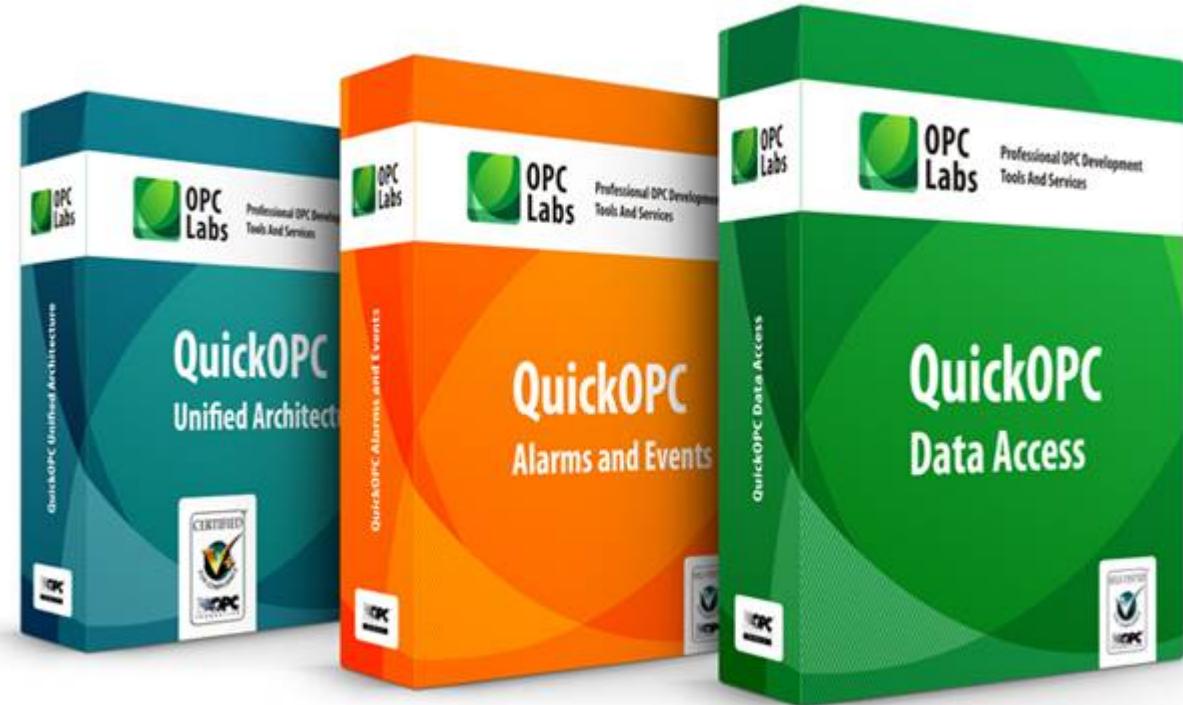


# QuickOPC User's Guide



## QuickOPC User's Guide

Version 2016.2



## Table of Contents

1.	Introduction	15
1.1.	Terminology	15-16
1.2.	Development Tools	16
2.	Getting Started	17
2.1.	Getting Started with OPC Classic under .NET	17
2.1.1.	Making a first OPC Classic application using Live Binding	17-18
2.1.2.	Making a first OPC Classic application using traditional coding	18-20
2.1.3.	Where do I go from here?	20
2.2.	Getting Started with OPC UA under .NET	20
2.2.1.	Making a first OPC UA application using Live Binding	20-21
2.2.2.	Making a first OPC UA application using traditional coding	21-22
2.2.3.	Where do I go from here?	22-23
2.3.	Getting Started with OPC Classic under COM	23
2.3.1.	Making a first COM application	23-24
2.3.2.	Where do I go from here?	24
2.4.	Getting Started with OPC UA under COM	24
2.4.1.	Making a first UA application	24-26
2.4.2.	Where do I go from here?	26
3.	Installation	27
3.1.	Common Concerns	27
3.1.1.	Operating Systems	27
3.1.2.	Hardware	27
3.1.3.	Included Software	27
3.1.4.	Prerequisites	27-28
3.1.5.	Licensing	28
3.1.6.	Related Products	28
3.2.	Setup Program	28
3.2.1.	Running the Setup	28-30
3.2.2.	Uninstallation	30-31
3.2.3.	Troubleshooting the Setup	31
3.3.	NuGet Packages	31-32
3.3.1.	List of Packages	32
3.3.2.	Using Package Manager GUI	32
3.3.3.	Using Package Manager Console	32

# QuickOPC User's Guide

2

4.	Product Parts	33
4.1.	The Launcher application	33
4.2.	Assemblies	33-35
4.2.1.	XML Comments	35
4.2.2.	ReSharper Annotations	35
4.2.3.	Code Contracts	35
4.3.	COM Components	35-36
4.4.	Development Libraries (COM)	36-37
4.5.	Demo Applications	37
4.5.1.	.NET Demo Applications	37
4.5.1.1.	.NET Demo Applications for OPC Classic	37-38
4.5.1.2.	.NET Demo Applications for OPC UA	38-40
4.5.2.	COM Demo Applications	40-41
4.6.	Examples	41
4.6.1.	LINQPad support	41
4.7.	Test Servers	41
4.7.1.	Simulation Server for OPC Classic	41-42
4.7.2.	OPC XML Sample Server	42
4.7.3.	OPC UA Sample Server	42
4.7.4.	QuickStart Alarm Condition Server (OPC UA)	42
4.8.	License Manager	43
4.8.1.	LMConsole Utility (License Manager Console)	43-44
4.9.	Documentation and Help	44
4.10.	Test Tools	44-45
5.	Fundamentals	46
5.1.	Typical Usage	46
5.1.1.	OPC Classic thick-client .NET applications on LAN	46
5.1.2.	OPC Classic or OPC UA thick-client COM applications on LAN	47
5.1.3.	OPC Classic Web applications (server side)	47-48
5.1.4.	OPC UA Thick-client applications on LAN, WAN or Internet	48-49
5.1.5.	OPC UA Web applications (server side)	49-50
5.2.	Development Models	50
5.2.1.	Procedural Coding Model	50-51
5.2.2.	Live Binding Model	51-52
5.2.3.	Live Mapping Model	52-53
5.2.4.	Reactive Programming Model	53

