access OPC data. Traditionally, OPC programming required complicated code, no matter whether you use OPC custom or automation interfaces. OPC Server objects must be instantiated, OPC Group objects must be created and manipulated, OPC Items must be added and managed properly, and subscriptions must be established and maintained. Too many lines of error-prone code must be written to achieve a simple goal – reading or writing a value, or subscribing to value changes.

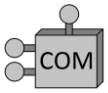
QuickOPC is a set of components that simplify the task of integrating OPC into applications. Reading a value from OPC Data Access server, or writing a data value can be achieved in just one or two lines of code! Receiving alarms from OPC Alarms and Events server is also easy.

The components can be used from various languages and environments.

QuickOPC-Classic is a product line that consists of two products: QuickOPC.NET and QuickOPC-COM. The text is document applies mostly to both products. When necessary, the differing text is marked with corresponding COM or .NET icon.

In QuickOPC.NET, the available examples show how the components can be used from C#, Visual Basic.NET, and managed C++. Windows Forms, ASP.NET pages, console applications, and WPF applications are all supported.

The development tools we have targeted primarily are Microsoft Visual Studio 2008 and Microsoft Visual Studio 2010.

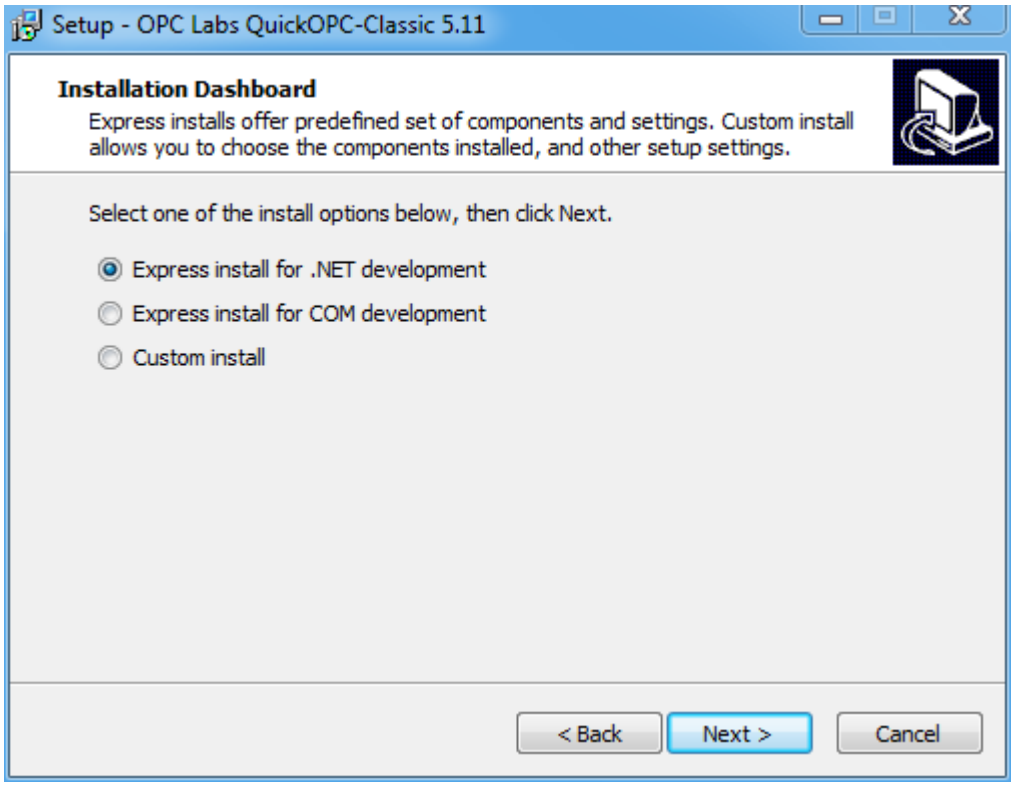


In QuickOPC-COM, the available examples show how the components can be used from Visual Basic (VB), C++, VBScript (e.g. in ASP, or Windows Script Host), JScript, PHP, Visual Basic for Applications (VBA, e.g. in Excel), Visual FoxPro (VFP), and other tools. Any tool or language that supports COM Automation is supported.

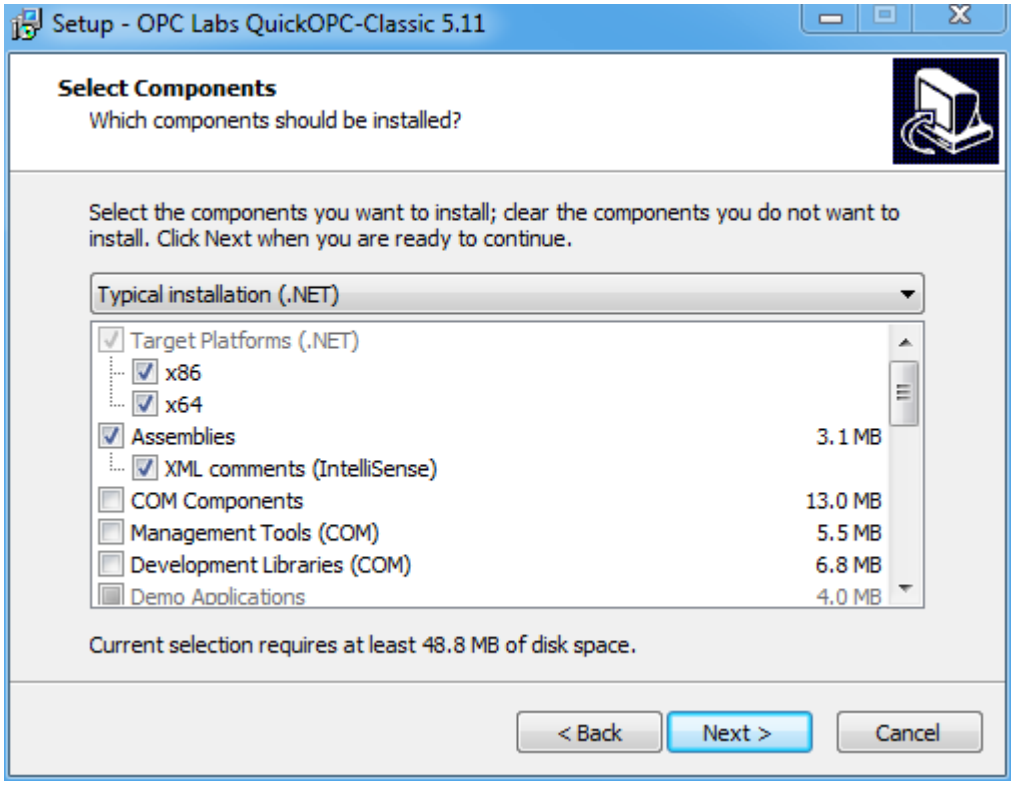
Installation

The installation can be started by running the setup program. Just follow the on-screen instructions. The installation program requires that you have administrative privileges to the system.

After an introductory screen, the setup wizard offers you the basic installation options:



For start, simply choose one of the “express” install options. If you decide to select “custom install”, the installation program then offers you several installation types, and also allows you to choose specifically which part of the product to install. In addition, with the “custom install”, you can also influence additional settings such as the destination location, and whether to automatically Launch the License Manager utility.



When the installation is finished, it opens the Quick Start document. You can access the documentation and various tools from your Start menu.

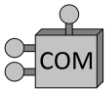
The product includes an uninstall utility and registers itself as an installed application. It can therefore be removed easily from Control Panel. Alternatively, you can also use the Uninstall icon located in the product's group in the Start menu.

Operating Systems

The product is supported on following operating systems:

- Microsoft Windows XP with Service Pack 2 or later (x86)
- Microsoft Windows Vista with Service Pack 1 or later (x86 or x64)
- Microsoft Windows 7 (x86 or x64)
- Microsoft Windows Server 2003 with Service Pack 1 or later (x86 or x64)
- Microsoft Windows Server 2008 (x86 or x64)
- Microsoft Windows Server 2008 R2 (x64), optionally with Service Pack 1

On x64 platforms, QuickOPC.NET can run in 32-bit or 64-bit mode.



QuickOPC.COM currently runs always in 32-bit mode, even on x64 platforms.

Prerequisites

For QuickOPC.NET, the following software must be present on the target system before the installation:

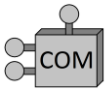
1. Microsoft .NET Framework 3.5 with Service Pack 1 (Full or Client Profile), or Microsoft .NET Framework 4 (Full or Client Profile).
   Needed when: Always (the choice depends on the framework that you target).

   The Client Profile of the .NET Framework is enough for QuickOPC.NET itself; however in some scenarios you will need the Full profile, such as when you are developing ASP.NET applications, or want to run the ASP.NET examples included with the product.

2. Adobe (Acrobat) Reader, or compatible PDF viewer.

The QuickOPC.NET setup program also installs following software on the target system, when needed for the selected set of installation components:

1. Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package ATL Security Update (x86).
2. Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package ATL Security Update (x64).
3. Microsoft Visual C++ 2010 Redistributable Package (x86).
4. Microsoft Visual C++ 2010 Redistributable Package (x64).
5. OPC Core Components 3.00 Redistributable (x86).
6. OPC Core Components 3.00 Redistributable (x64).



For QuickOPC-COM, the following software must be present on the target system before the installation:

1. Microsoft .NET Framework 3.5 with Service Pack 1.

   QuickOPC-COM does not directly require the Microsoft .NET Framework 3.5, but OPC Core Components setup may fail without it.

   Microsoft .NET Framework 3.5 is also needed for OPC UA COM Interop Components. QuickOPC-COM does not use it directly. This is only needed if you want to connect to OPC-UA (Universal Architecture) servers, and you check “OPC UA COM Interop Components” in the selection of components to be installed.

2. Adobe (Acrobat) Reader, or compatible PDF viewer.

The QuickOPC-COM setup program also installs following software on the target system:

1. Microsoft Visual C++ 2010 Redistributable Package (x86)
2. OPC Core Components 3.00 Redistributable (x86)

Licensing

QuickOPC is a licensed product. You must obtain a license to use it in development or production environment. For evaluation purposes, you are granted a trial license, which is in effect if no other license is available. The QuickOPC.NET and QuickOPC-COM parts are licensed separately.

With the trial license, the components only provide valid OPC data for 30 minutes since the application was started. After this period elapses, performing OPC operations will return an error. Restarting the application gives you additional 30 minutes, and so on. If you need to evaluate the product but the default trial license is not sufficient for you purposes, please contact the vendor or producer, describe your needs, and a special trial license may be provided to you.

The licenses are installed and managed using a License Manager utility, described further in this document.

Related Products

Additional products exist to complement the base QuickOPC.NET offering. Check the options available with your vendor.

Product Parts

Assemblies

At the core of QuickOPC.NET there are .NET assemblies that contain reusable library code. You reference these assemblies from the code of your application, and by instantiating objects from those assemblies and calling methods on them, you gain the OPC functionality.

The assembly files are installed into a subdirectory called **Assemblies** under the installation directory of the product. For easy recognition among other assemblies when used in a larger context, all assemblies start with "**OpcLabs.**" prefix.

Following assemblies are part of QuickOPC.NET:

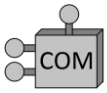
Assembly Name	File	Title	Description
OpcLabs.BaseLib	OpcLabs.BaseLib.dll	OPC Labs Base Library	Supporting code
OpcLabs.EasyOpcClassic-Internal	OpcLabs.EasyOpcClassic-Internal.dll	EasyOPC.NET Internal Library	Supporting code
OpcLabs.EasyOpcClassic	OpcLabs.EasyOpcClassic.dll	EasyOPC.NET Library	Contains classes that facilitate easy work with various OPC specifications, such as OPC Data Access and OPC Alarms and Events.
OpcLabs.EasyOpcClassic-Forms	OpcLabs.EasyOpcClassic-Forms.dll	EasyOPC.NET Forms	Contains classes that facilitate easy work with OPC Data Access and OPC Alarms and Events from Windows Forms applications.
OpcLabs.EasyOpcClassic-Extensions	OpcLabs.EasyOpcClassic-Extensions.dll	EasyOPC.NET Extensions	Extends functionality of OpcLabs.EasyOpcClassic

QuickOPC.NET components were consciously written to target Microsoft .NET Framework 3.5, i.e. they do not depend on features available only in the later version of the framework. As such, you can use the components in applications targeting version 3.5 or 4.0 of the Microsoft .NET Framework.

For the curious, QuickOPC.NET has been developed in Microsoft Visual Studio 2010 (with the use of Visual Studio 2008 platform toolset for .NET Framework 3.5 target). The layers that directly use COM (such as the **OpcLabs.EasyOpcClassic** assembly) are written in managed C++. More precisely, they contain mixed mode assemblies, where the bulk of the code is in MSIL instructions, with a few exceptions where necessary. All other parts are written in pure C#.

XML Comments

Together with the .DLL files of the assemblies, there are also .XML files that contain XML comments for them. The texts contained in these files are used by various tools to provide features such as IntelliSense and Object Browser information in Visual Studio.



COM Components

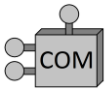
At the core of QuickOPC-COM there are COM components that contain reusable library code. You reference these components from the code of your application, and by instantiating objects from those components and calling methods on them, you gain the OPC functionality.

The component files are installed into a subdirectory called **Bin** under the installation directory of the product. Following components are part of QuickOPC-COM:

Component Name	File(s)	Title	Description
EasyOPC	easyopci.dll easyopcl.exe	EasyOPC Component	Contains classes that facilitate easy work with OPC Data Access and OPC Alarms and Events.
OPCUserObjects	OPCUserObjects.exe	OPC User Objects	Contains classes that make it easy to include OPC-related user interface in your application.

QuickOPC-COM components were consciously written so that a broad range of COM automation clients can use them, without limitation to programming language or tools used.

For the curious, QuickOPC-COM has been developed in Microsoft Visual Studio 2010, and is written in C++.

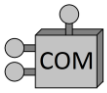


Management Tools

Management tools allow you to configure and monitor the QuickOPC-COM components. The setup program installs following management tools:

EasyOPC Options Application: Use this utility to configure the desired behavior of EasyOPC component. EasyOPC component comes with predefined settings that are suitable for most applications. For large-volume operations, or specialized needs, it may be necessary to fine-tune the settings.

Event Log Options Application: Use this utility to configure how errors and events will be generated and logged.



Development Libraries

In Microsoft COM, the components are described by their corresponding Type Libraries. Type libraries are binary files (.tlb, .dll or .exe files) that include information about types and objects exposed by an ActiveX (COM) application. A type library can contain any of the following:

- Information about data types, such as aliases, enumerations, structures, or unions.
- Descriptions of one or more objects, such as a module, interface, **IDispatch** interface (dispinterface), or component object class (coclass). Each of these descriptions is commonly referred to as a typeinfo.
- References to type descriptions from other type libraries.

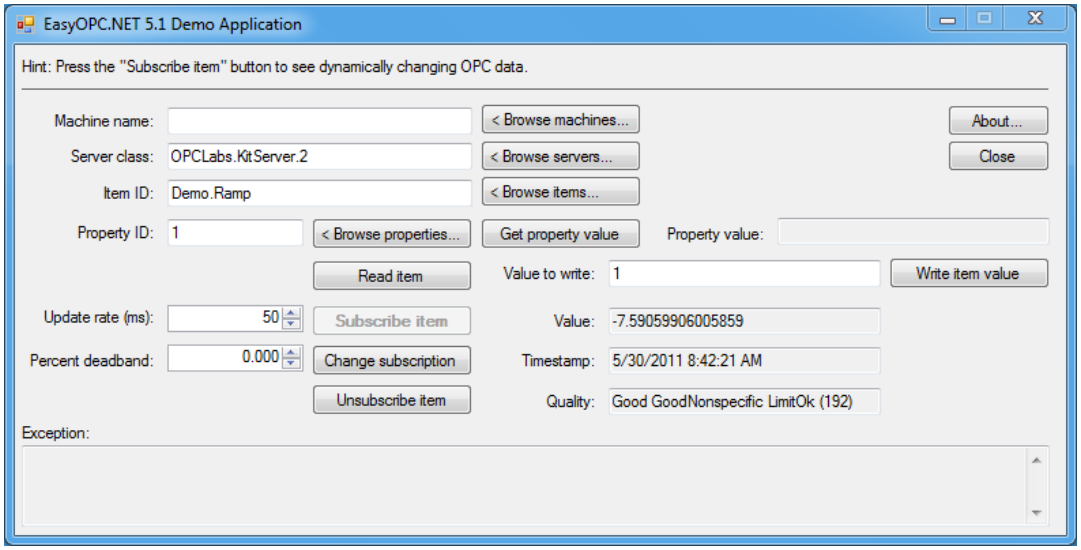
By including the type library with QuickOPC-COM, the information about the objects in the library is made available to the users of the applications and programming tools.

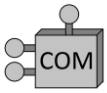
QuickOPC-COM comes with following type libraries:

- **OPC Labs EasyOPC Type Library** (Version 5.1, in file easyopct.dll): For EasyOPC Component.
- **OPC Labs OPC User Objects Type Library** (Version 5.1, in file OPCUserObjects.exe): For OPC User Objects.

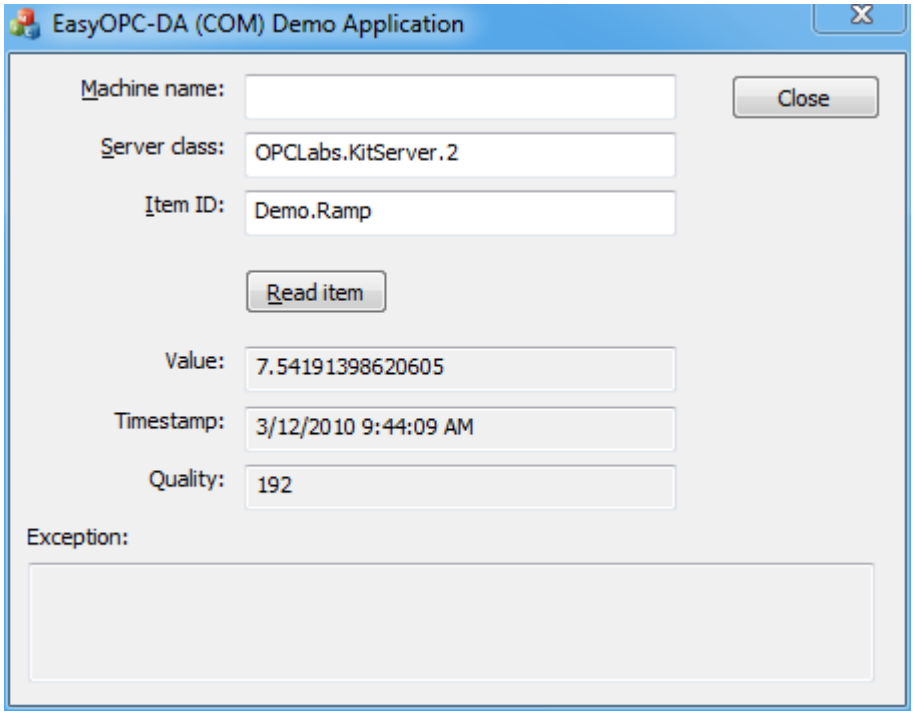
Demo Application

QuickOPC.NET installs with a demo application that allows exploring various functions of the product. The demo application is available from the Start menu.





QuickOPC-COM installs with a demo application that allows exploring basic functions of the product. The demo application is available from the Start menu.



Simulation OPC Server

To demonstrate capabilities of QuickOPC, some OPC server is needed. The demo application installed with the product, and most examples use an OPC Simulation

Server that is installed together with QuickOPC. The server's ProgID is "**OPCLabs.KitServer.2**".

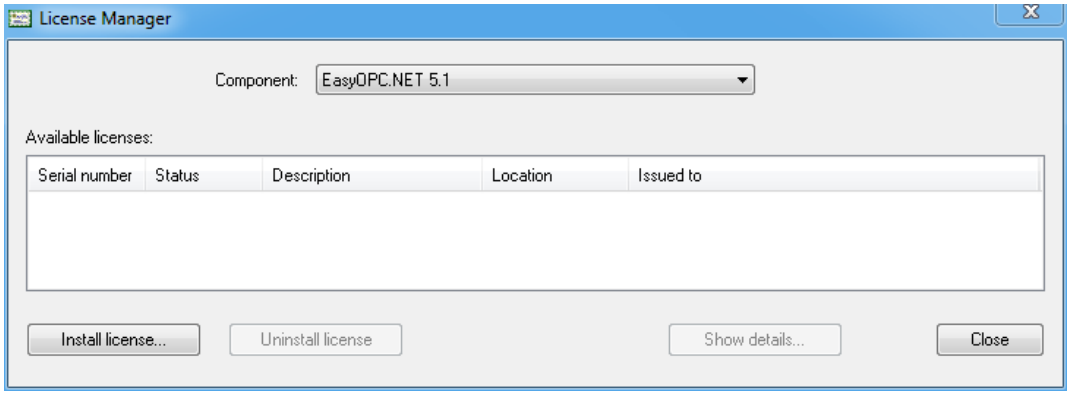
The demo application and the examples are designed to connect to the Simulation OPC Server in its default configuration (i.e. as shipped). In fact, some very simple examples connect to just one OPC item, named "**Demo.Ramp**". There are various other OPC items in this server that you can use in your own experiments too.