5.3.	Referencing the Assemblies (.NET)	53
5.3.1.	Overview of the Assemblies Available	53
5.3.2.	Manual Assembly Referencing	53-54
5.3.3.	Assembly Referencing with Visual Studio Toolbox	54
5.3.4.	Assembly Referencing with NuGet	54-55
5.3.5.	Licensing in Visual Studio	55
5.3.6.	Namespaces	55-57
5.4.	Referencing the Components (COM)	57
5.4.1.	COM Components for OPC Classic	58
5.4.2.	COM Components for OPC UA	58
5.4.3.	Importing Type Libraries to Delphi	58-62
5.5.	Conventions	62
5.5.1.	Date and Time Conventions	62
5.5.2.	Naming Conventions	62-63
5.5.2.1.	Common Naming Conventions	63
5.5.2.2.	Type Names	63
5.5.2.3.	Interface Names in COM	63
5.5.2.4.	ProgIDs (COM)	63
5.5.3.	Coloring Conventions	63-64
5.6.	Components and Objects	64-66
5.6.1.	Computational Objects	66-67
5.6.1.1.	Interfaces and Extension Methods	67-68
5.6.1.2.	Isolated Clients	68
5.6.1.3.	Shared Instance	68
5.6.2.	User Interface Objects	68
5.7.	Connection-less Approach	69
5.8.	Simultaneous Operations	69
5.9.	Event Pull Mechanism	69-80
5.10.	Error Handling	80-81
5.10.1.	Errors and Multiple-Element Operations	81-82
5.10.2.	Errors in Subscriptions	82
5.11.	Helper Types	82
5.11.1.	Time Periods	82
5.11.2.	Data Objects	82
5.11.2.1.	Quality in OPC Classic	82-83
5.11.2.2.	Value, Timestamp and Quality (VTQ) in OPC Classic	83

5.11.2.3. Variant Type (VarType) in OPC Classic	83-84
5.11.2.4. OPC UA Status Code	84
5.11.2.5. OPC UA Attribute Data	84
5.11.3. Argument Objects	84
5.11.4. Result Objects	84-85
5.11.5. Element Objects	85
5.11.6. Descriptor Objects	85-86
5.11.7. Browse Paths	87
5.11.7.1. Browse Paths for OPC Classic	87
5.11.7.2. OPC Classic Browse Path Format	87
5.11.7.3. Browse Paths in OPC-UA	87-88
5.11.7.4. OPC-UA Browse Path Format	88-90
5.11.8. Parameter Objects	90-91
5.11.9. Utility Classes	91
5.12. Identifying information in OPC XML	91
5.12.1. Servers	91-92
5.12.2. Nodes and Items	92
5.12.3. Properties	92
5.13. Identifying Information in OPC-UA	92
5.13.1. Server Endpoints	92-93
5.13.2. Node IDs	93-94
5.13.2.1. Namespace indices in Node Ids	94
5.13.2.2. Standard Node IDs	94
5.13.3. Qualified Names	94
5.13.4. Attribute IDs	95
5.13.5. Index Range Lists	95-96
5.14. OPC-UA Security	97
5.14.1. Security in Endpoint Selection	97
5.14.2. Trusting Server Instance Certificate	97-98
5.14.3. Providing Client Instance Certificate	98-101
6. Procedural Coding Model	102
6.1. Procedural Coding Model for OPC Data (Classic and UA)	102
6.1.1. Obtaining Information	102
6.1.1.1. Reading from OPC Classic Items	102-107
6.1.1.1.1. Reading just the value	107-110

6.1.1.1.2. Reading in OPC XML-DA	110-111
6.1.1.2. Getting OPC Classic Property Values	111-117
6.1.1.3. Reading Attributes of OPC UA Nodes	117-128
6.1.1.3.1. Reading just the value	128-137
6.1.2. Modifying Information	137
6.1.2.1. Writing to OPC Classic Items	137-141
6.1.2.1.1. Writing value, timestamp and quality	141-143
6.1.2.2. Writing Attributes of OPC UA Nodes	143-153
6.1.2.2.1. Data Type in OPC UA Write	153
6.1.2.2.2. Writing value, timestamps and status code	153-159
6.1.3. Browsing for Information	159-160
6.1.3.1. Browsing for OPC Classic Servers	160-161
6.1.3.2. Browsing for OPC Classic Nodes (Branches and Leaves)	161-164
6.1.3.3. Browsing for OPC Classic Access Paths	164-165
6.1.3.4. Browsing for OPC Classic Properties	165-166
6.1.3.5. Discovering OPC UA Servers	166
6.1.3.5.1. OPC UA Local Discovery	166-172
6.1.3.5.2. OPC UA Global Discovery	172-173
6.1.3.5.3. Generalized OPC UA Discovery	173
6.1.3.6. Browsing for OPC UA Nodes	173-180
6.1.4. Subscribing to Information	180
6.1.4.1. Subscribing to OPC Classic Items	180-186
6.1.4.2. Subscribing to OPC UA Monitored Items	186-200
6.1.4.3. Changing Existing Subscription	200-209
6.1.4.4. Unsubscribing	209-221
6.1.4.5. Obtaining Subscription Information	221-227
6.1.4.6. OPC Classic Item Changed Event or Callback	227-228
6.1.4.7. OPC UA Monitored Item Changed Event or Callback	228-229
6.1.4.8. Using Callback Methods Instead of Event Handlers	229-231
6.1.5. Calling Methods in OPC-UA	231-248
6.1.6. Setting Parameters	248-251
6.1.6.1. User Identity in QuickOPC-UA	251
6.1.6.2. Server Diagnostics in OPC-UA	251-252
6.2. Procedural Coding Model for OPC Classic A&E	252
6.2.1. Obtaining Information	252
6.2.1.1. Getting Condition State	252-254

6.2.2. <a href="#">Modifying Information</a>	254
6.2.2.1. <a href="#">Acknowledging a Condition</a>	254-258
6.2.3. <a href="#">Browsing for Information</a>	258
6.2.3.1. <a href="#">Browsing for OPC Servers</a>	258-260
6.2.3.2. <a href="#">Browsing for OPC Nodes (Areas and Sources)</a>	260-263
6.2.3.3. <a href="#">Querying for OPC Event Categories</a>	263-266
6.2.3.4. <a href="#">Querying for OPC Event Conditions on a Category</a>	266
6.2.3.5. <a href="#">Querying for OPC Event Conditions on a Source</a>	266-268
6.2.3.6. <a href="#">Querying for OPC Event Attributes</a>	268
6.2.4. <a href="#">Subscribing to Information</a>	268
6.2.4.1. <a href="#">Subscribing to OPC Events</a>	268-272
6.2.4.2. <a href="#">Specifying event filters</a>	272-276
6.2.4.3. <a href="#">Changing Existing Subscription</a>	276-279
6.2.4.4. <a href="#">Unsubscribing from OPC Events</a>	279-283
6.2.4.5. <a href="#">Refreshing Condition States</a>	283-285
6.2.4.6. <a href="#">Notification Event</a>	285-290
6.2.4.7. <a href="#">Using Callback Methods Instead of Event Handlers</a>	290-291
6.2.5. <a href="#">Setting Parameters</a>	291
6.3. <a href="#">Procedural Coding Model for OPC UA Alarms &amp; Conditions</a>	291-292
6.3.1. <a href="#">Obtaining Information</a>	292
6.3.2. <a href="#">Modifying Information</a>	292
6.3.2.1. <a href="#">Conditions - Disabling/Enabling, and Applying Comments</a>	292
6.3.2.2. <a href="#">Dialogs - Responding</a>	292
6.3.2.3. <a href="#">Acknowledgeable Conditions – Acknowledging and Confirmation</a>	292-294
6.3.2.4. <a href="#">Alarms – Shelving and Unshelving</a>	294
6.3.3. <a href="#">Browsing for Information</a>	294
6.3.3.1. <a href="#">Discovering OPC UA Servers</a>	294-295
6.3.3.2. <a href="#">Browsing for Event Sources</a>	295
6.3.3.3. <a href="#">Browsing for Notifiers</a>	295
6.3.4. <a href="#">Subscribing to Information</a>	295
6.3.4.1. <a href="#">Subscribing to OPC Events</a>	295-298
6.3.4.2. <a href="#">Specifying Event Filters</a>	298-299
6.3.4.2.1. <a href="#">The Select clauses</a>	299-302
6.3.4.2.2. <a href="#">The Where clause</a>	302-306
6.3.4.3. <a href="#">Changing Existing Subscription</a>	306
6.3.4.4. <a href="#">Unsubscribing from OPC Events</a>	306

6.3.4.5. Refreshing Condition States	306
6.3.4.6. Notification Event	306-310
6.3.4.7. Using Callback Methods Instead of Event Handler	310-311
6.3.5. Setting Parameters	311-312
6.4. Operation Monitoring and Control	312
7. Live Mapping Model	313
7.1. Live Mapping Model for OPC Data (Classic and UA)	313
7.1.1. Live Mapping Example	313-319
7.1.2. Live Mapping Overview	319
7.1.2.1. Mapping Sources	319
7.1.2.2. Mapping Targets	319
7.1.2.3. Mappings	319-320
7.1.3. Attributes for Live Mapping	320-321
7.1.4. Propagated Parameters	321-322
7.1.5. Type Mapping	322
7.1.6. Member Mapping	322
7.1.6.1. Mapping Property and Field Members	322
7.1.6.2. Mapping Method Members	322-323
7.1.6.3. Mapping Event Members	323
7.1.6.4. Map-through, OPC Nodes and OPC Data	323-324
7.1.6.5. Browse Path and Node Id Resolution	324-326
7.1.6.6. Meta- Members	326-327
7.1.6.7. Mapping Tags	327
7.1.6.8. Deferred Mapping	327
7.1.7. Mapped Node Classes	327-328
7.1.8. Mapping Operations	328
7.1.9. Mapping Kinds	328-330
7.1.10. Mapping Arguments and Phases	330-332
7.1.11. The Mapper Object	332
7.1.11.1. Mapping Your Objects	332-333
7.1.11.2. Mapping Context	333
7.1.11.3. Invoking the Operations	333-334
7.1.11.4. Type-less Mapping	334-338
7.1.11.5. Error Handling in the Mapper Object	338-339
7.2. OPC-UA Modelling (Preliminary)	339