Note: On 64-bit platforms, the installation program still registers the 32-bit (x86) binary of Simulation OPC Server. This configuration gives better OPC compatibility. The 64-bit binary of Simulation OPC Server is also installed to the disk, and can be registered manually if needed.

License Manager

The License Manager is a utility that allows you to install, view and uninstall licenses.

In order to install a license, invoke the License Manager application (from the Start menu), and press the **Install license** button. Then, point the standard Open File dialog to the license file (with .BIN) extension provided to you by your vendor.



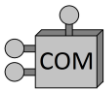
Note: You need administrative privileges to successfully install and uninstall licenses.

Documentation and Help

The documentation consists of following parts:

- Concepts (this document).
- Quick Start. Short step-by-step instructions to create your first project.
- Reference. The reference documentation is in .CHM format (Microsoft HTML Help), and formatted according to Visual Studio style and standards.

- What's New. Contains information about changes made in this and earlier versions.
- Bonus Material document.
- Examples document.



- EasyOPC Options Help. Describes the EasyOPC Options Application.

You can access all the above mentioned documentation from the Start menu.

In addition, there is IntelliSense and Object Browser information available from Visual Studio environment.

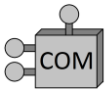
The QuickOPC.NET help content integrates with Microsoft Visual Studio 2008 Help (Microsoft Help 2 format) and Microsoft Visual Studio 2010 Help (Microsoft Help Viewer 1.0 format).

Fundamentals

This chapter describes the fundamental concepts used within QuickOPC component. Please read it through, as the knowledge of Fundamentals is assumed in later parts of this document.

Typical Usage

QuickOPC.NET is suitable for use from within any tool or language based on Microsoft .NET framework. There are many different scenarios for it, but some are more common.

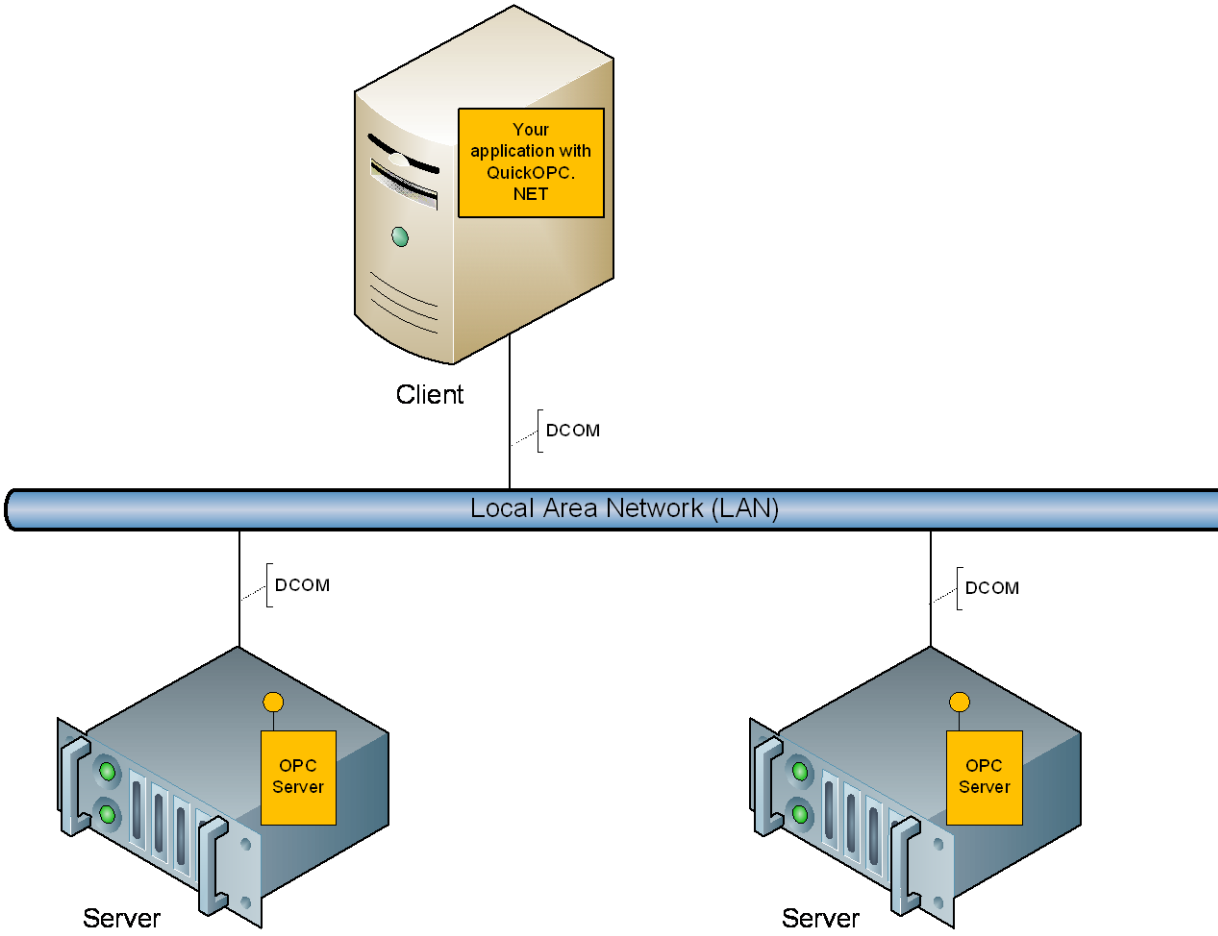


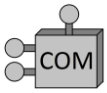
QuickOPC-COM is suitable for use from within any tool or language based on Microsoft (COM) Automation. There are many different scenarios for it, but some are more common.

Thick-client .NET applications on LAN

The most typical use of QuickOPC.NET involves a thick-client user application written in one of the Microsoft .NET languages. This application uses the types from QuickOPC.NET object model, and accesses data within OPC servers that are located on remote computers on the same LAN (Local Area Network). The communication with the target OPC server is performed by Microsoft COM/DCOM technology.

The following picture shows how the individual pieces work together:

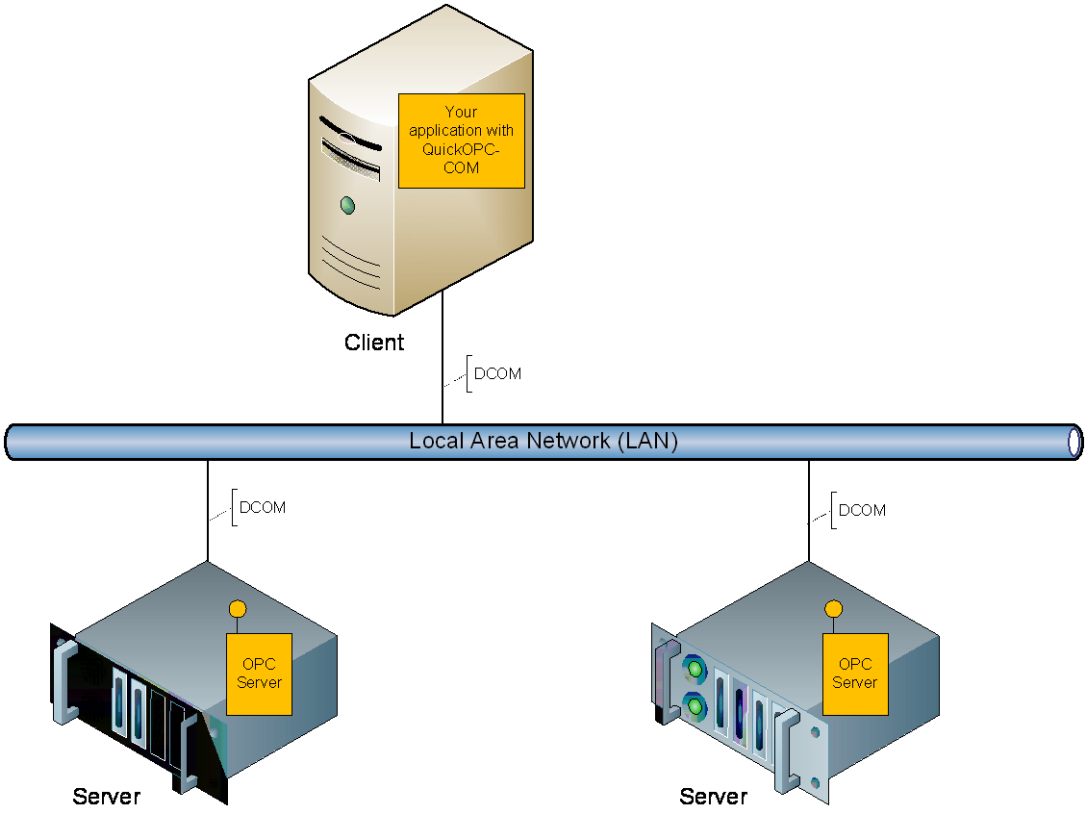




Thick-client COM applications on LAN

The most typical use of QuickOPC-COM involves a thick-client user application written in a tool or language that supports COM automation. This application uses the types from QuickOPC-COM object model, and accesses data within OPC servers that are located on remote computers on the same LAN (Local Area Network). The communication with the target OPC server is performed by Microsoft COM/DCOM technology.

The following picture shows how the individual pieces work together:



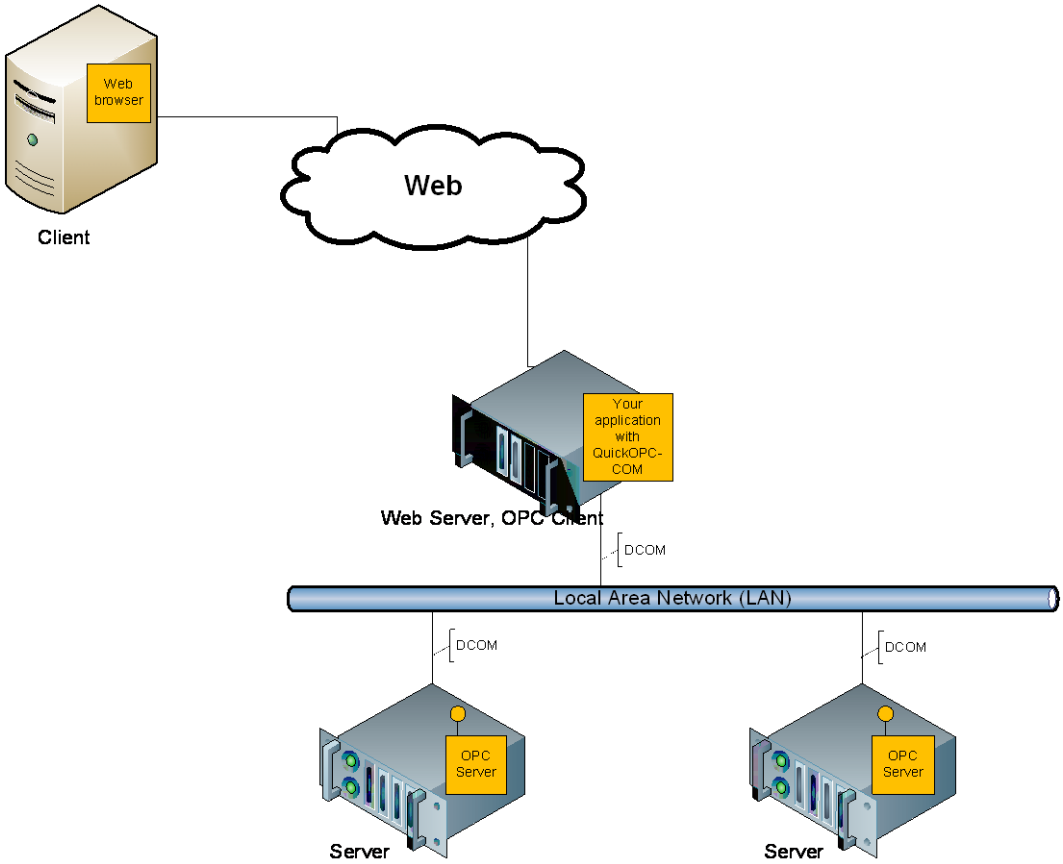
Web applications (server side)

The other typical use of QuickOPC is to place it on the Web server inside a Web application. The Web application provides HTML pages to the client's browser, runs in a Web server, such as Microsoft IIS (Internet Information Server), and is written using tools and languages such as

- ASP/VBScript, PHP, Python and others – any language that supports COM automation, or
- ASP.NET, C#, Visual Basic.NET, or any other .NET language.

The Web application uses the types from QuickOPC object model, and accesses data within OPC servers that are located on remote computers on the same LAN (Local Area Network). The communication with the target OPC server is performed by Microsoft COM/DCOM technology. No OPC-related (or indeed, COM or Microsoft-related) software needs be installed on the client machine; a plain Web browser such as Internet Explorer (IE) or FireFox is sufficient.

The following picture shows how the individual pieces work together:

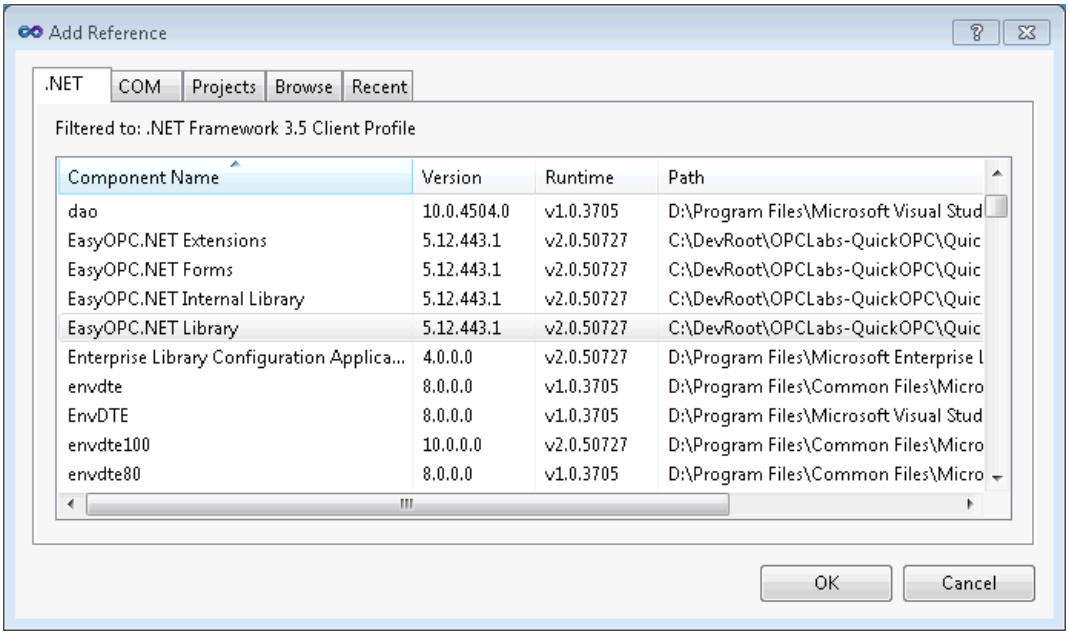


Referencing the Assemblies

Your application first needs to reference the QuickOPC.NET assemblies in order to use the functionality contained in them. How this is done depends on the language and tool you are using:

- For Visual Basic in Visual Studio, select the project in Solution Explorer, and choose Project -> Add Reference command.
- For Visual C# in Visual Studio, select the project in Solution Explorer, and choose Project -> Add Reference command.
- For Visual C++ in Visual Studio, select the project in Solution Explorer, and choose Project -> References command.

You are then typically presented with an “Add Reference” dialog. The QuickOPC.NET assemblies should be listed under its .NET tab. Select those that you need (see their descriptions in “Product Parts” chapter), and press OK.



Most projects will need “EasyOPC.NET Library”; some will also need “EasyOPC.NET Forms” or “EasyOPC.NET Extensions” component.

If you are using the Visual Studio Toolbox (described further below) to add instances of components to your project, the assembly references are created for you by Visual Studio when you drag the component onto the designer’s surface.

Application Configuration File Changes (Rarely Needed)

If you are targeting .NET Framework 4, in certain rare situations, you need to configure your application to properly load the core mixed-mode assembly of QuickOPC. In order to do so, add useLegacyV2RuntimeActivationPolicy="true" to ‘startup’ element in your application configuration file.

This setting influences how the CLR activated the mixed mode assemblies targeting .NET Framework 3.5 (CLR 2.0) that are part of EasyOPC.NET. The components include a loader code that attempt to configure this setting for you, so that .NET Framework 4 applications can use the EasyOPC.NET assemblies seamlessly. If, however, your application happens activate other CLR 2.0 assemblies using a .NET Framework 4 application policy before the EasyOPC.NET loader gets a chance to configure the setting, you will get a following exception (as an inner exception of type loading attempt failure): “Mixed mode assembly is built against version 'v2.0.50727' of the runtime and cannot be loaded in the 4.0 runtime without additional configuration information.”

In such case, an additional setting is needed in the application configuration file. In Visual Studio, you typically design the contents of the application configuration file is in the **app.config** file in your project. If you do not have this file in your project, add it first. After making this change, the file may look like this:

```
<?xml version="1.0"?>
<configuration>
<startup useLegacyV2RuntimeActivationPolicy="true">
  <supportedRuntime version="v4.0"/>
</startup>
</configuration>
```

Namespaces

The QuickOPC.NET class library is made up of namespaces. Each namespace contains types that you can use in your program: classes, structures, enumerations, delegates, and interfaces.

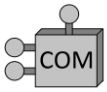
All our namespaces begin with **OpcLabs** name. QuickOPC.NET defines types in following namespaces:

Namespace Name	In Assemblies	Description
OpcLabs.EasyOpc	OpcLabs.EasyOpcClassic OpcLabs.EasyOpcClassic-Extensions	Contains classes that facilitate easy work with various OPC specifications (i.e. common functionality that is not tied to a single specification such as OPC Data Access or OPC Alarms and Events).
OpcLabs.EasyOpc.AlarmsAndEvents	OpcLabs.EasyOpcClassic OpcLabs.EasyOpcClassic-Extensions	Contains classes that facilitate easy work with OPC Alarms and Events.
OpcLabs.EasyOpc.DataAccess	OpcLabs.EasyOpcClassic OpcLabs.EasyOpcClassic-Extensions	Contains classes that facilitate easy work with OPC Data Access.
OpcLabs.EasyOpc.DataAccess.Forms	OpcLabs.EasyOpcForms	Contains classes that facilitate easy work with OPC Data Access from Windows Forms applications.

You can use symbols contained in the namespaces by using their fully qualified name, such as **OpcLabs.EasyOpc.DataAccess.EasyDAClient**. In order to save typing and achieve more readable code, you will typically instruct your compiler to make the namespaces you use often available without explicit reference. To do so:

- In Visual Basic, place the corresponding **Imports** statements at the beginning of your code files.

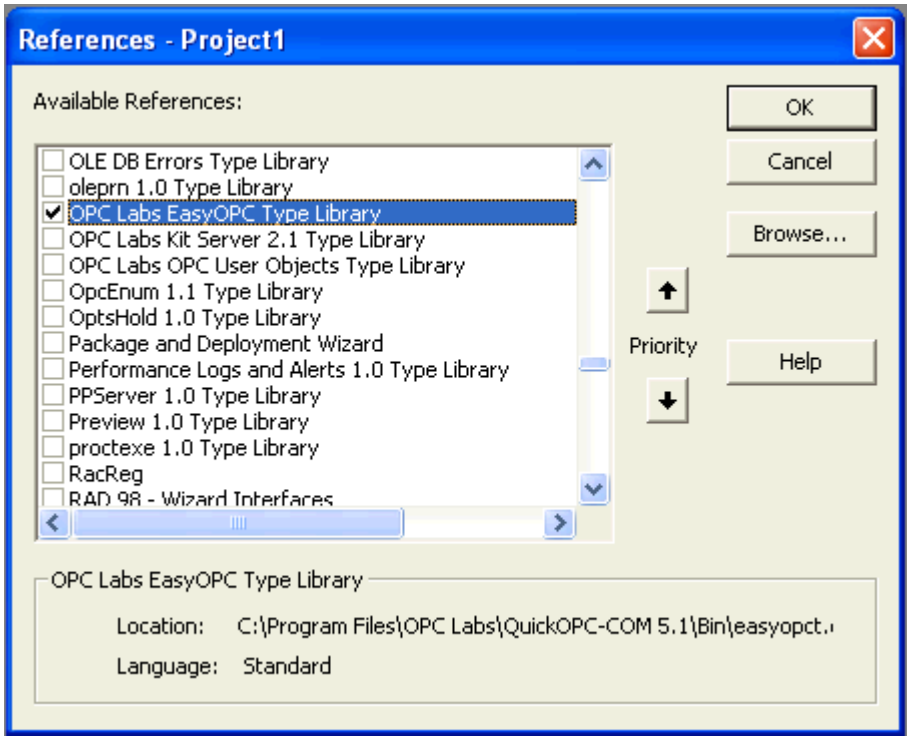
- In Visual C#, place the corresponding **using** directives at the beginning of your code files.
- In Visual C++, place the corresponding **using namespace** directives at the beginning of your code files.