7.2.1. <a href="#">How It Works</a>	339-340
7.2.2. <a href="#">Node Types</a>	340-342
8. <a href="#">Live Binding Model</a>	343
8.1. <a href="#">Live Binding Model for OPC Data (Classic and UA)</a>	343
8.1.1. <a href="#">Live Binding Overview</a>	343-344
8.1.1.1. <a href="#">Value Targets</a>	344-345
8.1.1.2. <a href="#">Binding Operations</a>	345
8.1.1.3. <a href="#">Bindings</a>	345-346
8.1.1.4. <a href="#">Binding Bags</a>	346
8.1.1.5. <a href="#">Binding Extender</a>	346
8.1.1.6. <a href="#">Binders and Binding Providers</a>	346
8.1.1.7. <a href="#">Event Sources</a>	346-347
8.1.1.8. <a href="#">Binding Groups</a>	347
8.1.2. <a href="#">Point Binder</a>	347
8.1.2.1. <a href="#">Connectivities</a>	347-348
8.1.2.2. <a href="#">Points</a>	348
8.1.2.3. <a href="#">Point Editor</a>	348-350
8.1.2.4. <a href="#">Parameters</a>	350
8.1.2.5. <a href="#">Arguments and Results</a>	350-351
8.1.2.6. <a href="#">Arguments Path</a>	351-352
8.1.2.7. <a href="#">Parameter Templates</a>	352
8.1.2.8. <a href="#">Queued Execution</a>	352
8.1.3. <a href="#">Value Conversions, String Formats and Converters</a>	352-353
8.1.3.1. <a href="#">The LinearConverter Component</a>	353
8.1.3.2. <a href="#">The StatusToColorConverter Component</a>	353-354
8.1.4. <a href="#">Live Binding in the Designer</a>	354
8.1.4.1. <a href="#">Toolbox Items</a>	354
8.1.4.2. <a href="#">Extender Properties</a>	355
8.1.4.3. <a href="#">Designer Commands</a>	355
8.1.4.4. <a href="#">Binding Collection Editor</a>	355-357
8.1.4.5. <a href="#">Binding Group Collection Editor</a>	357-358
8.1.4.6. <a href="#">Online Design</a>	358
8.1.5. <a href="#">Live Binding Details</a>	358-359
8.1.5.1. <a href="#">How the Binding Extender Automatically Goes Online and Offline</a>	359
8.1.5.2. <a href="#">What Happens When the Binding Extender Is Set Online or Offline</a>	359
8.1.5.3. <a href="#">What Happens When a Binding Operation Method Is Called</a>	359-360

8.1.5.4. Programmatic Access to Live Binding	360
8.1.5.5. Error Handling in Live Binding	360-361
8.1.5.6. Usage Guidelines	361
8.1.6. Typical Binding Scenarios	361-362
8.1.6.1. Automatic Subscription (With the Form)	362
8.1.6.2. Automatic Read (On Form Load)	362
8.1.6.3. Read On Custom Event	362-363
8.1.6.4. String Formatting	363
8.1.6.5. Change Color According to Status	363-364
8.1.6.6. ToolTip and Other Extenders	364
8.1.6.7. Display Errors with ErrorProvider	364
8.1.6.8. Various Kinds of Binding	364-365
8.1.6.9. Write Single Value on Custom Event	365
8.1.6.10. Write Group of Values on Custom Event	365
8.1.6.11. Subscribe & Write	365-366
8.1.6.12. Display Write Errors	366
8.1.6.13. Get an OPC Property on Custom Event	366
8.1.6.14. Automatically Get an OPC Property (On Form Load)	366
8.1.7. Obsolete Live Binding Features	367
8.1.7.1. Documentation for Obsoleted Features	367
8.1.7.2. Old Project Conversion	367
9. Reactive Programming Model	368
9.1. Reactive Programming Model for OPC Data (Classic and UA)	368
9.1.1. OPC Reactive Extensions (Rx/OPC)	368-369
9.1.2. OPC Data Observables	369-370
9.1.3. OPC Data Observers	370
9.2. Reactive Programming Model for OPC Alarms and Events	370-371
9.2.1. OPC Reactive Extensions (Rx/OPC)	371
9.2.2. OPC-A&E Observables	371
9.2.3. OPC-A&E Observers	371
10. User Interface	372
10.1. OPC Common Dialogs	372-373
10.1.1. OPC-DA Common Dialogs	373
10.1.1.1. Computer Browser Dialog	373-376
10.1.1.2. OPC Server Dialog	376-377
10.1.1.3. OPC Computer and Server Dialog	377

10.1.1.4. <a href="#">OPC-DA Item Dialog</a>	377-380
10.1.1.5. <a href="#">OPC-DA Property Dialog</a>	380-381
10.1.1.6. <a href="#">Generic OPC Browsing Dialog</a>	381-384
10.1.2. <a href="#">OPC-A&amp;E Common Dialogs</a>	384
10.1.2.1. <a href="#">Computer Browser Dialog</a>	384-385
10.1.2.2. <a href="#">OPC Server Dialog</a>	385
10.1.2.3. <a href="#">OPC Computer and Server Dialog</a>	385
10.1.2.4. <a href="#">OPC-A&amp;E Area or Source Dialog</a>	385-386
10.1.2.5. <a href="#">OPC-A&amp;E Category Dialog</a>	386-387
10.1.2.6. <a href="#">OPC-A&amp;E Category Condition Dialog</a>	387-388
10.1.2.7. <a href="#">OPC-A&amp;E Attribute Dialog</a>	388-389
10.1.2.8. <a href="#">Generic OPC Browsing Dialog</a>	389
10.1.3. <a href="#">OPC-UA Common Dialogs</a>	389
10.1.3.1. <a href="#">OPC-UA Endpoint Dialog</a>	389-393
10.1.3.2. <a href="#">OPC-UA Host and Endpoint Dialog</a>	393-396
10.1.3.3. <a href="#">OPC-UA Data Dialog</a>	396-400
10.1.3.4. <a href="#">Generic OPC-UA Browsing Dialog</a>	400-406
10.2. <a href="#">OPC Controls</a>	406
10.2.1. <a href="#">OPC Classic Controls</a>	406-407
10.2.1.1. <a href="#">Generic OPC Classic Browsing Control</a>	407-408
10.2.2. <a href="#">OPC-UA Controls</a>	408
10.2.2.1. <a href="#">Generic OPC UA Browsing Control</a>	408-410
10.3. <a href="#">Common Functionality in Browsing Dialogs and Controls</a>	410-411
11. <a href="#">EasyOPC Extensions for .NET</a>	412
11.1. <a href="#">Usage</a>	412
11.1.1. <a href="#">Generic Types</a>	412
11.2. <a href="#">Data Access Extensions</a>	412
11.2.1. <a href="#">Generic Types for OPC-DA</a>	412-413
11.2.2. <a href="#">Extensions for OPC Items</a>	413
11.2.2.1. <a href="#">Type-safe Access</a>	413
11.2.2.2. <a href="#">Generic Access</a>	413
11.2.3. <a href="#">Extensions for OPC Properties</a>	413
11.2.3.1. <a href="#">Type-safe Access</a>	413
11.2.3.2. <a href="#">Generic Access</a>	414
11.2.3.3. <a href="#">Well-known Properties</a>	414-415

11.2.3.4. Alternate Access Methods	415-417
11.3. Alarms&Events Extensions	417
11.3.1. OPC-A&E Queries	417
11.4. Unified Architecture Extensions	418
11.4.1. Extensions on Helper Types	418
11.5. Software Toolbox Extender Replacement	418
11.5.1. Introduction	418
11.5.2. Details	418
11.5.3. Assemblies and Deployment Elements	418-419
12. Application Deployment	420
12.1. Deployment Elements	420
12.1.1. Assemblies (.NET)	420
12.1.2. COM Components and Development Libraries	420
12.1.3. Prerequisites	421
12.1.4. Licensing	421
12.2. Deployment Methods	421
12.2.1. Manual Deployment	421
12.2.2. Automated deployment, roll your own	421-422
12.2.2.1. Direct License Deployment	422
12.2.2.1.1. License Store	422-423
12.2.2.1.2. Deployment Automation	423
12.2.3. Automated deployment with Production installer	423-424
13. Advanced Topics	425
13.1. OPC Specifications	425
13.2. OPC Interoperability	425-426
13.2.1. OPC-UA via UA Proxy	426-427
13.2.2. OPC "Classic" via UA Wrapper	427
13.3. Application Configuration	427-429
13.4. Event Logging	429
13.4.1. Event Logging for OPC "Classic"	429
13.4.2. Event Logging for OPC UA	429-434
13.5. Debugging	434
13.6. Object Serialization	434-435
13.7. Internal Optimizations	435
13.8. Failure Recovery	435-436
13.9. Timeout Handling	436-437

13.10. Data Types	437
13.10.1. Data Types in OPC Classic	437-439
13.10.2. Data Types in OPC-UA	439-440
13.11. Multithreading and Synchronization	440-441
13.12. 64-bit Platforms	441-442
13.12.1. 32-bit and 64-bit Code	442
13.12.2. Classic OPC on 64-bit Systems	442
13.13. Prerequisites Boxing	442
13.14. Version Isolation	442-443
14. Best Practices	444
14.1. Do not write any code for OPC failure recovery	444
14.2. With single-operand synchronous methods, only catch OpcException or UAException	444
14.3. Do not catch any exceptions with asynchronous or multiple-operand methods	444-445
14.4. Always test the Exception property before accessing the actual result (Value, Vtq, or AttributeData property)	445
14.5. Always test the HasValue property before accessing DAVtq.Value or UAAttributeData.Value	445
14.6. Use multiple-operand methods instead of looping	445
14.7. Do not block inside OPC event handlers or callback methods	445-446
14.8. Use generic or type-safe access wherever possible	446
14.9. Use the state instead of handles to identify subscribed entities	446-447
15. Additional Resources	448
16. Options	449
16.1. StreamInsight Option	449
16.1.1. Introduction	449
16.1.2. Installation and Getting Started	449-450
16.1.2.1. Licensing	450
16.1.2.2. Installing a StreamInsight Instance	450-451
16.1.2.3. Building and Running a First StreamInsight Application with OPC	451-452
16.1.2.4. Debugging the StreamInsight Event Flow	452-455
16.1.3. Fundamentals	455
16.1.3.1. Product Parts	455
16.1.3.1.1. Assemblies	455
16.1.3.1.2. Examples	455
16.1.3.1.3. Documentation and Help	455-456
16.1.3.2. Usage	456
16.1.3.2.1. How it Works	456

16.1.3.2.2. Referencing the Assemblies	456-457
16.1.3.2.3. Example Walkthrough	457-459
16.1.3.3. Creating OPC Event Sources	459
16.1.3.3.1. OPC Observables	459-460
16.1.3.3.2. Errors in OPC Sequences	460
16.1.3.3.3. Other Special Cases in OPC Sequences	460
16.1.3.4. OPC Event Payloads	460-461
16.1.3.4.1. Common Payload Characteristics	461
16.1.3.4.2. Payloads for OPC Data Access	461-462
16.1.3.4.3. Payloads for OPC Alarms and Events	462-463
16.1.3.4.4. Payloads for OPC Unified Architecture	463-464
16.1.3.5. Time in OPC and StreamInsight	464-465
16.1.3.5.1. Time Synchronization in OPC	465
16.1.3.5.2. Time in OPC Subscriptions	465-466
16.1.3.5.3. Conclusions	466
16.1.4. Examples	466
16.1.4.1. List of Examples	466-467
16.1.5. Application Deployment	467
16.1.6. Additional Resources	467
17. Examples	468
17.1. .NET Examples	468
17.1.1. Examples for OPC "Classic" (OPC-DA, OPC XML-DA and OPC-A&E)	468-471
17.1.2. Examples for OPC Unified Architecture (OPC-UA)	471-472
17.1.3. Integration Examples	472-473
17.1.4. Reactive Programming Examples	473-474
17.1.5. LINQPad Examples	474
17.2. COM Examples	474-475
17.2.1. Free Pascal Examples (Lazarus)	475
17.2.2. JScript Examples (IE, WSH)	475-476
17.2.3. Object Pascal Examples (Delphi)	476
17.2.4. Perl Examples	476
17.2.5. PHP Examples	476-477
17.2.6. Portable C++ Examples	477-478
17.2.7. Python Examples	478
17.2.8. REALbasic (Xojo) Examples	478
17.2.9. VBA Examples in Excel	478