Referencing the Components

Your application first needs to reference the QuickOPC-COM components in order to use the functionality contained in them. How this is done depends on the language and tool you are using.

In Visual Basic 6.0 and Visual Basic for Applications (e.g. Microsoft Excel, Word, Access, PowerPoint, and many non-Microsoft tools), select Project -> References (or Tools -> References) from the menu. You are then presented with a "References" dialog. The QuickOPC-COM type libraries should be listed in alphabetical order, prefixed with "OPC Labs" in their name so that you can easily find them grouped together:



Check the boxes next to the libraries you are referencing, and press OK.

Some tools provide a different user interface for referencing the components, while yet others do not have any user interface at all, and you need to write a source code

statement that references the type library directly. Following table contains all pieces of information that you may need to properly reference and use the components:

Type Library Name	Version	LIBID	File Name	Namespace	Description
OPC Labs EasyOPC Type Library	5.1	FAB7A1E3-3B79-4292-9C3A-DF39A6F65EC1	easyopct.dll	EasyOpcLib	Contains classes that facilitate easy work with OPC Data Access and OPC Alarms and Events.
OPC Labs OPC User Objects Type Library	5.1	965A3842-AEEA-4DF9-9241-28B963F76E24	OPCUser-Objects.exe	OPCUserObjects	Contains classes that provide user interface for OPC Data Access tasks.

For your convenience, we have listed below examples of source code reference statements in various languages and tools.

Language/ tool	Statement
C++	`#import "libid:FAB7A1E3-3B79-4292-9C3A-DF39A6F65EC1" version(5.1)  // EasyOpcLib`
WSH	`<reference guid="{FAB7A1E3-3B79-4292-9C3A-DF39A6F65EC1}" version="5.1" /> <!--OPC Labs EasyOPC Type Library-->`
ASP	`<!—METADATA TYPE="TypeLib" NAME=" OPC Labs EasyOPC Type Library" UUID="{ FAB7A1E3-3B79-4292-9C3A-DF39A6F65EC1}" VERSION="5.1"-->`

Note that referencing the type library in WSH or ASP is only needed if you want to use certain features, such as the named constants included in the library. It is not necessary to reference the type library for simply instantiating the components and making methods calls, as for method calls, VBScript or JScript code in WSH or ASP can interrogate the created objects and use late binding to perform the calls.

It is also possible to do away with referencing the component in Visual Basic 6.0 and Visual Basic for Applications, and proceed simply to instantiating the object(s) as described further below, in a way similar to VBScript, but you would lose several features such as the early-binding, IntelliSense, and ability to use symbolic objects names and enumeration constants from the type library.

Naming Conventions

In addition to being compliant with common Microsoft recommendations for names, and in QuickOPC.NET with Microsoft .NET Framework guidelines for names, QuickOPC follows certain additional naming conventions:

- Types which are specific to very simplified ("easy") model for working with OPC start with the prefix **Easy**.
- Types which are specific to OPC Data Access start with **DA** (or **EasyDA**) prefix, types which are specific to OPC Alarms and Events start with **AE** (or **EasyAE**) prefix. Types that are shared among multiple OPC specifications do not have these prefixes.

Note that the second convention works in addition and together with the fact that there are separate **DataAccess** and **AlarmsAndEvents** namespaces for this purpose too.

The above described conventions also give you a bit of hint where to look for a specific type, if you know its name. For example, if the name starts with **DA** or **EasyDA**, the type is most likely in the **OpcLabs.EasyOpc.DataAccess** namespace. Without any special prefix, the type is likely to be in the **OpcLabs.EasyOpc** namespace.

Components and Objects

QuickOPC is a library of many objects. They belong into two basic categories: *Computational objects* provide "plumbing" between OPC servers and your application. They are invisible to the end user. *User interface objects* provide OPC-related interaction between the user and your application.

Computational Objects

For easy comprehension, there is just one computational object that you need to start with, for each OPC specification: For OPC Data Access, it is the **EasyDAClient** object. For OPC Alarms and Events, it is the **EasyAEClient** object. All other computational objects are helper objects (see further in this chapter). You achieve all OPC computational tasks by calling methods on one of these objects. The remainder of this paragraph describes the use of **EasyDAClient** object; the steps for **EasyAEClient** object are similar.

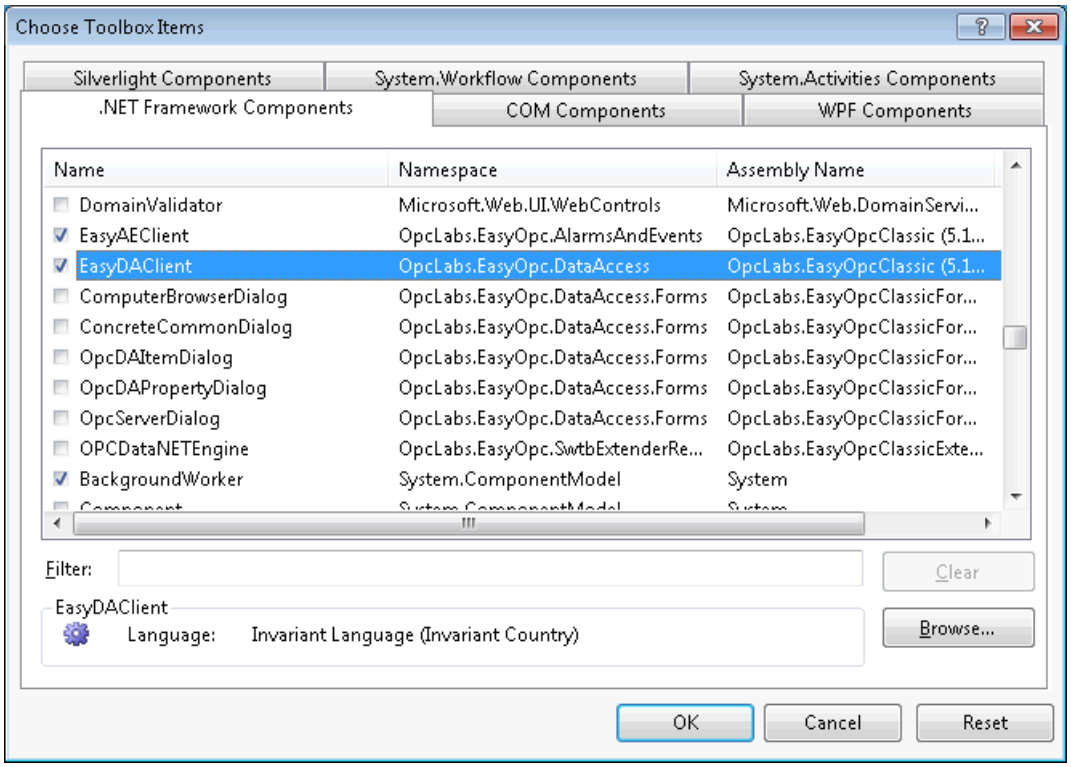
In order to be able to use the **EasyDAClient** object, you need to instantiate it first.

In QuickOPC.NET, there are two methods that you can use:

- Write the code that creates a new instance of the object, using am operator such as **New** (Visual Basic), **new** (Visual C#) or **gcnew** (Visual C++).
- Drag the component from the Toolbox to the designer surface. This works because **EasyDAClient** object is derived from

System.ComponentModel.Component (but is not limited to be used as a "full" component only). You can use this approach in Windows Forms applications, and in Console and some other types of applications if you first add a designable component to your application. The designer creates code that instantiates the component, assigns a reference to it to a field in the parent component, and sets its properties as necessary. You can then use designer's features such as the Properties grid to manipulate the component.

To add the **EasyDAClient** component to the Toolbox (you only need to this once): Right-click on the Toolbox, and select "Choose Items…".

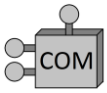


In the "Choose Toolbox Items" dialog, select ".NET Framework Components" tab, and sort the component by their namespace by clicking on the "Namespace" column header.

Then, look for "**EasyDAClient**" in the Name column (this is if you plan to use OPC Data Access). Check the box next to it, and press OK. The **EasyDAClient** item should appear in the Toolbox (note: it will only be visible in the proper context, i.e. when you have selected an appropriate parent component in the designer, or when you check "Show All" in the Toolbox context menu).

For OPC Alarms and Events, check "**EasyAEClient**" component.

For Windows Forms user interface components, check the components that you want under the "**OpcLabs.EasyOpc.DataAccess.Forms**" namespace.



In QuickOPC-COM, the precise way to instantiate the object depends on the programming language you are using. For example:

- In Visual Basic 6.0 and Visual Basic for Applications, if you have referenced the component(s), use the New keyword with the appropriate class name.
- In VBScript (or in VB if you have not referenced the component), use the CreateObject function with the ProgID of the class.
- In JScript, use the 'new ActiveXObject(…)' construct, with the ProgID of the class.
- In C++, create the smart interface pointer passing '__uuidof(…)' to the constructor, using the CLSID of the class.
- In Delphi (Object Pascal), call .Create on the class type create by importing the type library.
- In PHP, use the 'new COM(…)' construct, with the ProgID of the class.
- In Python, use 'win32com.client.Dispatch(…)' construct, with the ProgID of the class.
- In Visual FoxPro, use the CREATEOBJECT function with the ProgID of the class.

Following table contains information that is needed by different languages to instantiate the object(s):

Class Name	CLSID	ProgID	Version Independent ProgID
EasyDAClient	F1B8E6D7-955F-4C12-A015-9EF6282F73CC	OPCLabs.EasyDAClient.5.1	OPCLabs.EasyDAClient
EasyAEClient	ED35FC37-84EE-47BD-ADBA-BAA195B9B211	OPCLabs.EasyAEClient.5.1	OPCLabs.EasyAEClient

Default Instance

Instead of explicitly instantiating the **EasyDAClient** (or **EasyAEClient**) objects, you can also use a single, pre-made instance of it, resulting in shorter code. You can access it as a **DefaultInstance** static property on **EasyDAClient** (or **EasyAEClient**), and it contains a default, shared instance of the client object.

Use this property with care, as its usability is limited. Its main use is for testing, and for non-library application code where just a single instance is sufficient.

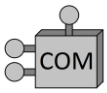
The default instance is not suitable for Windows Forms or similar environments, where a specific **SynchronizationContext** may be used with each form.

We also do not recommend using the default instance for library code, due to conflicts that may arise if your library sets some instance parameters which may not be the same as what other libraries or the final application expects.

User Interface Objects

In QuickOPC.NET, you instantiate a user interface object in Windows Forms applications by dragging the appropriate component from the Toolbox to the designer surface. The designer creates code that instantiates the component, assigns a reference to it to a field in the parent component, and sets its properties as necessary. You can then use designer's features such as the Properties grid to manipulate the component.

To add the user interface components to the Toolbox (you only need to this once): Right-click on the Toolbox, and select "Choose Items…". In the "Choose Toolbox Items" dialog, select ".NET Framework Components" tab, and look for "**OpcDAItemDialog**", " **OpcDAPropertyDialog**", and "**OpcServerDialog**" in the Name column. Check the boxes next to them, and press OK. The **OpcDAItemDialog**, **OpcDAPropertyDialog**, and **OpcServerDialog** items should appear in the Toolbox (note: they will only be visible in the proper context, i.e. when you have selected an appropriate parent component in the designer, or when you check "Show All" in the Toolbox context menu).



In QuickOPC-COM, you instantiate a user interface object in the same way as computational object (described above). Following table contains information needed to do so.

Class Name	CLSID	ProgID	Version Independent ProgID
OPCUserBrowseMachine	E13AC35E-CF9A-466a-A939-D02964A2AEF3	OPCLabs.UserBrowseMachine	
OPCUserBrowseServer	122D4D0E-BC48-45c2-B5CB-51D9D703F364	OPCLabs.UserBrowseServer	
OPCUserBrowseItem	19BB7459-5D7F-4d63-A119-A2803C1B2568	OPCLabs.UserBrowseItem	
OPCUserSelectItem	4E619C32-AF14-4d69-8056-02B4BE13A47F	OPCLabs.UserSelectItem	

Stateless Approach

OPC is inherently stateful. For starters, connections to OPC servers are long-living entities with rich internal state, and other objects in OPC model such as OPC groups have internal state too. QuickOPC hides most of the OPC's stateful nature by providing a stateless interface for OPC tasks.

This transformation from a stateful to a stateless model is actually one of the biggest advantages you gain by incorporating QuickOPC. There are several advantages to a stateless model from the perspective of your application. Here are the most important of them:

- The code you have to write is shorter. You do not have to make multiple method calls to get to the desired state first. Most common tasks can be achieved simply by instantiating an object (needed just once), and making a single method call.
- You do not have to worry about reconstructing all the state after some failure. QuickOPC reconstructs the OPC state silently in background when needed. This again brings tremendous savings in coding.

The internal state of QuickOPC components (including e.g. the connections to OPC servers) outlives the lifetime of the individual instances of the main **EasyDAClient** or **EasyAEClient** object. You can therefore create an instance of this object as many times as you wish, without incurring performance penalty to OPC communications. This characteristic comes extremely handy in programming server-side Web applications and similar scenarios: You can implement your code on page level, and make OPC requests from the page code itself. The OPC connections will outlive the page round-trips (if this was not the case, the OPC server would become bogged down quickly).

Simultaneous Operations

OPC works with potentially large quantities of relatively small data items that may change rapidly in time. In order to handle this kind of load effectively, it is necessary to operate on larger "chunks" of data whenever possible. When there is an operation to be performed on multiple elements, the elements should be passed to the operation together, and the results obtained together as well.

In order to ensure high efficiency, your code should allow the same. This is achieved by calling methods that are designed to work on multiple items in parallel. Where it makes sense, QuickOPC provides such methods, and they contain the work **Multiple** in their names. For example, for reading a value of an OPC item, a method named

ReadItemValue exists on the **EasyDAClient** object. There is also a corresponding method named **ReadMultipleItemValues** which can read multiple OPC items at once. It is strongly recommended that you call the methods that are designed for simultaneous operation wherever possible.

Methods for simultaneous operation return an array of **OperationResult** objects, or its derivatives. Each element in the output array corresponds to an element in the input array with the same index. Some methods or method overloads take multiple arguments, where some arguments are common for all elements, and one of them is the input array that has parts that are different for each element. There is always one method overload that takes a single argument which is an array of **OperationArguments** objects; this is the most generic method overload that allows each element be fully different from other elements.

Error Handling

Various kinds of errors may be returned by QuickOPC, e.g.:

- Errors returned by system calls when performing OPC-related operations.
- Errors returned by COM/DCOM infrastructure, including RPC and network-related errors.
- Errors reported by the OPC server you are connecting to.
- Errors detected by the QuickOPC library.

In general, you cannot safely prevent these errors from happening. Many conditions external to your code can cause OPC failures, e.g. network problems, improper OPC server configuration, etc. For this reason, you should always expect than OPC operation can fail.

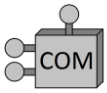
QuickOPC.NET defines one new type of exception, called **OpcException**, derived from **Exception** object. This exception is for all errors arising from OPC operations.

More details about the cause of the problem can be found by interrogating the **InnerException** property of **OpcException**, or by examining the **ErrorCode** property. In most scenarios, however, your code will be handling all **OpcException**-s in the same way.

If you need to display a meaningful error message to the user, or log it for further diagnostics, it is best to take the **OpcException** instance, obtain its base exception using **GetBaseException** method, and retrieve its **Message** property. The error message obtained in this way is closest to the actual cause of the problem.

QuickOPC.NET even tries to fill in the error text in cases when the system or OPC server has not provided it.

It should be noted that for QuickOPC.NET operations, **OpcException** is the ONLY exception class that your code should be explicitly catching when calling QuickOPC methods or getting properties. This is because you cannot safely influence or predict whether the exception will or will not be thrown. Other kinds of exception, such as those related to argument checking, should NOT be caught by typical application code, because they either indicate a programming bug in your code, or an irrecoverable system problem.



In QuickOPC-COM, the actual error handling concepts (and related terminology) depend strongly on the programming language and development tool you are using, for example:

- In C++, if you are using the raw interfaces provided by the type library, each function call will return an HRESULT value that you will test using macros such as SUCCEEDED() or FAILED().
- In C++, if you are using "Compiler COM Support" (the #import directive), errors will be converted to exceptions of **_com_error** type.
- In VBScript, failed function calls will either generate run-time error (with On Error Goto 0), or fill in the Err object with information about the error (with On Error Resume Next).

Errors and Multiple-Element Operations

Some methods on the main **EasyDAClient** object operate on multiple elements (such as OPC items) at once, and they also return results individually for each of the input elements. Such methods cannot simply throw an exception when there is a problem with processing one of the elements, because throwing an exception would make it impossible to further process other elements that are not causing errors. In addition, exception handling is very slow, and we need some efficient mechanism for dealing with situations when there may be multiple errors occurring at once.

For this reason, methods that work on multiple elements return an array of results, and each result may have an **Exception** associated with it. If this exception is a null reference, then there has been no error processing the operation on the element, and other data contained in the result object is valid. When the exception is not a null reference, it contains the actual error.

For multiple-element operations, the element-level exceptions are not wrapped in **OpcException**, because there is no need for you to distinguish them from other

exception types in the catch statement. If there is an exception inside a multiple-level operation, it is always an exception related to OPC operations. The Exception property of the result object in a multiple-element operation therefore contains what would be the **InnerException** of the **OpcException** thrown for a single-element operation.

Exceptions that are not related to OPC operations, such as argument-checking exceptions, are still thrown directly from the multiple-element operations, i.e. they are not returned in the **OperationResult** object.

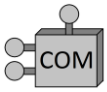
Errors in Subscriptions

Similarly as with multiple-element operations (above), errors in subscriptions are reported to your code by means of an **Exception** property in the event arguments passed to your event handler or callback method. If this exception is a null reference, then there has been no error related to the event notification, and other data contained in the event arguments object is valid. When the exception is not a null reference, it contains the actual error.

In event notifications, the exceptions are not wrapped in **OpcException**, because there is no need for you to distinguish them from other exception types in the catch statement. If there is an exception related to the event notification, it is always an exception related to OPC operations.

Helper Types

The types described here do not directly perform any OPC operations, but are commonly used throughout QuickOPC for properties and method arguments.



Dictionary Object

An associative array is used at several places in QuickOPC-COM interface, providing a way to store items associated with unique keys. The associative array is represented using a Dictionary object provide by Microsoft in their Scripting Runtime Library. For details, see http://msdn.microsoft.com/en-us/library/x4k5wbx4(VS.85).aspx.

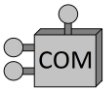
Time Periods

Many method arguments and properties describe time periods, such as update rates, delays, timeouts etc. For consistency, they are all integers, and they are expressed as number of milliseconds.

Some method arguments and properties (but only some – see Reference documentation for each method argument or property) allow a special value that represents an "infinite" time period.

In QuickOPC.NET, the value for "infinite" time period is equal to **Timeout**.**Infinite** (from **System**.**Threading** namespace).

Note: Time periods should not be confused with absolute time information, which is usually expressed by means of **DateTime** structure.



In QuickOPC-COM, the value for "infinite" time period is equal to **-1**.

Note: Time periods should not be confused with absolute time information, which is usually expressed by means of Windows **DATE** data type.

OPC Quality

OPC represents a quality of a data value by several bit-coded fields. QuickOPC.NET encapsulates the OPC quality in a **DAQuality** class. The bit fields in OPC quality are inter-dependent, making it a bit complicated to encode or decode it. The **DAQuality** type takes care of this complexity. In addition, it offers symbolic constants that represent the individual coded options, and also has additional functionality such as for converting the quality to a string.

The following table attempts to depict the elements of **DAQuality** and their relations:

<table>
<tr><th colspan="3">DAQuality</th></tr>
<tr><td>property QualityChoice QualityChoiceBitField
{ get; }
IsBad()
IsGood()
IsUncertain()</td><td>SubStatus</td><td rowspan="2">property DALimitChoice LimitBitField
{ get; set; }</td></tr>
<tr><td colspan="2">property DAStatusChoice StatusBitField
{ get; set; }
SetQualityAndSubStatus(…)</td></tr>
</table>

You can see that the **StatusBitField** is actually consisted of **QualityChoiceBitField**, and a SubStatus. But the semantics of SubStatus is highly dependent on **QualityChoiceBitField**, and therefore the SubStatus cannot be accessed separately. For the same reason, you cannot directly set the **QualityChoiceBitField** without providing a value for SubStatus at the same time. Instead, you can call **SetQualityAndSubstatus** method to modify the two fields at the same time.

Note that OPC Alarms and Events "borrows" the quality type from OPC Data Access, and therefore the **DAQuality** structure is used with OPC Alarms and Events as well.

Value, Timestamp and Quality (VTQ)

The combination of data value, timestamp and quality (abbreviated sometimes as VTQ) is common in OPC. This combination is returned e.g. when an OPC item is read. Note that according to OPC specifications, the actual data value is only valid when the quality is Good or Uncertain (with certain exception).