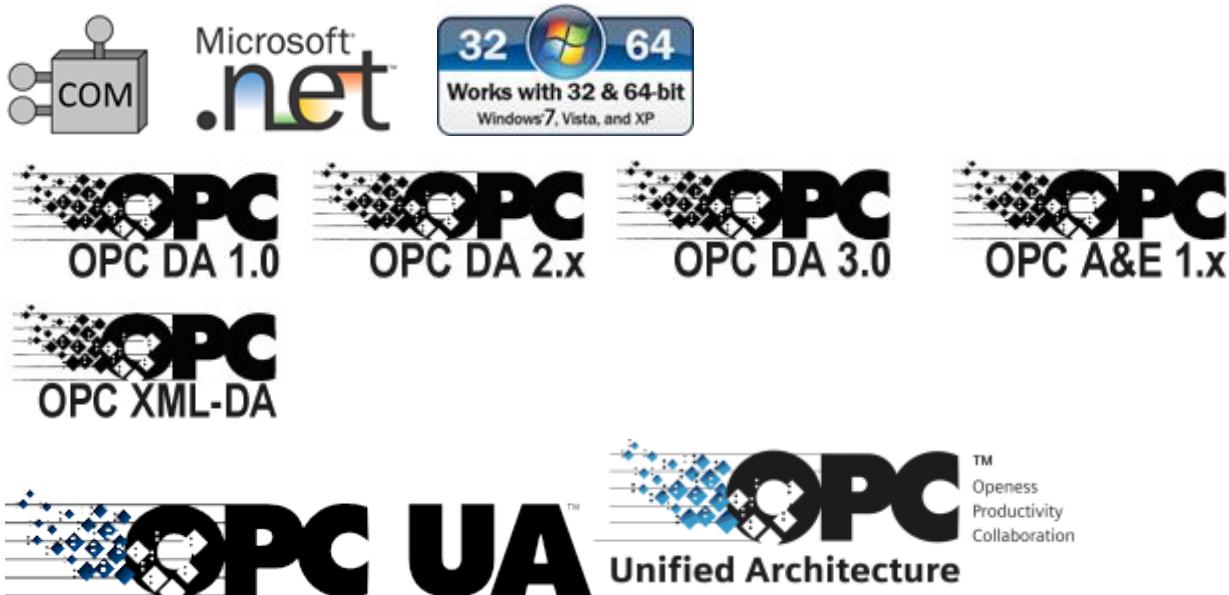
17.2.10. <a href="#">VBScript Examples (ASP, IE, WSH)</a>	478-479
17.2.11. <a href="#">Visual Basic Examples (VB 6.0)</a>	479
17.2.12. <a href="#">Visual C++ Examples</a>	479-480
17.2.13. <a href="#">Visual FoxPro Examples</a>	480
17.2.14. <a href="#">Xbase++ Examples</a>	480
18. <a href="#">Reference</a>	481
18.1. <a href="#">.NET Assemblies</a>	481
18.2. <a href="#">COM Type Libraries</a>	481-483
18.3. <a href="#">Format Strings</a>	483
19. <a href="#">Index</a>	484-522

## 1 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, or OPC Foundation SDKs. In OPC "Classic" world (COM/DCOM based), 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. Other hurdles exist in OPC Unified Architecture (OPC-UA), such as proper server endpoint discovery, security policy negotiation, etc. 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 an OPC server, writing a data value, or subscribing to data changes can be achieved in just one or two lines of code! Receiving alarms from OPC Alarms and Events server is also easy.



Notes:

- This documentation is for QuickOPC version 2016.2. This is known as the *external version* of the product. The *internal version* number is 5.41.

### 1.1 Terminology

The components can be used from various languages and environments.

QuickOPC-Classic is a product line that consists of two parts: QuickOPC.NET and QuickOPC-COM. The text in this document applies mostly to both parts. Similarly, the QuickOPC-UA also supports both .NET and COM targets, however there are no "special" simple names for these products parts; we will refer to them as "QuickOPC-UA for .NET" and "QuickOPC-UA for COM" when necessary.

When necessary, the differing text is marked with corresponding COM or .NET icon.

The following table explains the product names, and how they relate to OPC specifications they cover, and to the

development tools platform that can be used with them.

<b>Development Tools/Target Platform:</b>	 Microsoft .NET	 Microsoft COM
<b>OPC Specifications</b>		
OPC "Classic" (DCOM based)	OPC Data Access (OPC-DA) OPC Alarms and Events (OPC-A&E)	QuickOPC.NET
OPC XML-DA (XML Data Access)		QuickOPC-COM
OPC Unified Architecture (OPC-UA), for Data Access, Alarms & Conditions	QuickOPC-UA (for .NET)	QuickOPC-UA for COM

The OPC XML-DA support in QuickOPC.NET is transparently integrated into the component. Wherever this document refers to "OPC Data Access", it also usually applies to OPC XML-DA.

All products/product parts are delivered in a single installation package.

## 1.2 Development Tools

In QuickOPC.NET and QuickOPC-UA for .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 the commercial developer editions of Microsoft Visual Studio 2012, 2013 and 2015. We do not officially target or support the non-commercial editions, such as Visual Studio Community (they may work though... it's up to you to figure out, if you are interested).

The .NET projects you create need to target .NET Framework 4.5.2 or higher.

If you use Visual Studio 2012 or Visual Studio 2013, you will need to install the targeting pack for .NET Framework 4.5.2 (or higher). The targeting packs for .NET Framework 4.5.2 and .NET Framework 4.6 come preinstalled with Visual Studio 2015.



In QuickOPC-COM and QuickOPC-UA for 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, Object Pascal (in Delphi), 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.

## 2 Getting Started

This section of the documentation consists of following parts:

- **Getting Started with OPC Classic under .NET (Section 2.1),**
- **Getting Started with OPC UA under .NET (Section 2.2),**
- **Getting Started with OPC Classic under COM (Section 2.4), and**
- **Getting Started with OPC UA under COM (Section 2.4).**

Depending on whether you develop OPC "Classic" applications for Microsoft .NET, or using Microsoft COM, or application for OPC Unified Architecture, please refer to the corresponding part.

### 2.1 Getting Started with OPC Classic under .NET

QuickOPC.NET allows you to develop using various development models. In this Getting Started chapter, we will present two of them:

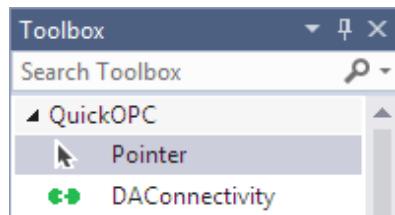
- **Making a first QuickOPC.NET application using Live Binding (Section 2.1.1)** (without any coding), and
- **Making a first QuickOPC.NET application using traditional coding (Section 2.1.2)** (procedural).

Other development models available are *live mapping*, and *reactive programming (Rx)*.

#### 2.1.1 Making a first OPC Classic application using Live Binding

In this Getting Started procedure, we will create a Windows Forms application in C# or Visual Basic that subscribes to OPC Data Access (OPC-DA) item and continuously displays its changes. The live binding model allows achieving this functionality by simply configuring the QuickOPC components, without any manual coding.

1. Install QuickOPC.NET. If you are reading this text, chances are that you have already done this.
2. Start Microsoft Visual Studio 2012, 2013 or 2015, and create new Visual C# or Visual Basic project, selecting "Windows Forms Application" template. **Make sure that in the upper part of the "New Project" dialog, the target framework is set to ".NET Framework 4.5.2" or later.**
3. Drag the DAConnectivity component from the "QuickOPC" tab of the Toolbox to the form's design surface.



Notice that three icons will appear below the form: One labeled "daConnectivity1", the second labeled "pointBinder1", and the third labeled "bindingExtender1".

If the needed components do not show in the Toolbox, right-click in the Toolbox area, select "Choose Items...", enter **opclabs** into the Filter box, check all boxes next to the displayed components, and press OK.

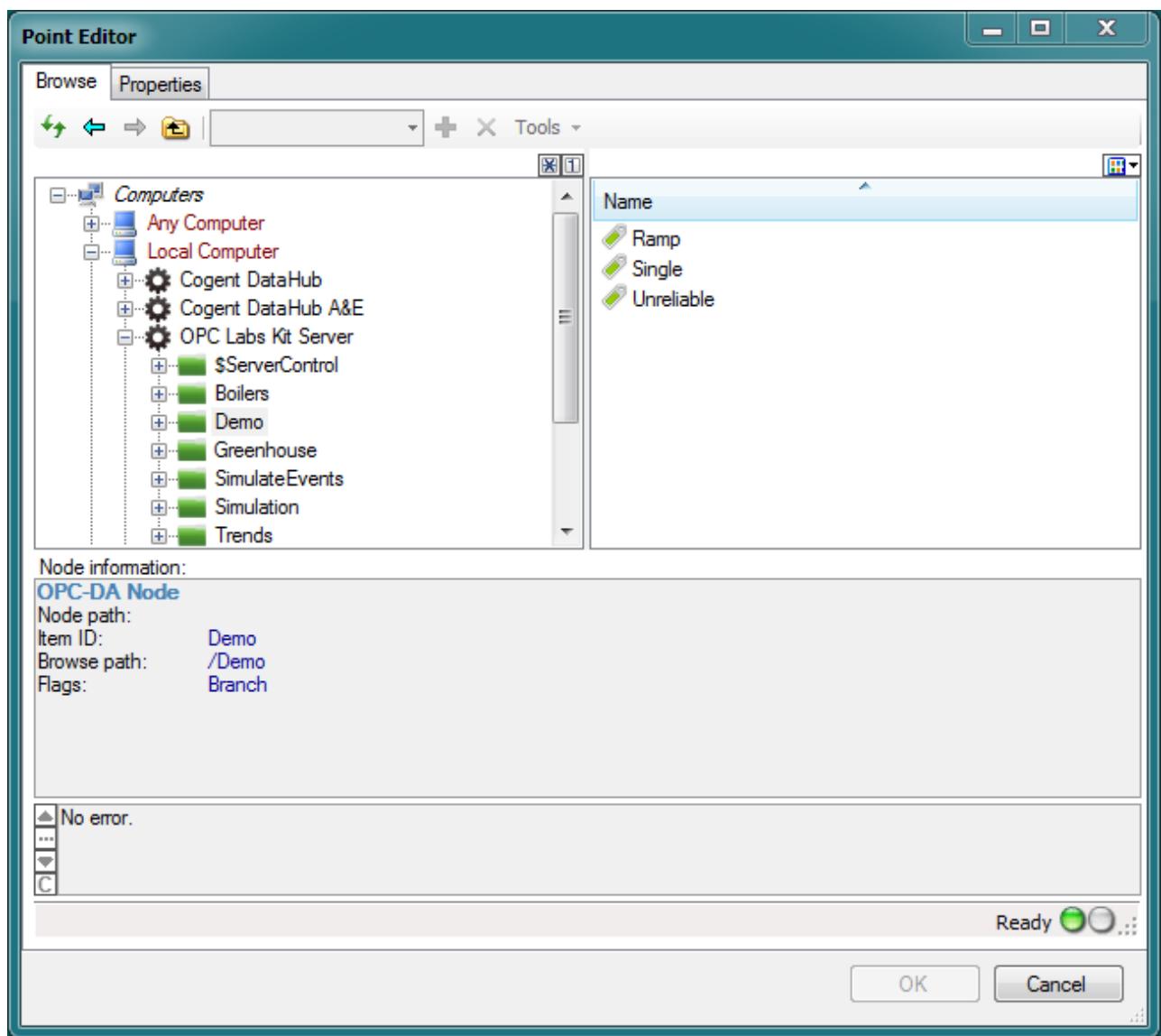
4. Drag a standard TextBox from the Toolbox to the form.
5. Right-click on the text box control on the form, and choose "Bind to Point" command.  
Alternatively, you can select the text box control first, and then choose the "Bind to Point" command in the

Properties window; for this to work, however, the commands must be displayed in the Properties window.