QuickOPC has a **DAVtq** object for this combination. The object provides access to individual elements of the VTQ combination, and also allows common operations such as comparisons. It can also be easily converted to a string containing textual representation of all its elements.

If you only want a textual representation for a data value from the VTQ, use the **DisplayValue** method. This is recommended over trying to extract the **Value** property and converting it to string, as you will automatically receive an empty string if the value is not valid according to OPC rules (e.g. Bad quality), and a null reference case is handled as well.

Result Objects

Result objects are returned by methods that work on multiple elements simultaneously such as **EasyDAClient**.**ReadMultipleItems**. Such methods return an array of **OperationResult** objects, or an array of objects derived from **OperationResult**. This approach is chosen, among other reasons, because the method cannot throw an exception if an operation on a single element fails.

Each **OperationResult** has an **Exception** property, which indicates the outcome of the operation. If the **Exception** is a null reference, the operation has completed successfully. There is also a **State** property, which contains user-defined information that you have passed to the method.

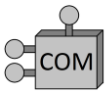
The objects derived from **OperationResult** have additional properties, which contain the actual results in case the operation was successful. Such objects are e.g. **ValueResult** (contains a data value), or **DAVtqResult** (contains value, timestamp, and quality combination).

Variant Type (VarType)

In some places in OPC, your code needs to indicate which type of data you expect to receive back, or (in the opposite direction), you receive indication about which type of data certain piece of information is. OPC uses Windows VARTYPE for this (describes a data contained in Windows VARIANT).

QuickOPC.NET gives you a .NET encapsulation for indicating variant data types, so that you do not have to look up and code in the numeric values of Windows VARTYPE. Instead, wherever you see that a method argument, a property, or other element is of **VarType** type, you can supply one of the constants defined in the **VarType**. For example, **VarType**.**I2** denotes a 16-bit signed integer, **VarType**.**R4** denotes a 32-bit float, and **VarType**.**BStr** denotes a string. For arrays of values, use logical 'or' to combine the element type with **VarType**.**Array** constant.

Note: Microsoft.NET framework contains a similar type, **System**.**Runtime**.**InteropServices**.**VarEnum**. The types have some similarities, but should not be confused.



QuickOPC-COM gives you an enumeration for indicating variant data types, so that you do not have to look up and code in the numeric values of Windows VARTYPE. Instead, wherever you see that a method argument, a property, or other element accepts a data type, you can supply one of the constants defined in the **VarType** enumeration. For example, **VTI2** denotes a 16-bit signed integer, **VTR4** denotes a 32-bit float, and **VTBStr** denotes a string. For arrays of values, use logical 'or' to combine the element type with **VTArray** constant. Note, however, that the enumeration symbols are only accessible if you reference or import the EasyOPC Type Library, and not all languages and tools are capable of doing it.

Element Objects

Element objects contain all information gathered about some OPC entity. They are typically returned by browsing methods. There are following types of element objects:

- **ServerElement** object contains information gathered about an OPC server.
- **DANodeElement** object contains information gathered about an OPC node (branch or leaf in OPC Data Access server's address space).
- **DAPropertyElement** contains information gathered about an OPC Data Access property.
- **AEAttributeElement** contains information gathered about an OPC Alarms and Events attribute.
- **AECategoryElement** contains information gathered about an OPC Alarms and Events category.
- **AEConditionElement** contains information gathered about an OPC Alarms and Events condition.
- **AENodeElement** object contains information gathered about an OPC node (areas or source in OPC Alarms and Events server's address space).

- **AESubconditionElement** contains information gathered about an OPC Alarms and Events subcondition.

Element objects are also returned when you invoke one of the common OPC dialogs for selecting OPC server, OPC-DA item or an OPC-DA property.

Descriptor Objects

A *descriptor object* contains information that fully specifies certain OPC entity (but does not contain any "extra" information that is not needed to identify it uniquely). Descriptor objects are used by some method overloads to reduce the number of individual arguments, and to organize them logically. There are following types of descriptor objects:

- **ServerDescriptor** contains information necessary to identify and connect to an OPC server, such as the server's ProgID.
- **DAItemDescriptor** contains information necessary to identify an OPC item, such as its Item Id.

If you have received an element object (e.g. from browsing methods or common OPC dialogs), you can convert it to a descriptor object. For example, there is a constructor for **ServerDescriptor** object that accepts **ServerElement** as an input, and there is a constructor for **DAItemDescriptor** object that accepts **DANodeElement** as an input, too.

Parameter Objects

Parameter objects are just holders for settings that influence certain aspect of how QuickOPC works (for example, timeouts). There are several types of parameter objects (such as **Timeouts**, **HoldPeriods**, and more). For more information, see the "Setting Parameters" and "Advanced Topics" chapters, and also the Reference documentation.

Note that there is no way for your code to create new instances of parameter objects (and assign them to properties of main **EasyDAClient** or **EasyAEClient** object). Your code simply manipulates properties of parameter objects that are already created and made available to you.

OPC Data Access Tasks

This chapter gives you guidance in how to implement the common tasks that are needed when dealing with OPC Data Access server from the client side. You achieve these tasks by calling methods on the **EasyDAClient** object.

Obtaining Information

Methods described in this chapter allow your application to obtain information from the underlying data source that the OPC server connect to (reading OPC items), or from the OPC server itself (getting OPC property values). It is assumed that your application already somehow knows how to identify the data it is interested in. If the location of the data is not known upfront, use methods described in Browsing for Information chapter first.

Reading from OPC Items

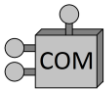
In OPC Data Access, reading data from OPC items is one of the most common tasks. The OPC server generally provides current data for any OPC item in form of a Value, Timestamp and Quality combination (VTQ).

If you want to read the current VTQ from a specific OPC item, call the **ReadItem** method. You pass in individual arguments for machine name, server class, ItemID, and an optional data type. You will receive back a **DAVtq** object holding the current value, timestamp, and quality of the OPC item. The **ReadItem** method returns the current VTQ, regardless of the quality. You may receive an Uncertain or even Bad quality (and no usable data value), and your code needs to deal with such situations accordingly.

In QuickOPC.NET, you can also pass **ServerDescriptor** and **DAItemDescriptor** objects in place of individual arguments to the **ReadItem** method.

For reading VTQs of multiple items simultaneously in an efficient manner, call the **ReadMultipleItems** method (instead of multiple **ReadItem** calls in a loop). You will receive back an array of **DAVtqResult** objects.

In QuickOPC.NET, you can pass in a **ServerDescriptor** object and an array of **DAItemDescriptor** objects, or an array of **DAItemArguments** objects, to the **ReadMultipleItems** method.

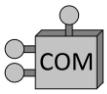


In QuickOPC-COM, in place of each **ReadItem** argument, you can pass in an array of values, or (if the value is the same for the whole operation, e.g. the machine name), a single value as with the **ReadItem** call, to the **ReadMultipleItems** method.

Some applications need the actual data value for further processing (e.g. for computations that need be performed on the values), even if it involves waiting a little for the quality to become Good. For such usage, call the **ReadItemValue** method, passing it the same arguments as to the **ReadItem** method. The method will wait until the OPC item's quality becomes Good (or until a timeout expires), and you will receive back an **Object** (a VARIANT in QuickOPC-COM) holding the actual data value.

For reading just the data values of multiple data values (with wait for Good quality) in an efficient manner, call the **ReadMultipleItemValues** method (instead of multiple **ReadItemValue** calls in a loop). You will receive back an array of **ValueResult** objects.

In QuickOPC.NET, you can pass in a **ServerDescriptor** object and an array of **DAItemDescriptor** objects, or an array of **DAItemArguments** objects, to the **ReadMultipleItemValues** method.



In QuickOPC-COM, in place of each **ReadItemValue** argument, you can pass in an array of values, or (if the value is the same for the whole operation, e.g. the machine name), a single value as with the **ReadItemValue** call, to the **ReadMultipleItemValues** method.

Note: You can set the **DesiredValueAge** in **ClientMode** property to control how "old" may be the values you receive by reading from OPC items. Be aware that it is physically impossible for any system to always obtain fully up-to-date values.

Getting OPC Property Values

Each OPC item has typically associated a set of OPC properties with it. OPC properties contain additional information related to the item.

If you want to obtain the value of specific OPC property, call the **GetPropertyValue** method, passing it the machine name, server class**,** the ItemId, and a PropertyId. You will receive back an **Object** (a VARIANT in QuickOPC-COM) containing the value of the requested property.

In QuickOPC.NET, you can also pass in the **ServerDescriptor** inplace of the machine name and server class strings.

There are many property-related methods in EasyOPC.NET Extensions component. Please refer to a chapter in this document that describes EasyOPC.NET Extensions.

For obtaining multiple properties simultaneously in an efficient manner, call the **GetMultiplePropertyValues** method (instead of multiple **GetPropertyValue** calls in a loop). The arguments are similar, except that in place of a single PropertyId you pass in an array of them. You will receive back an array of **Object** values (a SAFEARRAY of VARIANT values in QuickOPC-COM).

Modifying Information

Methods described in this chapter allow your application to modify information in the underlying data source that the OPC server connects to (writing OPC items). It is assumed that your application already somehow knows how to identify the data it is interested in. If the location of the data is not known upfront, use methods described Browsing for Information chapter first.

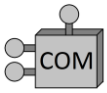
Writing to OPC Items

If you want to write a data value into a specific OPC item, call the **WriteItemValue** method, passing it the data value you want to write, arguments for machine name, server class, ItemID, and an optional data type.

In QuickOPC.NET, you can also pass in the **ServerDescriptor** and **DAItemDescriptor** objects in place of corresponding individual arguments.

For writing data values into multiple OPC items in an efficient manner, call the **WriteMultipleItemValues** method.

In QuickOPC.NET, you pass in an array of **DAItemValueArgument** objects, each specifying the location of OPC item, and the value to be written.



In QuickOPC-COM, in place of each **WriteItemValue** argument, you can pass in an array of values, or (if the value is the same for the whole operation, e.g. the machine name), a single value as with the **WriteItemValue** call.

Some newer OPC servers allow a combination of value, timestamp, and quality (VTQ) be written into their items. If you need to do this, call **WriteItem** or **WriteMultipleItems** method.

Browsing for Information

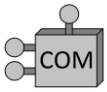
QuickOPC contains methods that allow your application to retrieve and enumerate information about OPC servers that exist on the network, and data available within these servers. Your code can then make use of the information obtained, e.g. to accommodate to configuration changes dynamically.

Note that if you just want to allow your user to browse interactively for various OPC elements, you can simply code your application to invoke the common dialogs that are already implemented in QuickOPC (they are described further down in this document). The methods we are describing here are for programmatic browsing, with no user interface (or when you provide the user interface by your own code).

Browsing for OPC Servers

If you want to retrieve a list of OPC Data Access servers registered on a local or remote computer, call the **BrowseServers** method, passing it the name or address of the remote machine (use empty string for local computer).

In QuickOPC.NET, you will receive back a **ServerElementCollection** object. If you want to connect to this OPC server later in your code by calling other methods, use the built-in conversion of **ServerElement** to **String**, and pass the resulting string as a **serverClass** argument either directly to the method call, or to a constructor of **ServerDescriptor** object.



In QuickOPC-COM, you will receive back a **Dictionary** of **ServerElement** objects. If you want to connect to this OPC server later in your code by calling other methods, obtain the value of **ServerElement.ServerClass** property, and pass the resulting string as a **serverClass** argument to the method call that accepts it.

Each **ServerElement** contains information gathered about one OPC server found on the specified machine, including things like the server's CLSID, ProgID, vendor name, and readable description.

Browsing for OPC Nodes (Branches and Leaves)

Items in an OPC server are typically organized in a tree hierarchy (address space), where the branch nodes serve organizational purposes (similar to folders in a file system), while the leaf nodes correspond to actual pieces of data that can be accessed (similar to files in a file system) – the OPC items. Each node has a "short" name that is unique among other branches or leaves under the same parent branch (or a root). Leaf nodes can be fully identified using a "long" ItemID, which determines the OPC item without a need to further qualify it with its position in the tree. ItemIDs may look like "Device1.Block101.Setpoint", however their syntax and meaning is fully determined by the particular OPC server they are coming from.

QuickOPC gives you methods to traverse through the address space information and obtain the information available there. It is also possible to filter the returned nodes by various criteria, such as node name matching certain pattern, or a particular data type only, or writeable items only, etc.

If you want to retrieve a list of all sub-branches under a given branch (or under a root) of the OPC server, call the **BrowseBranches** method. In QuickOPC.NET, you will receive back a **DANodeElementCollection** object. In QuickOPC-COM, you will receive back a **Dictionary** of **DANodeElement** objects. Each **DANodeElement** contains information gathered about one sub-branch node, such as its name, or indication whether it has children. Similarly, if you want to retrieve a list of leaves under a given branch (or under a root) of the OPC server, call the **BrowseLeaves** method. You will also receive back a **DANodeElementCollection** object (in QuickOPC.NET) or a **Dictionary** of **DANodeElement** objects(in QuickOPC-COM), this time containing the leaves only. You can find information such as the Item ID from the **DANodeElement** of any leaf, and pass it further to methods like **ReadItem** or **SubscribeItem**.

The most generic address space browsing method is **BrowseNodes**. It combines the functionality of **BrowseBranches** and **BrowseLeaves**, and it also allows the widest range of filtering options by passing in an argument of type **DANodeFilter** (in QuickOPC.NET), or individual arguments for data type filter and access rights filter (in QuickOPC-COM).

Browsing for OPC Access Paths

Access paths are somewhat obsolete feature of OPC Data Access specification, and few OPC server actually use it; but if a particular OPC server does use access paths, specifying the proper access path together with ItemID may be the only way to retrieve the data you want.

If you want to retrieve a list of possible access paths available for a specific OPC item, call the **BrowseAccessPaths** method, passing it the information about the OPC server, and the ItemID. You will receive back an array of strings; each element of this array is an access path that you can use with methods such as **ReadItem** or **SubscribeItem**.

In QuickOPC.NET, you can also pass the access path to a constructor of **DAItemDescriptor** object and later use that descriptor with various methods.

Browsing for OPC Properties

Each OPC item has typically associated a set of OPC properties with it. OPC properties contain additional information related to the item. The OPC specifications define a set of common properties; however, each OPC server is free to implement some more, vendor-specific properties as well.

If you want to retrieve a list of all properties available on a given OPC item, call the **BrowseProperties** method, passing it the ItemID you are interested in. In QUickOPC.NET, you will receive back a **DAPropertyElementCollection** object. In

QuickOPC-COM, you will receive back a **Dictionary** of **DAPropertyElement** objects. Each **DAPropertyElement** contains information about one OPC property, such as its (numeric) PropertyId, data type, or a readable description. The PropertyId can be later used as an argument in calling methods such as **GetPropertyValue**.

Subscribing for Information

If your application needs to monitor changes of certain process value (OPC item), it can subscribe to it, and receive notifications when the value changes. For performance reasons, this approach is preferred over repeatedly reading the item's value (polling). Note that QuickOPC has internal optimizations which greatly reduce the negative effects of polling, however subscription is still preferred.

QuickOPC contains methods that allow you to subscribe to OPC items, change the subscription parameters, and unsubscribe.

Subscribing to OPC Items

Subscription is initiated by calling either **SubscribeItem** or **SubscribeMultipleItems** method. For any change in the subscribed item's value, your application will receive the **ItemChanged** event notification, described further below. Obviously, you first need to hook up event handler for that event, and in order to prevent event loss, you should do it before subscribing. Alternatively, you can pass a callback method into the **SubscribeItem** or **SubscribeMultipleItems** call.

Values of some items may be changing quite frequently, and receiving all changes that are generated is not desirable for performance reasons; there are also physical limitations to the event throughput in the system. Your application needs to specify the *requested update rate*, which effectively tells the OPC server that you do not need to receive event notifications any faster than that. For OPC items that support it, you can optionally specify a *percent deadband*; only changes that exceed the deadband will generate an event notification.

In QuickOPC.NET, the requested update rate, percent deadband, and data type are all contained in a **DAGroupParameters** object.

If you want to subscribe to a specific OPC item, call the **SubscribeItem** method. You can pass in individual arguments for machine name, server class, ItemID, data type, requested update rate, and an optional percent deadband. Usually, you also pass in a **State** argument of type **Object** (in QuickOPC.NET) or VARIANT (in QuickOPC-COM). When the item's value changes, the **State** argument is then passed to the **ItemChanged** event handler in the **EasyDAItemChangedEventArgs** object. The

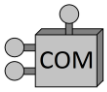
SubscribeItem method returns a subscription handle that you can later use to change the subscription parameters, or unsubscribe.

In QuickOPC.NET, you can also pass in a combination of **ServerDescriptor**, **DAItemDescriptor** and **DAGroupParameters** objects, in place of individual arguments.

The **State** argument is typically used to provide some sort of correlation between objects in your application, and the event notifications. For example, if you are programming an HMI application and you want the event handler to update the control that displays the item's value, you may want to set the **State** argument to the control object itself. When the event notification arrives, you simply update the control indicated by the **State** property of **EasyDAItemChangedEventArgs**, without having to look it up by ItemId or so.

To subscribe to multiple items simultaneously in an efficient manner, call the **SubscribeMultipleItems** method (instead of multiple **SubscribeItem** calls in a loop). You receive back an array of **HandleResult** objects (containing the subscription handles).

In QuickOPC.NET, you pass in an array of **DAItemGroupArguments** objects (each containing information for a single subscription to be made), to the **SubscribeMultipleItems** method.



In QuickOPC-COM, you pass in an array or arrays of arguments (each element containing information for a single subscription to be made, to the **SubscribeMultipleItems** method.

Note: It is NOT an error to subscribe to the same item twice (or more times), even with precisely same parameters. You will receive separate subscription handles, and with regard to your application, this situation will look no different from subscribing to different items. Internally, however, the subscription made to the OPC server will be optimized (merged together) if possible.

There is also an event called **MultipleItemsChanged** that can deliver multiple item changes in one call to the event handler. See Multiple Notifications in One Call in Advanced Topics for more.

Changing Existing Subscription

It is not necessary to unsubscribe and then subscribe again if you want to change parameters of existing subscription (such as its update rate). Instead, change the parameters by calling the **ChangeItemSubscription** method, passing it the

subscription handle, and the new parameters in form of **DAGroupParameters** object (in QuickOPC.NET) or individually a requested update rate, and optionally a percent deadband (in QuickOPC-COM).

For changing parameters of multiple subscriptions in an efficient manner, call the **ChangeMultipleItemSubscriptions** method.

Unsubscribing from OPC Items

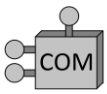
If you no longer want to receive item change notifications, you need to unsubscribe from them. To unsubscribe from a single OPC item, call the **UnsubscribeItem** method, passing it the subscription handle.

To unsubscribe from multiple OPC items in an efficient manner, call the **UnsubscribeMultipleItems** method (instead of calling **UnsubscribeItem** in a loop), passing it an array of subscription handles, or an array of **HandleArguments** objects.

You can also unsubscribe from all items you have previously subscribed to (on the same instance of **EasyDAClient** object) by calling the **UnsubscribeAllItems** method.

If you are no longer using the parent **EasyDAClient** object, you do not necessarily have to unsubscribe from the items, but it is highly recommended that you do so.

In QuickOPC.NET, the subscriptions will otherwise be internally alive until the .NET CLR (garbage collector) decides to finalize and destroy the parent **EasyDAClient** object (if ever); you cannot, however, determine that moment. You can alternatively call the **Dispose** method of the **EasyDAClient** object's **IDisposable** interface, which will unsubscribe from all items for you.



In QuickOPC-COM, the subscriptions will otherwise be internally alive. You can alternatively release all references to the **EasyDAClient** object, which will unsubscribe from all items for you.

Item Changed Event or Callback

When there is a change in a value of an OPC item you have subscribed to, the **EasyDAClient** object generates an **ItemChanged** event. For subscription mechanism to be useful, you should hook one or more event handlers to this event.

To be more precise, the **ItemChanged** event is actually generated in other cases, too.

First of all, you always receive at least one **ItemChanged** event notification after you make a subscription; this notification either contains the initial data for the item, or an indication that data is not currently available. This behavior allows your

application to rely on the component to provide at least some information for each subscribed item.

Secondly, the **ItemChanged** event is generated every time the component loses connection to the item, and when it reestablishes the connection. This way, your application is informed about any problems related to the item, and can process them accordingly if needed.

You will also receive the **ItemChanged** notification if the quality of the item changes (not just its actual data value).

The **ItemChanged** event notification contains an **EasyDAItemChangedEventArgs** argument. You will find all kind of relevant data in this object. Some properties in this object contain valid information no matter what kind of change the notification is about. These properties are **ServerDescriptor**, **ItemDescriptor**, **GroupParameters**, **Handle**, and **State**.

For further processing, your code should always inspect the value of **Exception** property of the event arguments. If this property is set to a null reference, the notification carries an actual change in item's data, and the **Vtq** property has the new value, timestamp and quality of the item, in form of **DAVtq** object. If the **Exception** property is not a null reference, there has been an error related to the item, and the **Vtq** property is not valid. In such case, the **Exception** property contains information about the problem.

The **ItemChanged** event handler is called on a thread determined by the **EasyDAClient** component. For details, please refer to "Multithreading and Synchronization" chapter under "Advanced Topics".

In short, however, we can say that if you are writing e.g. Windows Forms application, the component takes care of calling the event handler on the user interface thread of the form, making it safe for your code to update controls on the form, or do other form-related actions, from the event handler.

Using Callback Methods Instead of Event Handlers

Using event handlers for processing notifications is a standard way with many advantages. There also situations, however, where event handlers are not very practical. For example, if you want to do fundamentally different processing on different kinds of subscriptions, you end up with all notifications being processed by the same event handler, and you need to put in extra code to distinguish between different kinds of subscriptions they come from. Event handlers also require additional code to set up and tear down.

In order to overcome these problems, QuickOPC components allow you to pass in a delegate for a callback method to subscription methods. There are subscription methods overloads that accept the callback method parameter. The callback method has the same signature (arguments) as the event handler, and is called by the component in addition to invoking the event handler (if you do not hook a handler to the event, only the callback method will be invoked). You can therefore pass in the delegate for the callback method right into the subscription method call, without setting up event handlers.

The callback method can also be specified using an anonymous delegate or a lambda expression, i.e. without having to declare the method explicitly elsewhere in your code. This is especially useful for short callback methods.

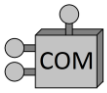
For subscription methods that work with multiple subscriptions at once, there is also a **Callback** property in the arguments objects that you can use for the same purpose.

Note that if you specify a non-null callback parameter to the subscription method, the callback method will be invoked in addition to the event handlers. If you use both event handlers and callback methods in the same application, and you do not want the event handlers to process the notifications that are also processed by the callback methods, you can either

- test the **Callback** property in the event arguments of the event handler, and if it is not a null reference, the event has been processed by the callback method and you can ignore it, or
- use two instances of the **EasyDAClient** (or **EasyAEClient**) object, and set up event handlers on one instance and use the callback methods on the other instance.

Setting Parameters

While the most information needed to perform OPC tasks is contained in arguments to method calls, there are some component-wide parameters that are not worth repeating in every method call, and also some that have wider effect that influences more than just a single method call. You can obtain and modify these parameters through properties on the **EasyDAClient** object.



In QuickOPC-COM, you can also modify these parameters by the EasyOPC Options Utility, available from the Start menu.

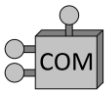
Following are instance properties, i.e. if you have created multiple **EasyDAClient** object, each will have its own copy of them:

- **ClientMode**: Specifies common parameters such as allowed and desired methods of accessing the data in the OPC server.
- **HoldPeriods**: Specifies optimization parameters that reduce the load on the OPC server.
- **UpdateRates**: Specifies the "hints" for OPC update rates used when other explicit information is missing.
- **Timeouts**: Specifies the maximum amount of time the various operations are allowed to take.

- **SynchronizationContext**: Contains synchronization context used by the object when performing event notifications.

Instance properties can be modified from your code.

In QuickOPC.NET, if you have placed the **EasyDAClient** object on the designer surface, most instance properties can also be directly edited in the Properties window in Visual Studio.



In QuickOPC-COM, the EasyOPC Options utility is used to set the default values of instance properties. Your code can override the defaults if needed.

Following properties are static, i.e. shared among all instances of **EasyDAClient** object:

- **EngineParameters**: Contains global parameters such as frequencies of internal tasks performed by the component.
- **MachineParameters**: Contains parameters related to operations that target a specific computer but not a specific OPC server, such as browsing for OPC servers using various methods.
- **ClientParameters**: Contains parameters that influence operations that target a specific OPC server a whole.
- **TopicParameters**: Contains parameters that influence operations that target a specific OPC item.

Static properties can only be modified from your code (in QuickOPC.NET) or using the EasyOPC Options utility (in QuickOPC-COM). If you want to modify any of the static properties, you must do it before the first instance of **EasyDAClient** or **EasyAEClient** object is created.

Please use the Reference documentation (and EasyOPC Options Help in QuickOPC-COM) for details on meaning of various properties and their use.

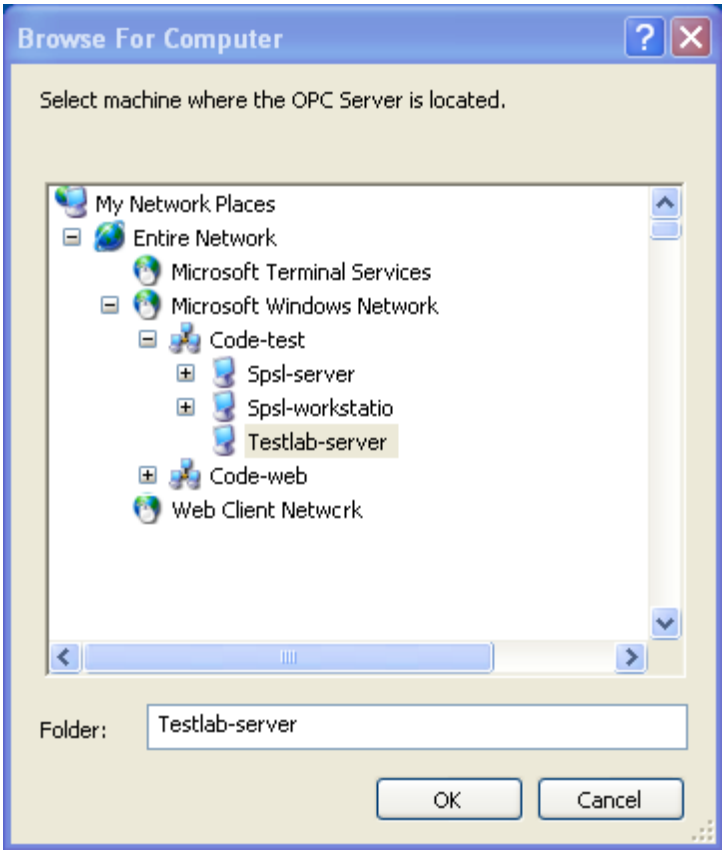
OPC Common Dialogs

QuickOPC.NET contains a set of Windows Forms dialog boxes for performing common OPC-related tasks such as selecting an OPC server or OPC item.

The dialog objects are all derived from **System.Windows.Forms.CommonDialog**, providing consistent and well-known programming interface to use.

Computer Browser Dialog

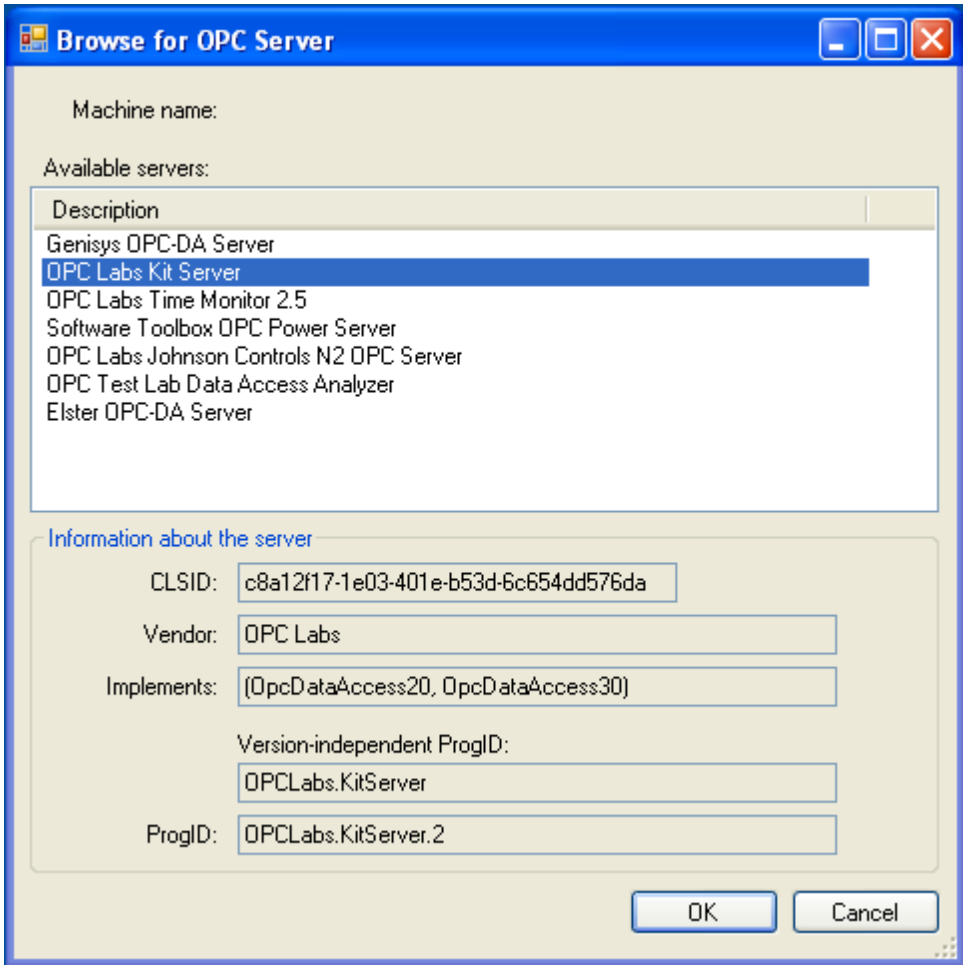
OPC servers are usually deployed on the network, and accessed via DCOM. In order to let the user select the remote computer where the OPC server resides, you can use the **ComputerBrowseDialog** object.



Call the **ShowDialog** method, and if the result is equal to **DialogResult.OK**, the user has selected the computer, and its name can be retrieved from the **SelectedName** property.

OPC Server Dialog

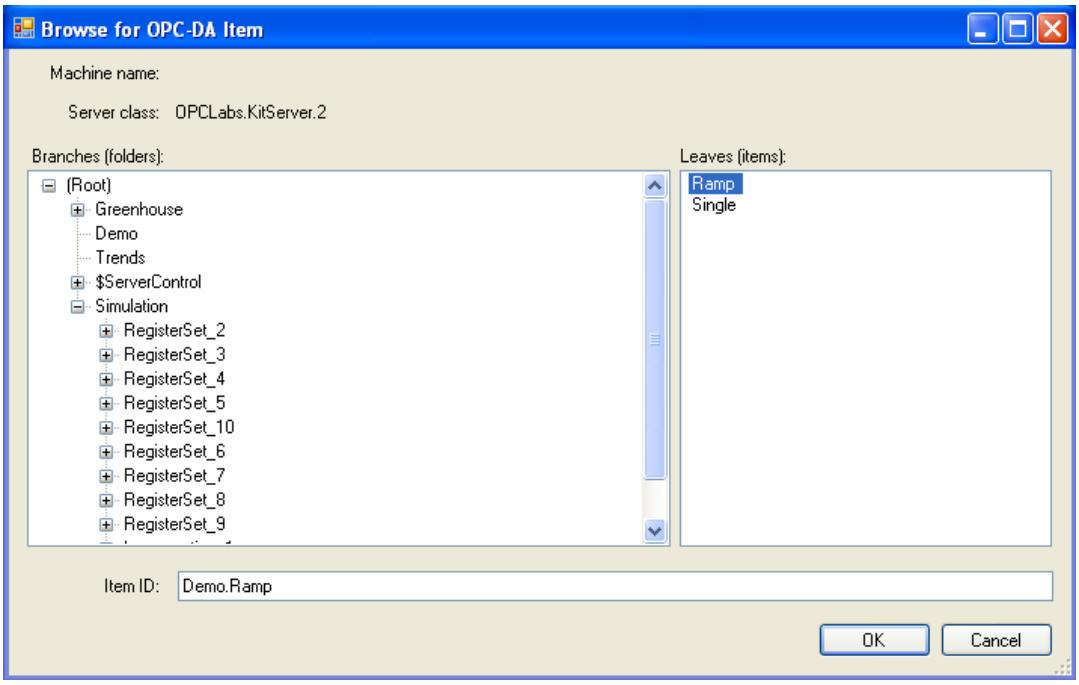
If you do not know upfront which OPC server to connect to, and do not have this information from any other source, your application will need to allow the user select the OPC server(s) to work with. The OPC Server Dialog allows the user to select the OPC server interactively from the list of OPC Data Access servers installed on a particular machine.



Set the **MachineName** property to the name of the computer that is to be browsed, and call the **ShowDialog** method. If the result is equal to **DialogResult.OK**, the user has selected the OPC Data Access server, and information about it can be retrieved from the **SelectedServer** property.

OPC-DA Item Dialog

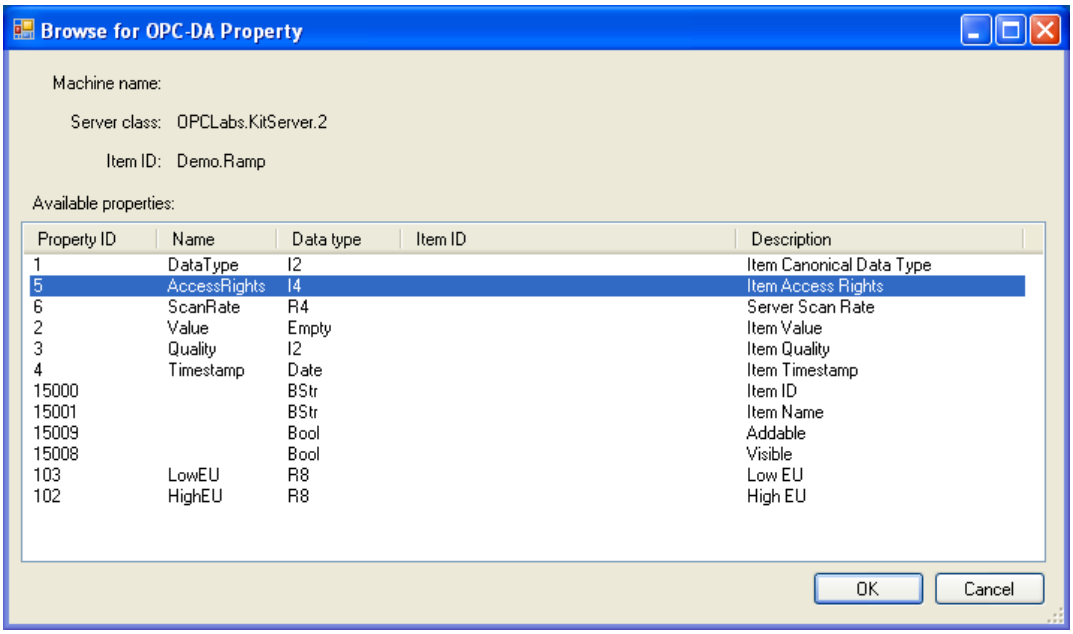
The OPC-DA Item Dialog allows the user to interactively select the OPC item residing in a specific OPC server.



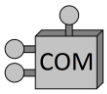
Use the **Server** property to specify the OPC Data Access server whose items are to be browsed, and call the **ShowDialog** method. If the result is equal to **DialogResult.OK**, the user has selected the OPC item, and information about it can be retrieved from the **SelectedNode** property.

OPC-DA Property Dialog

The OPC-DA Property Dialog allows the user to interactively select the OPC property on a specific OPC item.



Use the **Server** property to specify the OPC Data Access server whose items are to be browsed, set the **ItemId** property to the OPC Item Id needed, and then call the **ShowDialog** method. If the result is equal to **DialogResult.OK**, the user has selected the OPC property, and information about it can be retrieved from the **SelectedProperty** property.



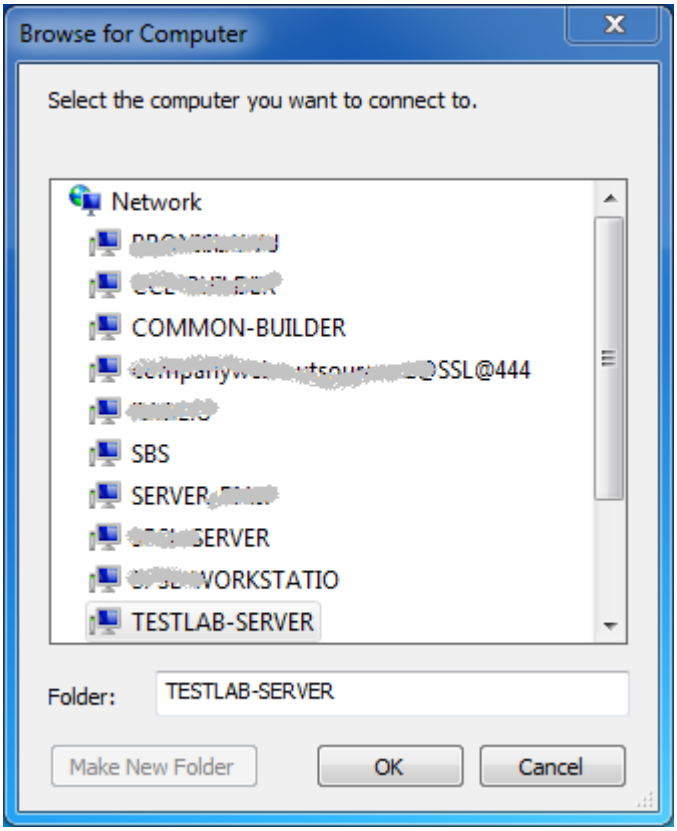
OPC User Objects

QuickOPC-COM contains a set of Windows dialog boxes for performing common OPC-related tasks such as selecting an OPC server or OPC item.

The programming interfaces for the dialog objects are all similar, providing consistent programming interface to use.

Computer Browser Dialog

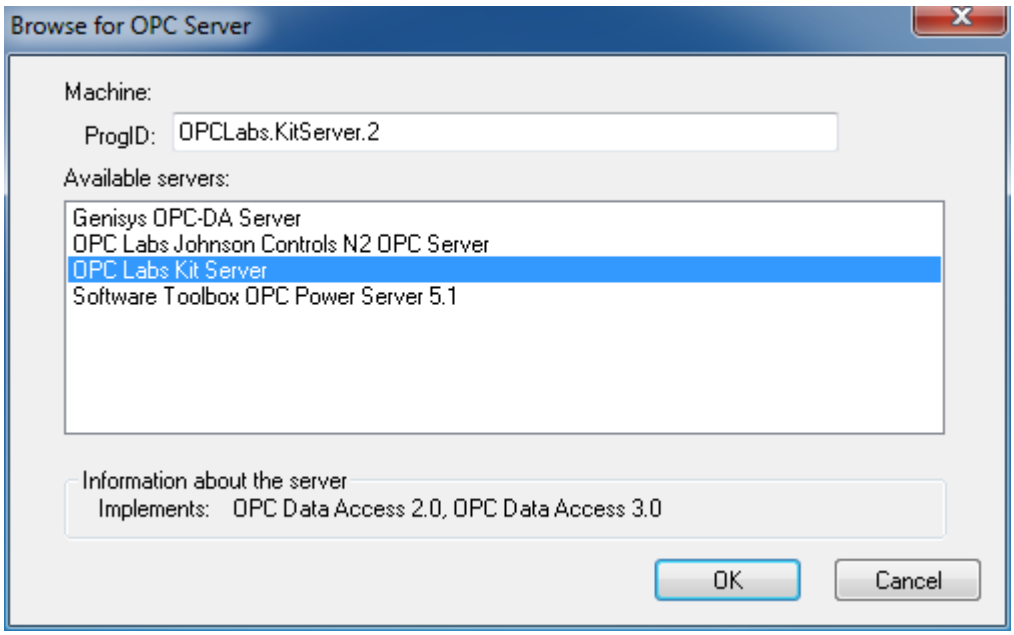
OPC servers are usually deployed on the network, and accessed via DCOM. In order to let the user select the remote computer where the OPC server resides, you can use the **OPCUserBrowseMachine** object.



Call the **RunMe** method, and if the result is equal to 1 (**DialogResult.OK**), the user has selected the computer, and its name can be retrieved from the **MachineName** property.

OPC Server Browse Dialog

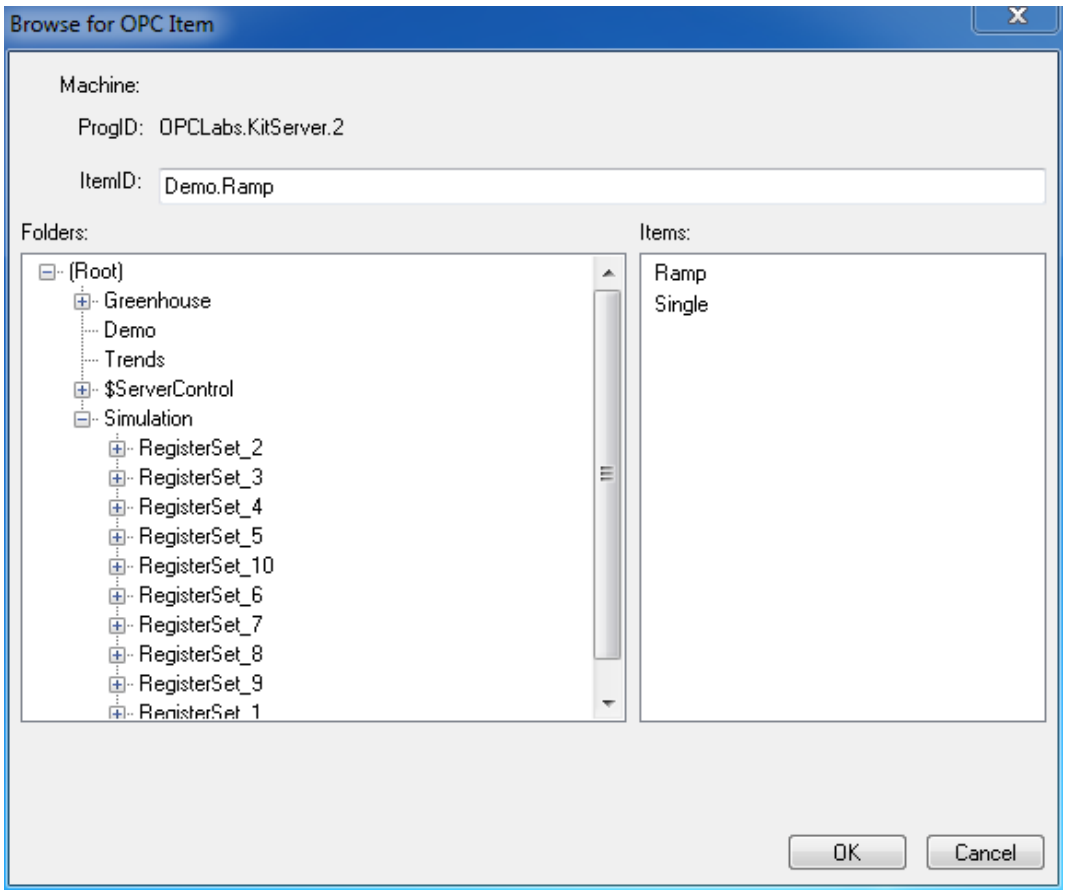
If you do not know upfront which OPC server to connect to, and do not have this information from any other source, your application will need to allow the user select the OPC server(s) to work with. The OPC Server Browse Dialog allows the user to select the OPC server interactively from the list of OPC Data Access servers installed on a particular machine.



Set the **MachineName** property to the name of the computer that is to be browsed, and call the **RunMe** method. If the result is equal to 1 (**DialogResult.OK**), the user has selected the OPC Data Access server, and information about it can be retrieved from the **ServerClass** property.

OPC-DA Item Browse Dialog

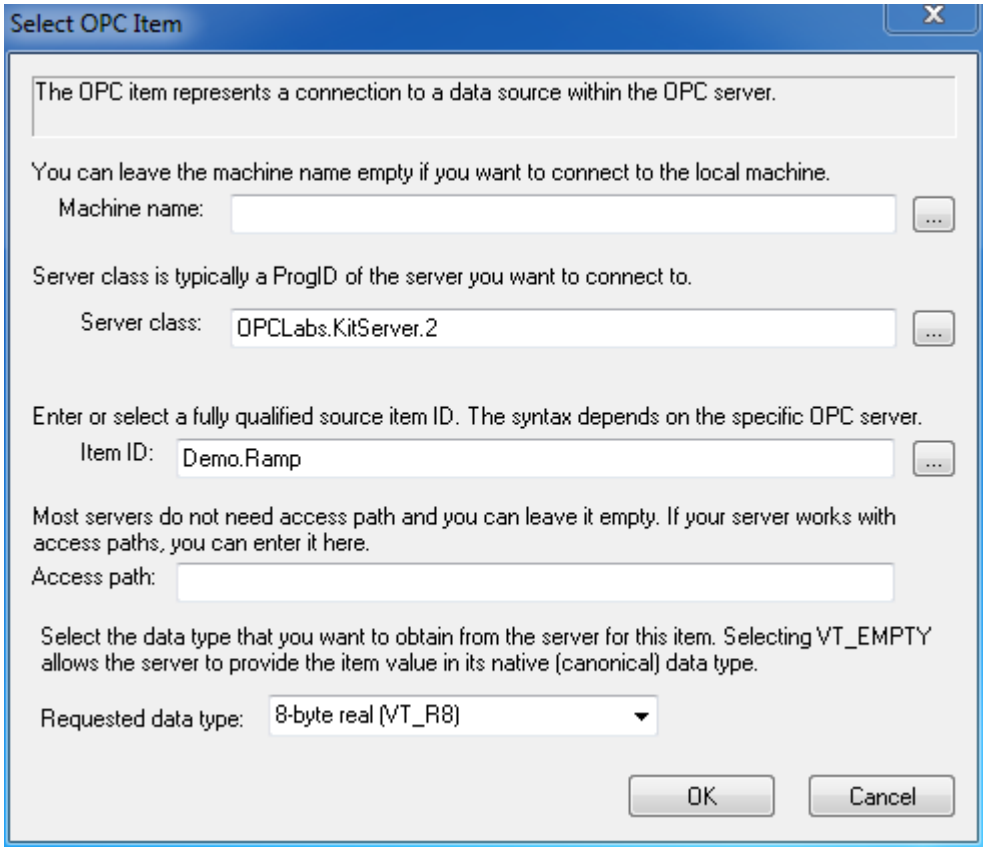
The OPC-DA Item Browse Dialog allows the user to interactively select the OPC item residing in a specific OPC server.



Use the **MachineName** and **ServerClass** properties to specify the OPC Data Access server whose items are to be browsed, and call the **RunMe** method. If the result is equal to 1 (**DialogResult.OK**), the user has selected the OPC item, and information about it can be retrieved from the **ItemId** property.

OPC-DA Item Select Dialog

The OPC-DA Item Select Dialog combines together all functionality that allows the user to completely select the OPC item. The selection starts with machine name, and continues with server class (ProgID), Item ID and so on.



After the user is finished with the selection, your code can retrieve the information from properties of the object.

OPC Alarms and Events Tasks

This chapter gives you guidance in how to implement the common tasks that are needed when dealing with OPC Alarms and Events server from the client side. You achieve these tasks by calling methods on the **EasyAEClient** object.

Obtaining Information

Methods described in this chapter allow your application to obtain information from the underlying data source that the OPC Alarms and Events server connects to, or from the OPC server itself (getting condition state). It is assumed that your application already somehow knows how to identify the data it is interested in. If the location of the data is not known upfront, use methods described in Browsing for Information chapter first.

Getting Condition State

In OPC Alarms and Events, information is usually provided in form of event notifications, especially for transient (simple and tracking) events. For condition-related events, however, it is also possible to get (upon request) information about the current state of a specified condition.

If you want to obtain the current state information for the condition instance in an OPC Alarms and Events sever, call the **GetConditionState** method. You pass in individual arguments for machine name, server class, fully qualified source name, condition name, and optionally an array of event attributes to be returned. You will receive back an **AEConditionState** object holding the current state information about an OPC condition instance.

In QuickOPC.NET, you can alternatively pass in a **ServerDescriptor** in place of machine name and server class arguments.

Modifying Information

Methods described in this chapter allow your application to modify information in the underlying data source that the OPC server connects to or in the OPC server itself (acknowledging conditions). It is assumed that your application already somehow knows how to identify the data it is interested in. If the location of the data is not known upfront, use methods described Browsing for Information chapter first.

Acknowledging a Condition

If you want to acknowledge a condition in OPC Alarms and Events server, call the **AcknowledgeCondition** method. You pass in individual arguments for machine name, server class, fully qualified source name, condition name, and an active time and cookie corresponding to the transition of the condition you are acknowledging. Optionally, you can pass in acknowledger ID (who is acknowledging the condition), and a comment string.

In QuickOPC.NET, you can alternatively pass in a **ServerDescriptor** in place of machine name and server class arguments.

Browsing for Information

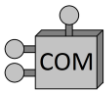
QuickOPC-COM contains methods that allow your application to retrieve and enumerate information about OPC Alarms and Events servers that exist on the network, and data available within these servers. Your code can then make use of the information obtained, e.g. to accommodate to configuration changes dynamically.

The methods we are describing here are for programmatic browsing, with no user interface (or when the user interface is provided by our own code).

Browsing for OPC Servers

If you want to retrieve a list of OPC Alarms and Events servers registered on a local or remote computer, call the **BrowseServers** method, passing it the name or address of the remote machine (use empty string for local computer).

In QuickOPC.NET, you will receive back a **ServerElementCollection** object. If you want to connect to this OPC server later in your code by calling other methods, use the built-in conversion of **ServerElement** to **String**, and pass the resulting string as a **serverClass** argument either directly to the method call, or to a constructor of **ServerDescriptor** object.



In QuickOPC-COM, you will receive back a **Dictionary** of **ServerElement** objects. If you want to connect to this OPC server later in your code by calling other methods, obtain the value of **ServerElement.ServerClass** property, and pass the resulting string as a **serverClass** argument to the method call that accepts it.

Each **ServerElement** contains information gathered about one OPC server found on the specified machine, including things like the server's CLSID, ProgID, vendor name, and readable description.

Browsing for OPC Nodes (Areas and Sources)

Information in an OPC Alarms and Events server is organized in a tree hierarchy (process space), where the branch nodes (*event areas*) serve organizational purposes (similar to folders in a file system), while the leaf nodes actually generate events (similar to files in a file system) – they are called *event sources*. Each node has a "short" name that is unique among other branches or leaves under the same parent branch (or a root). Event sources can be fully identified using a "fully qualified" source name, which determines the OPC event source without a need to further qualify it with its position in the tree. OPC event source may look a process tag (e.g. "FIC101", or "Device1.Block101"), or possibly a device or subsystem identification; their syntax and meaning is fully determined by the particular OPC server they are coming from.

QuickOPC gives you methods to traverse through the address space information and obtain the information available there. It is also possible to filter the returned nodes by a server specific filter string.

If you want to retrieve a list of all event areas under a given parent area (or under a root) of the OPC server, call the **BrowseAreas** method. In QuickOPC.NET, you will receive an **AENodeElementCollection** object. In QuickOPC-COM, you will receive back a **Dictionary** of **AENodeElement** objects. Each **AENodeElement** contains information gathered about one sub-area node, such as its name, or an indication whether it has children.

Similarly, if you want to retrieve a list of event sources under a given parent area (or under a root) of the OPC server, call the **BrowseSources** method. You will also receive back an **AENodeElementCollection** object (in QuickOPC.NET) or a **Dictionary** of **AENodeElement** objects (in QuickOPC-COM), this time containing the event sources only. You can find information such as the fully qualified source name from the **AENodeElement** of any event source, extract it and pass it further to methods like **GetConditionState** or **SubscribeEvents**.

Querying for OPC Event Categories

Each OPC Alarms and Events server supports a set of specific event categories. The OPC specifications define a set of recommended categories; however, each OPC server is free to implement some more, vendor-specific event categories as well.

If you want to retrieve a list of all categories available in a given OPC server, call the **QueryEventCategories** method. In QuickOPC.NET, you will receive back an **AECategoryElementCollection** object; in QuickOPC-COM, you will receive back a **Dictionary** of **AECategoryElement** objects. Each **AECategoryElement** contains information about one OPC event category, such as its (numeric) CategoryId,

readable description, and associated event conditions and attributes. The CategoryId can be later used when creating an event filter, and is provided to you in event notifications.

Subscribing for Information

If your application needs to be informed about events occurring in the process and provided by the OPC Alarms and Events server, it can subscribe to them, and receive notifications.

QuickOPC contains methods that allow you to subscribe to OPC events, change the subscription parameters, and unsubscribe.

Subscribing to OPC Events

Subscription is initiated by calling the **SubscribeEvents** method. The component will call handlers for Notification event for each event that satisfies the filter criteria of the created subscription. Obviously, you first need to hook up event handler for that event, and in order to prevent event loss, you should do it before subscribing.

Events may be generated quite rapidly. Your application needs to specify the *notification rate*, which effectively tells the OPC Alarms and Events server that you do not need to receive event notifications any faster than that.

If you want to subscribe to particular set of OPC Events, call the **SubscribeEvents** method. You can pass in individual arguments for machine name, server class, and notification rate. Optionally, you can specify a subscription filter; it is a separate object of **AESubscriptionFilterType** that you create using **CreateSubscriptionFilter** method. Other optional parameters are attributes that should be returned in event notifications (separate set of attributes for each event category is needed), and the "active" and "refresh when active" flags. You can also pass in a **State** argument of any type. When any event notification is generated, the **State** argument is then passed to the **Notification** event handler in the **EasyAENotificationEventArgs** object.

In QuickOPC.NET, you can alternatively pass in a **ServerDescriptor** in place of machine name and server class arguments. You can also replace the individual notification rate, subscription filter, and returned attributes arguments by passing in an **AESubscriptionParameters** object.

The **State** argument is typically used to provide some sort of correlation between objects in your application, and the event notifications. For example, if you are programming an HMI application and you want the event handler to update the control that displays the event messages, you may want to set the **State** argument to

the control object itself. When the event notification arrives, you simply update the control indicated by the **State** property of **EasyAENotificationEventArgs**, without having to look it up.

The "refresh when active" flag enables a functionality that is useful if you want to keep a "copy" of condition states (that primarily exist in the OPC server) in your application. When this flag is set, the component will automatically perform a subscription Refresh (see further below) after the connection is first time established, and also each time it is reestablished (after a connection loss). This way, the component assures that your code will get notifications that allow you to "reconstruct" the state of event conditions at any given moment.

Note: It is NOT an error to subscribe to the same set of events twice (or more times), even with precisely the same parameters. You will receive separate subscription handles, and with regard to your application, this situation will look no different from subscribing to different set of events.

Changing Existing Subscription

It is not necessary to unsubscribe and then subscribe again if you want to change parameters of existing subscription (such as its notification rate). Instead, change the parameters by calling the **ChangeEventSubscription** method, passing it the subscription handle, and the new parameters (notification rate, and optionally a filter and an "active" flag).

Unsubscribing from OPC Events

If you no longer want to receive event notifications, you need to unsubscribe from them. To unsubscribe from events that you have previously subscribed to, call the **UnsubscribeEvents** method, passing it the subscription handle.

You can also unsubscribe from all events you have previously subscribed to (on the same instance of **EasyAEClient** object) by calling the **UnsubscribeAllEvents** method.

If you are no longer using the parent **EasyAEClient** object, you should unsubscribe from the events, or dispose of the **EasyAEClient** object, which will do the same for you. Otherwise, the subscriptions will internally be still alive, and may cause problems related to COM reference counting.

Refreshing Condition States

Your application can obtain the current state of all conditions which are active, or which are inactive but unacknowledged, by requesting a "refresh" from an event subscription. The component will respond by sending the appropriate event

notifications to the application, via the event handlers, for all conditions selected by the event subscription filter. When invoking the event handler, the component will indicate whether the invocation is for a refresh or is an original notification. Refresh and original event notifications will not be mixed in the same event notifications.

If you want to force a refresh, call the **RefreshEventSubscription** method, passing it the subscription handle.

Notification Event

When an OPC Alarms and Events server generates an event, the **EasyAEClient** object generates a **Notification** event. For subscription mechanism to be useful, you should hook one or more event handlers to this event.

To be more precise, the **Notification** event is actually generated in other cases, too - if there is any significant occurrence related to the event subscription. This can be for three reasons:

1. You receive the **Notification** when a successful connection (or re-connection) is made. In this case, the **Exception** and **Event** properties of the event arguments are null references.

2. You receive the **Notification** when there is a problem with the event subscription, and it is disconnected. In this case, the **Exception** property contains information about the error. The **Event** property is a null reference.

3. You receive one additional **Notification** after the component has sent you all notifications for the forced "refresh". In this case, the **RefreshComplete** property of the event arguments is set to True, and the **Exception** and **Event** properties contain null references.

The notification for the **Notification** event contains an **EasyAENotificationEventArgs** argument. You will find all kind of relevant data in this object. Some properties in this object contain valid information under all circumstances. These properties are **ServerDescriptor**, **SubscriptionParameters**, **Handle**, and **State**. Other properties, such as **Event**, contain null references when there is no associated information for them. When the **Event** property is not a null reference, it contains an **AEEvent** object describing the detail of the actual OPC event received from the OPC Alarms and Events server.

Before further processing, your code should always inspect the value of **Exception** property of the event arguments. If this property is not a null reference, there has been an error related to the event subscription, the **Exception** property contains

information about the problem, and the **Event** property does not contain a valid object.

If the **Exception** property is a null reference, the notification may be informing you about the fact that a "forced" refresh is complete (in this case, the **RefreshComplete** property is True), or that an event subscription has been successfully connected or re-connected (in this case, the **Event** property is a null reference). If none of the previous applies, the **Event** property contains a valid **AEEvent** object with details about the actual OPC event generated by the OPC server.

Pseudo-code for the full **Notification** event handler may look similar to this:

> **if** notificationEventArgs.Exception is not null **then**
> *An error occurred and the subscription is disconnected, handle it (or ignore)*
> **else if** notificationEventArgs.RefreshComplete **then**
> *A "refresh" is complete; handle it (only needed if you are invoking a refresh explicitly)*
> **else if** notificationEventArgs.Event is null **then**
> *Subscription has been successfully connected or re-connected, handle it (or ignore)*
> **else**
> *Handle the OPC event, details are in notificationEventArgs.Event. You may use notificationEventArgs.Refresh flag for distinguishing refreshes from original notifications.*

The **Notification** event handler is called on a thread determined by the **EasyAEClient** component. For details, please refer to "Multithreading and Synchronization" chapter under "Advanced Topics".

There is also an event called **MultipleNotifications** that can deliver multiple notifications in one call to the event handler. See Multiple Notifications in One Call in Advanced Topics for more.

Using Callback Methods Instead of Event Handlers

The subscription methods also allow you to directly specify the callback method (delegate) to be invoked for each event notification you are subscribing to.

For detailed discussion on this subject, please refer to "Using Callback Methods Instead of Event Handlers" under the "OPC Data Access Tasks" chapter. All information presented there applies to OPC Alarms and Events as well.

Setting Parameters

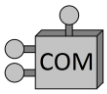
While the most information needed to perform OPC tasks is contained in arguments to method calls, there are some component-wide parameters that are not worth repeating in every method call, and also some that have wider effect that influences more than just a single method call. You can obtain and modify these parameters through properties on the **EasyAEClient** object, and (in QuickOPC-COM) by EasyOPC Options Utility.

Following are instance properties, i.e. if you have created multiple **EasyAEClient** object, each will have its own copy of them:

- **ClientMode**: Allows you to influence how EasyOPC performs various operations on OPC Alarms and Events servers.
- **HoldPeriods**: Specifies optimization parameters that reduce the load on the OPC server.

Instance properties can be modified from your code.

In QuickOPC.NET, if you have placed the **EasyAEClient** object on the designer surface, the instance properties can also be directly edited in the Properties window in Visual Studio.



In QuickOPC-COM, the default values of instance properties can be set using the EasyOPC Options utility. Your code can still override the defaults if needed.

Following properties are static, i.e. shared among all instances of **EasyAEClient** object:

- **EngineParameters**: Contains global parameters such as frequencies of internal tasks performed by the component.
- **MachineParameters**: Contains parameters related to operations that target a specific computer but not a specific OPC server, such as browsing for OPC servers using various methods.
- **ClientParameters**: Contains parameters that influence operations that target a specific OPC server a whole.
- **LinkParameters**: Contains parameters that influence how EasyOPC works with live OPC event subscriptions.

Static properties can only be modified from your code (in QuickOPC.NET) or using the EasyOPC Options utility (in QuickOPC-COM).

Please use the Reference documentation (and, in QuickOPC-COM, the EasyOPC Options Help) for details on meaning of various properties and their use.

EasyOPC.NET Extensions

EasyOPC.NET Extensions is a pack of .NET classes that adds more functionality to the EasyOPC.NET component. It is built upon and relies on EasyOPC.NET, so in fact you could build all these extensions yourself, but it is meant to provide a ready-made, verified code for useful “shortcut” methods to achieve common tasks.

Usage

In order to use the EasyOPC.NET Extensions, you need to reference the **OpcLabs.EasyOpcClassicExtensions** assembly in your project.

A significant part of the EasyOPC.NET Extensions functionality is provided in form of so-called Extension Methods. In languages that support them (including C#, VB.NET), extension methods will appear as additional methods on the classes that are being extended. For example, if you reference **OpcLabs.EasyOpcClassicExtensions** assembly in your project, the **EasyDAClient** class will appear as having many more new methods that you can choose from. This way, you can write a code that call e.g. **GetDataTypePropertyValue** (extension) method on the **EasyDAClient** object, although the method is actually located in the **EasyDAClientExtension** class which you do not even have to be aware of.

In languages that do not support extension method syntax, it is still possible to use them, but they need to be called as static method on the extension class, and you provide the object reference as an additional argument. In the above example, you would call **EasyDAClientExtension.GetDataTypePropertyValue** instead, and pass it the **EasyDAClient** object as the first argument.

Data Access Extensions

OPC Properties

Type-safe Access

With EasyOPC.NET Extensions, you can use type-safe methods that allow obtaining a value of an OPC property value already converted to the specified type, with methods such as **EasyDAClient.GetPropertyValueInt32**. There is such a method for each primitive type, named **GetPropertyValueXXXX**, where **XXXX** is the name of the type. Using these methods allows your code be free of unnecessary conversions.

A corresponding set of methods also exists for one-dimensional arrays of primitive types. Such methods are named **GetPropertyValueArrayOfXXXX**, where **XXXX** is the name of the element type. For example, **EasyDAClient.GetPropertyValueArrayOfString** will obtain a value of a property as an array of strings.

Well-known Properties

A common scenario is to work with well-known OPC properties. With EasyOPC.Net Extensions, you can quickly obtain a value of any well-known OPC property of a given OPC item, with methods such as **EasyDAClient.GetDataTypePropertyValue**. All these methods are named **GetXXXXPropertyValue**, where **XXXX** is the name of the property. The methods also check the value type and convert it to the type that corresponds to the property. For example, **GetDataTypePropertyValue** method returns a **VarType**, **GetScanRatePropertyValue** method returns a **float**, and **GetDescriptionPropertyValue** method returns a **string**.

Alternate Access Methods

The **GetPropertyValueDictionary** method allows you to obtain a dictionary of property values for a given OPC item, where a key to the dictionary is the property Id. You can pass in a set of property Ids that you are interested in, or have the method obtain all well-known OPC properties. You can then easily extract the value of any property by looking it up in a dictionary (as opposed to having to numerically index into an array, as with the base **GetMultiplePropertyValues** method).

The **GetItemPropertyRecord** method allows you to obtain a structure filled in with property values for a given OPC item. It can retrieve all well-known properties at once, or you can pass in a set of property Ids that you are interested in. You can then simply use the properties in the resulting **DAItemPropertyRecord** structure, without further looking up the values in any way.

The static **DAPropertyIDSet** class gives you an easy way to provide pre-defined sets of properties to the above methods. There are well-known sets such as the Basic property set, Extension set, or Alarms and Events property set. It also allows you to combine the property sets together (a union operation), with the **Add** method or the '+' operator.

OPC Items

Type-safe Access

With EasyOPC.NET Extensions, you can use type-safe methods that allow reading an item value already converted to the specified type, with methods such as

EasyDAClient.ReadItemValueInt32. There is such a method for each primitive type, named **ReadItemValueXXXX**, where **XXXX** is the name of the type. Using these methods allows your code be free of unnecessary conversions. The methods also take care of passing a proper requested data type to the OPC server.

A corresponding set of methods also exists for one-dimensional arrays of primitive types. Such methods are named **ReadItemValueArrayOfXXXX**, where **XXXX** is the name of the element type. For example, **EasyDAClient.ReadItemValueArrayOfInt32** will read from an item as an array of 32-bit signed integers.

You can also use type-safe methods that allow writing an item value of a specified type, with methods such as **EasyDAClient.WriteItemValueInt32**. There is such a method for each primitive type, named **WriteItemValueXXXX**, where **XXXX** is the name of the type. Using these methods allows your code be free of unnecessary conversions. The methods also take care of passing a proper requested data type to the OPC server.

A corresponding set of methods also exists for one-dimensional arrays of primitive types. Such methods are named **WriteItemValueArrayOfXXXX**, where **XXXX** is the name of the element type. For example, **EasyDAClient.WritetemValueArrayOfInt32** will write into an item as an array of 32-bit signed integers.

Software Toolbox Extender Replacement

QuickOPC.NET can serve as a replacement for Software Toolbox Extender (www.opcextender.net) component. For further details, please refer to a separate document installed with the product.

Application Deployment

This chapter describes how to deploy applications developed with use of QuickOPC.

For some uses, it is acceptable to perform the deployment manually. In other cases, you will find yourself writing an installer for your application, and you will probably want to include the installation of QuickOPC components in it as well.

Deployment Elements

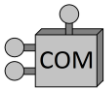
Assemblies

Depending on which assemblies you have referenced in your application, you may need one or more of the following files be installed with your application:

- OpcLabs.BaseLib.dll
- OpcLabs.EasyOpcClassic.dll
- OpcLabs.EasyOpcClassicExtensions.dll
- OpcLabs.EasyOpcClassicForms.dll
- OpcLabs.EasyOpcClassicInternal.dll

Please refer to “Product Parts” chapter for description of purpose of individual assemblies. You will find them under the Assemblies subdirectory in the installation directory of the product.

The assemblies need to be placed so that the referencing software (your application) can find them, according to standard Microsoft .NET loader (Fusion) rules. The easiest is to simply place the assembly files alongside (in the same directory as) your application file.



Development Libraries and COM Components

Depending on which components you have referenced in your application, you may need one or more of the following files be installed with your application:

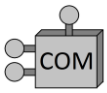
File Name(s)	Description	Additional Action(s)
easyopcl.exe	EasyOPC Local Server/Service	Register by running with /RegServer or /Service switch (depending on your needs)
easyopci.dll	EasyOPC In-process Server	Register using RegSvr32
easyopcm.dll	EasyOPC Messages	Register using RegSvr32

easyopct.dll	EasyOPC Type Library	Register using RegSvr32
OPCUserObjects.exe	OPC User Objects Component and Type Library	Register by running with /Register or /RegServer Service switch (these switches are equivalent)

Please refer to "Product Parts" chapter for description of purpose of individual libraries and components. You will find them under the Bin subdirectory in the installation directory of the product.

The EasyOPC Type Library is needed with either EasyOPC Local Server/Service or EasyOPC In-process Server. The EasyOPC Messages file is needed if you want to view the descriptive texts of categories and events generated by the EasyOPC Local Server/Service in the event viewer.

You can choose the placement of the files freely. The registration stores the path information into the registry, and that's how the system finds them consequently. Usually, you may have a folder calling something like 'Bin' in your application to hold these files, but that is just a recommendation.



Management Tools

Depending on which tools the users of your application will use, you may need one or more of the following files be installed with your application:

File Name(s)	Description	Additional Action(s)
EasyOpcOptions.chm **EasyOpcOptions.exe**	EasyOPC Options Utility	(none)
EventLogOptions.chm, **EventLogOptions.exe**	Event Log Options Utility	(none)

You can choose the placement of the files freely. The only condition is that the CHM file has to be in the same folder as the corresponding EXE file. Usually, you may have a folder calling something like 'Bin' in your application to hold these files, but that is just a recommendation.

Prerequisites

When using QuickOPC.NET: Besides the actual library assemblies, QuickOPC.NET requires that following software is present on the target system:

1. Microsoft .NET Framework 3.5 with Service Pack 1 (Full or Client Profile), or Microsoft .NET Framework 4 (Full or Client Profile).

Needed when: Always (the choice depends on the framework that you target).

2. Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package ATL Security Update. It is located in the "Redist" folder under the QuickOPC installation. You need to select the package appropriate for the targeted platform (x86 or x64), or both. If you are going to redistribute it within your own installer, you can call it with "/q" on the command line, for silent installation.

   Needed when: Always.

   IMPORTANT: Only use the package from the "Redist" folder in QuickOPC installation. Do not use the package from Visual Studio installation or other sources on the Web, as there are several versions of it, and only one is guaranteed to work.

3. OPC Core Components 3.00 Redistributable (x86 or x64), version 3.00.105.0 or later. It is located in the "Redist" folder under the QuickOPC installation. You need to select the package appropriate for the targeted platform (x86 or x64).

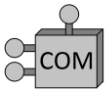
   

   IMPORTANT: On 64-bit system, only install the 64-bit package, even if you intent to use some 32-bit OPC components. The 64-bit package includes everything that is needed for 32-bit OPC as well. If you are going to redistribute it within your own installer, you can call it via MSIEXEC.EXE with "/q REBOOT=Suppress" on the command line, for silent installation.

   Needed when: Always.

   Note that due to a bug in the OPC Core Components installation, it may silently skip installation of certain critical components (such as OPC COM proxy/stub) if .NET Framework 2.0 or later isn't present on the system at the time of installation. Always make sure that .NET Framework is installed, BEFORE installing OPC Core Components.



When using QuickOPC-COM: Besides the development libraries and COM components, QuickOPC-COM requires that following software is present on the target system:

1. Microsoft .NET Framework 3.5 with Service Pack 1 (Full or Client Profile), or Microsoft .NET Framework 4 (Full or Client Profile).

   QuickOPC-COM does not directly require the Microsoft .NET Framework 3.5, but OPC Core Components setup may fail without it.

1. Microsoft Visual C++ 2010 Redistributable Package. It is located in the "Redist" folder under the QuickOPC installation. You need to select the package appropriate for the targeted platform (x86 or x64). If you are going to redistribute it within your own installer, you can call it with "/q" on the command line, for silent installation.

2. OPC Core Components 3.00 Redistributable (x86). It is located in the "Redist" folder under the QuickOPC installation. If you are going to redistribute it within your own installer, you can call it via MSIEXEC.EXE with "/q REBOOT=Suppress" on the command line, for silent installation.

Licensing

Proper license (for runtime usage) must be installed on the target computer (the computer where the product will be running). The License Manager utility is needed for this. It is contained in LicenseManager.exe file, located under the Bin subdirectory in the installation directory of the product.

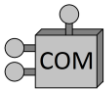
Deployment Methods

Manual Deployment

In order to deploy your application with QuickOPC.NET manually, follow the steps below:

1. Check that proper version of Microsoft .NET Framework is installed, and if it is not, install it. Note that QuickOPC-COM also needs the Microsoft .NET Framework 3.5, for OPC Core Components.
2. Run the QuickOPC installation program, selecting "Production installation" type.
3. Perform any steps needed to install your own application.

4a) Copy the QuickOPC.NET assemblies to their target locations for your application.



4b) Copy QuickOPC-COM development libraries, COM components and management tools to their target locations for your application, and perform additional actions (such as their registration) as described with the respective files.

5. Run the License Manager from the Start menu, and install the runtime license.

Automated Deployment

The installer for your application should contain following steps:

1. Check that proper version of Microsoft .NET Framework is installed, and if it is not, instruct the user to install it. Note that QuickOPC-COM also needs the Microsoft .NET Framework, for OPC Core Components.
2. Only with QuickOPC.NET: Install Microsoft Visual C++ 2008 SP1 Redistributable Package for the appropriate platform, if needed.
3. Install Microsoft Visual C++ 2010 Redistributable Package for the appropriate platform.
4. Install OPC Core Components 3.00 Redistributable for the appropriate platform.
5. Perform any steps needed to install your own application.
3. Install QuickOPC.NET assemblies to their target locations for your application.
6. Install LicenseManager.exe (from Bin or Bin\x64)
7. Offer the user an option to run the License Manager.

The user who deploys the application will then:

1. Run your installer and follow the steps.
2. Use the License Manager and install the runtime license.

If the above described procedure for installing the license (presenting the user with the License Manager utility user interface) does not fit the needs of your application, please contact your vendor, describe the situation, and ask for alternative license installation options.

Advanced Topics

OPC Specifications

QuickOPC.NET components directly support following OPC specifications:

- all OPC DA (Data Access) 1.0x Specifications (Released)
- all OPC DA (Data Access) 2.0x Specifications (Released)
- all OPC DA (Data Access) 3.0x Specifications (Released)
- OPC Alarms and Events Custom Interface Standard 1.00, 1.01 and 1.10 (Released)
- all OPC Common 1.0x Specifications (Released)
- OPC Common 1.10 Specification (Draft)

QuickOPC.NET components support following OPC specifications indirectly:

- OPC UA (Universal Architecture) 1.00 Specifications for Data Access
- OPC UA (Universal Architecture) 1.01 Specifications for Data Access

OPC-UA (Universal Architecture)

The Unified Architecture (UA) is the next generation OPC standard that provides a cohesive, secure and reliable cross platform framework for access to real time and historical data and events.

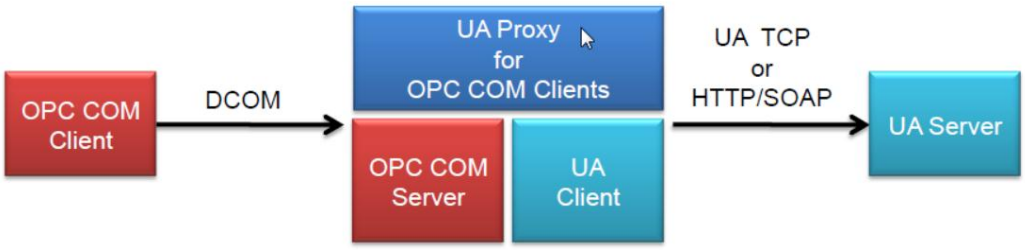
A separate product, QuickOPC-UA, allows native connections to OPC Unified Architecture servers.

QuickOPC-Classic is not a native OPC UA client, but you can still use it to connect to OPC UA servers, using so-called UA COM Proxy that is shipped with the product as part of OPC UA COM Interop Components.

The OPC UA COM Interop Components from OPC Foundation make it possible for existing COM-based applications to communicate with UA applications. OPC Labs is using them to add UA support to existing products. Support for OPC UA COM Interop Components is not currently provided.

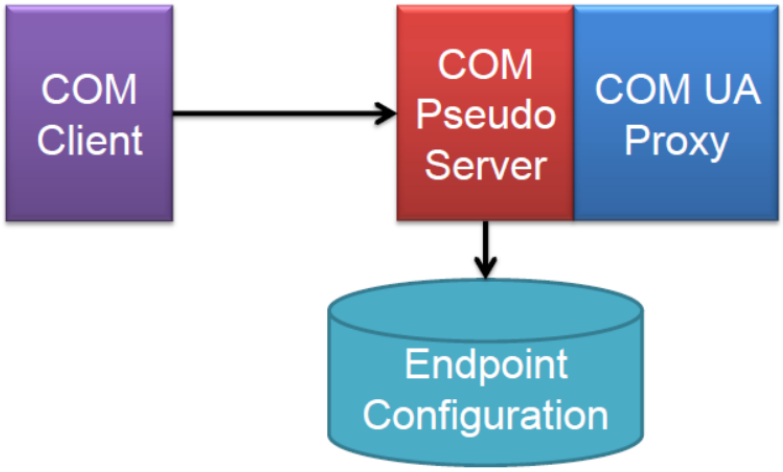
The components that allows COM clients (such as QuickOPC-COM and QuickOPC.NET) to talk to UA server is called the UA Proxy (or UA COM Proxy). It is a DCOM server that implements the different OPC COM specifications. A COM client (QuickOPC) uses DCOM to talk to this proxy, usually on the local machine (i.e. the same machine

where the client is), and the proxy translates OPC COM calls into UA calls, and fetches the information as required from the UA server. It is a dynamic operation – all the information about address space and data values is retrieved dynamically from the UA server.



The UA COM Proxy is a DCOM server, meaning that it has its own ProgID and CLSID (class ID). However, there needs to be some mapping between the particular ProgID and a particular UA endpoint. The UA COM proxy relies on a concept of a Pseudo-server, which maps a ProgID to a specific UA endpoint, which is stored in the configuration file. The configuration tool that is made available as part of OPC UA COM Interop Components has the ability to select a UA endpoint and create one of these pseudo-servers that a COM client can then connect to. This endpoint configuration file is stored on disk in XML format.

This file also stores state information, which is necessary for replicating COM client configuration across multiple machines. If you set up a client on a particular machine, talking to a particular UA server, and do all the necessary configuration, and you then want to take the configuration and install it on multiple other machines, you can simply copy that endpoint configuration file along. The file contains the ProgID and CLSID of the pseudo-server, and the endpoint information.



The configuration tool for OPC UA COM Proxy can be found in your Start menu, under OPC Foundation → UA SDK 1.01 → UA Configuration Tool. In order to make a UA server visible to COM clients through the UA COM Proxy, select the "Manage COM Interop" tab, press the "Wrap UA Server" button, and fill in the information required by the program. You need to specify the endpoint information for a UA server (possibly using a UA discovery mechanism), the UA connection parameters, and finally the COM pseudo-server CLSID and ProgID (the tool offers reasonable defaults). After you have finished this step, the pseudo-server appears in the list, and OPC COM clients (QuickOPC-COM, QuickOPC.NET) can connect to it.

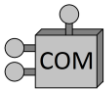
Note: The QuickOPC setup program has the "OPC UA COM Interop Components" option turned off by default, because it has other dependencies and effects on the system that can complicate the typical setup. If you want to connect to OPC-UA servers, make sure that you enable "OPC UA COM Interop Components" in the installation first.

OPC Interoperability

The QuickOPC.NET and QuickOPC-COM components have been extensively tested for OPC interoperability, with various OPC servers from many vendors, and on different computing environments. The tests also include regular participation in OPC Foundation's Interoperability Workshops, where OPC software is intensively "grilled" for interoperability.

Having tried so many different OPC servers to connect to, we have encountered different (though still correct) interpretations of the same OPC specifications, and also certain common (and less common) divergences from the specifications. QuickOPC.NET components do not try to "turn down" any OPC server for compliance problems. Just the opposite: Wherever possible, we have taken a "relaxed" approach to how the OPC servers are written, and allow and accept the above mentioned variations. This gives QuickOPC.NET even wider interoperability scope.

Based on the interoperability results (which can be viewed on OPC Foundation's web site), QuickOPC.NET and QuickOPC-COM have also been granted the OPC Foundations' "Self-tested for Compliance" logo. Note that in contrast with the logo title, the conditions of this logo program actually require the OPC client software be tested in presence and with cooperation of OPC Foundation's Test Lab delegate.

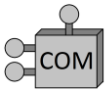


Event Logging

QuickOPC-COM is capable of logging various errors and events, using the standard Windows mechanisms, or other means. By default, only the most important events are logged.

You can use the Event Log Options utility (available from Start menu) to influence which events get logged, and also to select logging into a plain text file instead of Windows Even Log. The Event Log Options utility has separate documentation and help file; please refer to it for details on setting the options.

Event logging is only performed by the EasyOPC Local Server/Service component. The EasyOPC In-process Server component and OPC User Objects do not log any events.

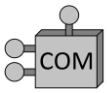


EasyOPC Options Utility

EasyOPC component in QuickOPC-COM comes with predefined settings that are suitable for most applications. For large-volume operations, or specialized needs, it may be necessary to fine-tune the settings, using the EasyOPC Options utility. You can invoke the utility from the Start menu, under the application's program group. The EasyOPC Options utility has separate documentation and help file; please refer to it for details on setting the options.

Be aware that the component only "picks up" the EasyOPC options at the startup time. You should therefore set the proper options in advance, and start the application afterwards.

EasyOPC Options utility can also be invoked from command line with a /ResetOptions switch. Doing so acts like pressing the “Factory Defaults” button, but the application will not display any user interface, and will quickly return the control back. You can use this feature e.g. in automated installations.



COM Registration and Server Types

The EasyOPC component of QuickOPC-COM can be deployed as following COM server types:

- In-process Server, or
- Local Server, or
- Service.

There are various advantages and disadvantages of the above options, which are discussed in details in Microsoft COM materials. In brief, the In-process Server loads

the component's code into the client process, which isolates it from other uses on the same computer, and provides fastest data exchange between the component and your code, but there are also security concerns that have to do with the fact the component has access to your application's memory. The Local Server and Service options run in a process that is separate from your application. The Service can be better controlled while running (started, stopped, etc., using the Service Manager).

The EasyOPC component can be registered as multiple server types at the same time, e.g. Local Server and In-process Server. The client can then choose which server type it connects to, or let the COM infrastructure select the server type automatically. Note that, however, registration as Local Server is mutually exclusive with registration as Service.

You can use the EasyOPC Options utility to register and unregister the available server types, except for making a choice between Local Server or Service.

To register EasyOPC component to run as Local Server (and not as Service):

- Open the Command Prompt window.
- Navigate to the "bin" subdirectory of the QuickOPC-COM installation folder.
- Type the following command:
  ```
  easyopcl /RegServer
  ```

To register EasyOPC component to run as Service (and not as Local Server):

- Open the Command Prompt window.
- Navigate to the "bin" subdirectory of the QuickOPC-COM installation folder.
- Type the following command:
  ```
  easyopcl /Service
  ```

Object Serialization

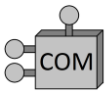
QuickOPC.NET allows you to easily store objects (and object graphs, i.e. interconnected objects) into files and streams, and also to load them back.

Two types of serialization are supported:

- Basic serialization using **Serializable** attribute and/or **ISerializable** interface. This serialization type is typically used with **BinaryFormatter** for storing objects in a binary format.
- XML serialization using **XmlSerializer** and **IXmlSerializable**. This serialization provides objects storage in XML format.

Practically all QuickOPC objects (and their collections and dictionaries) can be serialized and deserialized. For example:

- You can load and store **EasyDAClient** and **EasyAEClient** objects (their instance properties, i.e. various parameters, are serialized).
- You can load and store parameter objects, such as **DAGroupParameters**.
- You can load and store arguments (and arrays of arguments) passed to functions, such as lists of items to be subscribed to (**DAItemGroupArguments**). This functionality can be used e.g. with storing lists of subscribed items in file, outside your application code.
- You can load and store results of browsing and querying (e.g. **DANodeElementCollection, AECategoryElementCollection**).
- You can load and store results of reading (e.g. **DAVtq**).
- You can load and store condition states (**AEConditionState**) and event data (**AEEvent**).
- You can load and store all notification data contained in event arguments, for OPC Data Access item changes (**EasyDAItemChangedEventArgs**) or OPC Alarms and Events notifications (**EasyAENotificationEventArgs**). This functionality can be used e.g. for logging significant changes and events in the underlying system.



Asynchronous Operations

All “normal” method calls on EasyOPC object in QuickOPC-COM are performed synchronously with respect to the caller, i.e. the methods perform their work and the caller is blocked until the operation is either complete, or times out. Internally the actual work is performed on a different thread, but this is more or less an implementation details that does not you much as a developer.

For advanced scenarios, EasyOPC component supports a concept of asynchronous operations, too. The methods that perform asynchronously are prefixed with the word **Invoke**, and they are:

- **InvokeReadItem**,
- **InvokeReadItemValue**,
- **InvokeWriteItem**,
- **InvokeWriteItemValue**.

It should be made clear that the “asynchronicity” discussed here refers only to the interaction between your application and the EasyOPC Component. It does NOT refer to the nature of the calls made by the EasyOPC Components to the target OPC Server. The usage of various OPC methods such as synchronous and asynchronous operations

is controlled separately (see **ClientMode** in Setting Parameters), and by default, EasyOPC always prefers the recommended method, i.e. asynchronous read/write methods, internally. With proper settings, it is perfectly possible to make synchronous call to EasyOPC that would internally trigger an asynchronous OPC method, and also vice versa.

All **InvokeXXXX** methods accept a **State** among their input arguments, and they all return an **AsyncActivity** object. The call to **InvokeXXXX** returns very quickly, just initiating the requested operation inside the component. There are two ways how you can monitor the progress and obtain the actual result of the operation:

- By polling the **Completed** property of the returned **AsyncActivity**, you can determine whether the operation has completed. When completed, the outcome is available through **OperationResult** property of the **AsyncActivity** object.
- By hooking up an event handler for the **OperationCompleted** event on the **EasyDAClient** object, your code will be notified when the asynchronous operation completes. The event notification carries an **OperationCompletedEventArgs** object that, among other things, contains **OperationResult** object which has the actual outcome of the operation.

The type of **OperationResult** object contained in **AsyncActivity** or **OperationCompletedEventArgs** depends on the actual asynchronous method that had been invoked. For example, when you call **InvokeReadItem**, the operation outcome is of **DAVtqResult** type.

Note that timeouts apply to asynchronous operations in the same way as to synchronous operation, and therefore at some moment, every asynchronous operation always completes, though it may be with timeout (or some other) error.

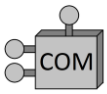
There is also an event called **MultipleOperationsComplete** that can deliver multiple results in one call to the event handler. See Multiple Notifications in One Call for more.

Multiple Notifications in One Call

QuickOPC offers the possibility to process multiple event notifications in one call to an event handler. For applications that involve working with events generated with high frequency, this approach may improve the efficiency and therefore throughput, however the precise performance needs to be measured for each application, and is highly influenced by other factors (such as the code in the event handler, and the way it accessed the event arguments).

QuickOPC processes events in “chunks”, and this gives it the opportunity to merge multiple event notifications into a single call. For each event that has a handler for individual calls, QuickOPC also provides a complementary event handler that delivers multiple notifications in a single call. The complementary event also has a different type for its arguments. The correspondence is described in a table below.

Object	Individual Notifications		Multiple Notifications in One Call	
	Handler Name	Arguments Type	Handler Name	Arguments Type
EasyDAClient	Operation-Completed (*)	OperationCompleted-EventArgs	MultipleOperations-Completed	MultipleOperationsCompleted-EventArgs
	ItemChanged	EasyDAItemChanged-EventArgs	MultipleItems-Changed	EasyDAMultipleItemsChanged-EventArgs
EasyAEClient	Notification	EasyAENotification-EventArgs	Multiple-Notifications	EasyAEMultipleNotifications-EventArgs



(*) OperationCompleted event is available in QuickOPC-COM only.

The event arguments types for events bearing multiple notifications are all constructed the same: They have only one member, a property called **ArgsArray**. This property contains an array of what would be the event arguments of individual notifications. The event handler is supposed to loop through this array sequentially and process its elements as they would be event arguments of separate calls. The elements of the array should be processed from beginning to the end, as this order corresponds to the time order in which the individual calls would be made.

Internally, the component “chops” the event stream by event type, so that only events of the same type are delivered together. The component always repeatedly calls the handler for individual notifications first (if such handler exists), and then proceeds to call the handler for multiple notifications (if it exists). Although it is possibly to hook to both handlers on the same object, such practice would rarely if ever make any sense. Always hook to either the handler for individual notifications, or the handler for multiple notifications, but not both.

Internal Optimizations

OPC is quite “sensitive” to proper usage, with regard to efficiency. QuickOPC performs many internal optimizations, and uses the knowledge of proper approaches and procedures to effectively handle the communication with OPC servers.

Here are some of the optimizations implemented in QuickOPC:

- Wherever possible, OPC operations are performed on multiple elements at once.

- OPC items with similar update rates are put into a common OPC group.
- OPC items with similar percentage deadbands are put into a common OPC group.
- OPC items are not removed from OPC groups immediately, but only if not used for certain amount of time.
- OPC item data is held in memory, and if fresh enough, the value from memory is taken, and no OPC call is made to satisfy the Read request.
- OPC asynchronous calls are preferred over synchronous calls.
- Minimum update rates are enforced, so that the system cannot be easily overloaded.
- Multiple uses of the same OPC server or same OPC item in the user application are merged into a single request to the OPC.
- Internal queues are used to make sure that OPC callbacks cannot be blocked by user code.

Failure Recovery

The OPC communication may fail in various ways, or the OPC client may get disconnected from the OPC server. Here are some examples of such situations:

- The OPC server may not be registered on the target machine – permanently, or even temporarily, when a new version is being installed.
- The DCOM communication to the remote computer breaks due to unplugged network cable.
- The remote computer running the OPC server is shut down, or restarted, e.g. for security update.
- The configuration of the OPC server is changed, and the OPC item referred to by the OPC clients no longer exists. Later, the configuration could be changed again and the OPC item may reappear.
- The OPC server indicates a serious failure to the OPC client.
- The OPC server asks its clients to disconnect, e.g. for internal reconfiguration.

QuickOPC handles all these situations, and many others, gracefully. Your application receives an error indication, and the component internally enters a “wait” period, which may be different for different types of problems. The same operation is not reattempted during the wait period; this approach is necessary to prevent system overload under error conditions. After the wait period elapses, QuickOPC will retry the operation, if still needed at that time.

All this logic happens completely behind the scenes, without need to write a single line of code in your application. QuickOPC maintains information about the state it

has created inside the OPC server, and re-creates this state when the OPC server is disconnected and then reconnected. Objects like OPC groups and OPC items are restored to their original state after a failure.

Even if you are using the subscriptions to OPC items or events, QuickOPC creates illusion of their perseverance. The subscriptions outlive any failures; you do not have to (and indeed, you should not) unsubscribe and then subscribe again in case of error. After you receive event notification which indicates a problem, simply stay subscribed, and the values will start coming in again at some future point.

Timeout Handling

The core QuickOPC methods (**ReadItem, ReadItemValue**, **WriteItemValue**, and their counterparts that work with multiple items at once) are all implemented as synchronous function calls with respect to the caller, i.e. they perform some operation and then return, passing the output to the caller at the moment of return. However, this does not mean that the component makes only synchronous calls to OPC servers while you are calling its methods. Instead, QuickOPC works in background (in separate threads of execution) and only uses the method calls you make as "hints" to perform proper data collection and modifications.

Internally, QuickOPC maintains connections to requested OPC servers and items, and it establishes the connections when you ask for reading or writing of certain OPC item. QuickOPC eventually disconnects from these servers and items if they are no longer in use or if their count goes beyond certain limits, using its own LRU-based algorithm (Least Recently Used).

When you call any of the core QuickOPC methods, the component first checks whether the requested item is already connected and available inside the component. If so, it uses it immediately (for reading, it may provide a cached value of it). At the same time, the request that you just made by calling the method is used for dynamic decisions on how often the item should be updated by the server etc.

If the item is not available, QuickOPC starts a process to obtain it. This process has many steps, as dictated by the OPC specifications, and it may take some significant time. The method call you just made does not wait indefinitely until the item becomes available. Instead, it uses certain timeout values, and if the item does not become available within these timeouts, the method call returns. The connection process is totally independent of the method that was called, meaning that no problem in the connection process (even an ill-behaved server, or a broken DCOM connection) can cause the calling method to wait longer than the timeouts dictate.

The timeout values are accessible via **Timeouts** property on the **EasyDAClient** object. The explanation of the individual timeout values is provided in the Reference documentation.

Note that if the nature of the situation allows the component to determine that the item will NOT be available, the method will return earlier (before the timeouts elapse) with the proper error indication. This means that not every connection problem causes the method to actually use the full value of the timeouts. For example, when the server refuses the item because the item has an incorrect name, this error is passed immediately to the caller.

It is important to understand that even if the method call times out because the connection process was not finished in time, the connection process itself is not cancelled and may continue internally. This means that next time the same item is requested, it may be instantly available if the connection process has finished. In other words, the timeouts described above affect the way the method call is executed with respect to the caller, but do not necessarily affect at all the way the connection is performed.

When you create an **EasyDAClient** object, the timeout values are set to reasonable defaults that work well with reporting or computation type of OPC applications. In these applications, you know that you MUST obtain certain value within a timeout, otherwise the application will not be doing what is intended to do: e.g. the report will not contain valid data, or the computations will not be performed. When the requested item is not instantly available (for example, the server is not started yet), the application can afford delays in processing (method calls made to **EasyDAClient** object may block the execution for certain time). For this kind of applications, you may leave the default timeout values, or you may adjust them based on the actual configuration and performance of your system.

There is also a different kind of applications, typically an HMI screen, which wants to periodically update the values of controls displayed to the user. The screen usually contains larger number of these controls that are refreshed in a cyclic way by the application. If possible, you should use subscription-based updated for these applications, and in such case the timeouts are of much lesser importance. But, in some application the subscriptions are not practical, and you resort to periodic reading (polling). The fact that SOME data is not instantly available should not be holding the update of others. It is perfectly OK to display an indication that the data is not available momentarily, and possibly display them in some future refresh cycle when they become available. For this kind of application, you may prefer to set all the above mentioned timeout properties to lower values. This assures that the refresh

loop always goes quickly over all the controls on the screen, no matter whether the data is available for them immediately, or only in a postponed fashion.

To simplify this explanation, you can also say that if you need the OPC values for further *sequential* processing, reasonably long timeouts are needed (and the defaults should serve well in most situations). If you are refreshing the data on a *cyclic* basis by polling, you will probably need to set the timeouts to lower values.

Data Types

OPC Data Access specification is based on COM, and uses Windows VARIANT type (from COM Automation) for representing data values.

Note: Some OPC servers even use certain VARIANT types that are not officially supported by Microsoft.

Microsoft .NET Framework has a different concept, and all data is represented using an **Object** type. Conversions between the two are available, but not always fully possible.

In addition, not everything that can be stored in an **Object** can later be processed by all .NET tools and languages. Microsoft has created so-called Common Language Specification (CLS), which has certain rules and restrictions that, if followed, guarantee cross-language compatibility. Public QuickOPC.NET components (assemblies) are fully CLS compliant, and that includes the way the data types are converted to and from OPC types.

QuickOPC.NET converts data from COM to .NET according to following table:

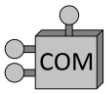
COM type (VARIANT)	.NET type (Object)
VT_EMPTY	System.Object (null reference)
VT_NULL	System.DBNull (singleton class)
VT_I2	System.Int16
VT_I4	System.Int32
VT_R4	System.Single
VT_R8	System.Double
VT_CY	System.Decimal
VT_DATE	System.DateTime
VT_BSTR	System.String
VT_DISPATCH	System.Object (not tested)
VT_ERROR	System.Int32
VT_BOOL	System.Boolean
VT_VARIANT	converted type of the target VARIANT

VT_DECIMAL	System.Decimal
VT_I1	System.Int16
VT_UI1	System.Byte
VT_UI2	System.Int32
VT_UI4	System.Int64
VT_I8	System.Int64
VT_UI8	System.Decimal
VT_INT	System.Int32
VT_UINT	System.Int64
VT_ARRAY | *vtElement*	System.Array of the converted *vtElement* type

Types that are highlighted do not convert from COM to their "natural" .NET counterparts, because the corresponding .NET type is not CLS compliant. Instead, a "wider" type that is CLS compliant is chosen.

Types not listed in the above table at all are not supported.

Strings are internally represented in Unicode wherever possible.



QuickOPC-COM is meant to be used from applications based on COM Automation, and in general, any valid VARIANT can be processed by such application. Some automation tools and programming languages, however, have restrictions on types of data they can process. If your tool does not support the data type that the OPC server is using, without QuickOPC, you would be out of luck.

In order to provide the ability to work with widest range of OPC servers and the data types they use, QuickOPC-COM converts some data types available in OPC. We have made a research into the data types supported by various tools, and QuickOPC-COM uses a subset of VARIANT types that is guaranteed to work in most tools that are in use today (one of the most restrictive languages appears to be VBScript).

Note that the QuickOPC-COM only converts the data that it passes to your application – either in output arguments of property accessors or methods, or input arguments in event notifications. In the opposite direction, i.e. for data that your application passes to QuickOPC-COM, we use very "relaxed" approach, and accept the widest range of possible data types.

QuickOPC-COM converts data from OPC Data Access according to following table:

VARTYPE in OPC Data Access	**VARTYPE In QuickOPC-COM**
VT_EMPTY	VT_EMPTY
VT_NULL	VT_NULL

VT_I2	VT_I2
VT_I4	VT_I4
VT_R4	VT_R4
VT_R8	VT_R8
VT_CY	VT_CY
VT_DATE	VT_DATE
VT_BSTR	VT_BSTR
VT_DISPATCH	VT_DISPATCH
VT_ERROR	VT_R8
VT_BOOL	VT_BOOL
VT_VARIANT	VT_VARIANT
VT_DECIMAL	VT_DECIMAL
VT_I1	VT_I2
VT_UI1	VT_UI1
VT_UI2	VT_I4
VT_UI4	VT_R8
VT_I8	VT_R8 (may lose precision)
VT_UI8	VT_R8 (may lose precision)
VT_INT	VT_I4
VT_UINT	VT_R8
VT_ARRAY | *vtElement*	VT_ARRAY | *vtElement*

Types that are highlighted are converted to a different data type. If a precise match does not exist, a "wider" type is chosen.

Types not listed in the above table at all are not supported.

Strings are internally represented in Unicode wherever possible.

Multithreading and Synchronization

The **EasyDAClient** and **EasyAEClient** objects and all their related helper objects are thread-safe.

In QuickOPC.NET, objects in the **OpcLabs.EasyOpc.DataAccess.Forms** namespace follow the general Windows Forms rules and conventions, meaning that any public **static** (**Shared** in Visual Basic) type members are thread-safe. Any instance members are not guaranteed to be thread safe.

In QuickOPC.NET, if you are hooking to the event notifications provided by the **EasyDAClient** or **EasyAEClient** component, or processing these notifications in your callback methods, make sure that you understand how the component generates these events and what threading issues it may involve. This is how it works:

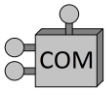
The **EasyDAClient** or **EasyAEClient** object has a **SynchronizationContext** property. This property can either be set to a null reference, or to an instance of **SystemThreading.SynchronizationContext** class. When the **SynchronizationContext** is set to a null reference, **EasyDAClient** or **EasyAEClient** calls any event handlers or callback methods on its own internal thread, which is different from any threads that you have in your application. When the **SynchronizationContext** is set to a concrete instance, the synchronization model implemented by this object is used. The **EasyDAClient** or **EasyAEClient** then typically uses the **Post** method of this synchronization context to invoke event handlers in your application.

When the **EasyDAClient** or **EasyAEClient** object is created, it attempts to obtain the synchronization context from the current thread (the thread that is executing the constructor). As a result, if the thread constructing the **EasyDAClient** or **EasyAEClient** object has a synchronization context associated with it, it will become the value of the **SynchronizationContext** property. Otherwise, the **SynchronizationContext** property will be set to a null reference. This way, the synchronization context propagates from the calling thread.

Access to Windows Forms controls is not inherently thread safe. If you have two or more threads manipulating the state of a control, it is possible to force the control into an inconsistent state. Other thread-related bugs are also possible, such as race conditions and deadlocks. It is important to make sure that access to your controls is performed in a thread-safe way. Thanks to the mechanism described above, this is done automatically for you, provided that the constructor **EasyDAClient** or **EasyAEClient** object is called on the form's main thread, as is the case if you place the **EasyDAClient** or **EasyAEClient** component on the form's design surface in Visual Studio. This works because by default, Windows Forms sets the synchronization context of the form's main thread to a properly initialized instance of **System.Windows.Forms.WindowsFormsSynchronizationContext** object.

Similarly, Windows Presentation Foundation (WPF) applications use **System.Windows.Threading.DispatcherSynchronizationContext** to achieve the same thing.

If your application is not based on the above frameworks, or is using them in an unusual way, you may have to take care of the synchronization issues related to event notification yourself, either by directly coding the synchronization mechanism, or by implementing and using a class derived from **SystemThreading.SynchronizationContext.**



In QuickOPC-COM, if you are hooking to the event notifications provided by the **EasyDAClient** or **EasyAEClient** component, make sure that you understand how the

component generates these events and what threading issues it may involve. The event notifications generated by **EasyDAClient** or **EasyAEClient** object originate from a thread that may be (and generally is) different from the thread that you used to create an instance of the object or call its methods. The code in your event handler must be prepared to deal with it.

A typical issue that arises is that access to Windows controls is not inherently thread-safe, and should be done from a dedicated thread only. It is important to make sure that access to your controls is performed in a thread-safe way. This typically involves setting up some communication mechanism between the event handler code, and a thread dedicated to handling the user interface of your application.

64-bit Platforms

You can create 32-bit or 64-bit, or platform-independent applications with QuickOPC.NET. 32-bit applications can also run on 64-bit systems. 64-bit applications can only run on 64-bit systems. Normally, you will target your application to “Any CPU”, and the same code will then run on both x86 and x64 platforms.

Supported platforms are x86 (i.e. 32-bit), and x64 (i.e. 64-bit). The product has not been tested and is not supported on Itanium platform (IA-64).

32-bit and 64-bit Code

QuickOPC.NET assemblies contain certain parts in native 32-bit code (for x86 platform) and in native 64-bit code (for x64 platform). QuickOPC.NET uses a special technique to merge the so-called mixed mode assemblies (assemblies that contain both managed and native code) for multiple platforms into a single set of assemblies.

Any application built with QuickOPC.NET assemblies can also be run on 32-bit Windows, or on 64-bit Windows for x64 processors. By default, such applications run as 32-bit processes or 32-bit machines and as 64-bit processes on 64-bit machines. You can also build your code specifically for x86 or x64 platform, if you have such need.

OPC on 64-bit Systems

Classic OPC is based on Microsoft COM/DCOM, which has originally been designed for 32-bit world, and later ported to and enhanced for 64-bit systems. There are several issues with COM/DCOM on 64-bit systems and some additional issues specific to OPC.

The most notable issue is the fact that browsing for OPC servers does not always fully work between 32-bit and 64-bit worlds. This is because the OPCEnum component (provided by OPC Foundation) runs in 32-bit process and only enumerates 32-bit OPC

servers. Consequently, native 64-bit OPC servers may be “invisible” for browsing from 32-bit OPC clients, although it is possible to connect to them, provided that the OPC client has OPC server’s ProgID or CLSID.

Version Isolation

Product versions that differ in major version number or the first digit after decimal point can be installed on the same computer in parallel. For example, version 5.12 can be installed together with version 5.02 or even version 3.03. Product versions that differ only in second digit after decimal point are designed to be replaceable, i.e. cannot be installed on the same computer simultaneously. For example, version 5.02 replaces version 5.01 or version 5.00, and version 5.12 replaces versions 5.10 and 5.11.

The simulation OPC server is not subject to the versioning rules described above for the QuickOPC product. Just one instance of simulation OPC server can exist on a computer. Features are being added to the simulation OPC server with newer versions of QuickOPC. Always install the simulation OPC server from the latest QuickOPC version in order to guarantee that all examples are functional.

Additional Resources

If you are migrating from earlier version, please read the “What’s New” document (available from Start menu).

Study the Reference documentation (also available from Start menu).

Explore the examples and bonus tools and materials installed with the product.

You may have a look at “OPC Foundation Whitepapers” folder under the documentation group in Start menu. We have included a selection of OPC Foundation White Papers that you may find useful while getting accustomed with OPC in general, or dealing with its specific aspects. Please pay particular attention to document titled “Using OPC via DCOM with Windows XP Service Pack 2”, as it contains useful hints that apply not only to Windows XP SP2 users.

Check the vendor’s Web page for updates, news, related products and other information.