[Bind to Point...](#), [Edit Live Bindings...](#)  
[Remove Live Bindings](#)

If you do not see these commands, display them by right-clicking in the Properties window, and checking "Commands" in the menu.

6. In the "Point Editor" dialog, select Computers -> Local Computer -> OPC Labs Kit Server -> Demo -> Ramp, and press OK.



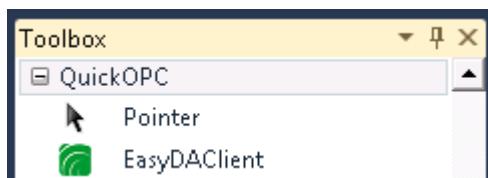
7. Build and run your application. You will see live OPC data changing in the text box. It is also possible to verify the effects of live binding without building the application: On the **bindingExtender** component placed on the form, select the "Design Online" command. This will cause Visual Studio to perform the live binding immediately. Use the "Design Offline" command to revert back to normal mode.

## 2.1.2 Making a first OPC Classic application using traditional coding

If you are interested in QuickOPC.NET, your task will most certainly involve reading data from an OPC server. Here are a few steps that illustrate how to achieve that, using Microsoft Visual Studio 2012, 2013 or 2015, and a simple Windows Forms application in C#. The steps are quite similar if you are using other tools or a different programming language.

We will create a form that will read a float value from the OPC server when loaded, and immediately display the formatted value in the text box on the form.

1. Install QuickOPC.NET. If you are reading this text, chances are that you have already done this.
2. Start Microsoft Visual Studio 2012, 2013 or 2015, and create new Visual C# project, selecting "Windows Forms Application" template. **Make sure that in the upper part of the "New Project" dialog, the target framework is set to ".NET Framework 4.5.2" or later.**
3. Instantiate the EasyDAClient component in the form: Drag the EasyDAClient component from the "QuickOPC" tab of the Toolbox to the form's design surface.



An icon labeled "easyDAClient1" will appear below the form.

If the needed components do not show in the Toolbox, right-click in the Toolbox area, select "Choose Items...", enter **opclabs** into the Filter box, check all boxes next to the displayed components, and press OK.

Note: This is just shortcut way in Windows Forms to declaring and instantiating the component. In other environments, you can instantiate the component using the 'new' (C#) or 'New' (VB.NET) keyword, or any other way provided by the language or tool you are using.

4. Add a textbox to the form: Drag the TextBox component from the Toolbox to the form's design surface. The text box should appear on the form, and in the Properties window, you will see that it has been given a "textBox1" name.

5. Add handler for the Load event of the form: Select the form by clicking on its design surface. Then, in the Properties window, choose the "Events" view (orange lightning icon), and double-click the line with "Load" event (it is under Behavior group). A "Form1\_Load" text in bold font will appear next to the event name, and a new event handler will be created for the Load event. The text editor will open for the code, and the caret will be placed into the Form1\_Load method.

6. Add following code to the beginning of the Form1.cs file:

```
using OpcLabs.EasyOpc.DataAccess;
```

7. Write the event handler implementation. Add the following code to the body of Form1\_Load method:

```
textBox1.Text = easyDAClient1.ReadItemValue("", "OPCLabs.KitServer.2",  
"Demo.Single").ToString();
```

8. Build the project.

9. Run the application and observe the results. The text box on the form will be filled with a float value we have read from the OPC server, using a single code line!

If you are targeting an environment different from Windows Forms (such as console application, Web, or WPF),

your steps will be similar. In place of Step 3, simply reference the OpcLabs.EasyOpcClassic assembly, and create a new instance of **OpcLabs.EasyOpc.DataAccess.EasyDAClient** object.

## 2.1.3 Where do I go from here?

You now have a basic idea of how to use the QuickOPC.NET. There are many additional resources that you can use to gain additional knowledge about the product. Here are some of them:

- Play with the Demo applications (accessible from the Start menu or the Launcher application) and explore the various functionalities that the product provides.
- Read the User's Guide.
- Study the Reference documentation.
- Review the additional examples and tools included with the product.

## 2.2 Getting Started with OPC UA under .NET

QuickOPC-UA allows you to develop using various development models. In this Getting Started procedure, we will present two of them:

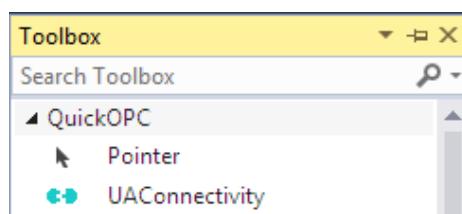
- **Making a first QuickOPC-UA application using Live Binding (without any coding) (Section 2.2.1)**, and
- **Making a first QuickOPC-UA application using traditional coding (Section 2.2.2)** (procedural).

Other development models available are *live mapping*, and *reactive programming (Rx)*.

### 2.2.1 Making a first OPC UA application using Live Binding

In this Getting Started procedure, we will create a Windows Forms application in C# or Visual Basic that subscribes to OPC Data Access (OPC-DA) item and continuously displays its changes. The live binding model allows achieving this functionality by simply configuring the QuickOPC components, without any manual coding.

1. Install QuickOPC-UA. If you are reading this text, chances are that you have already done this.
2. Start Microsoft Visual Studio 2012, 2013 or 2015, and create new Visual C# or Visual Basic project, selecting "Windows Forms Application" template. **Make sure that in the upper part of the "New Project" dialog, the target framework is set to ".NET Framework 4.5.2" or later.**
3. Drag the UAConnectivity component from the "QuickOPC" tab of the Toolbox to the form's design surface.



Notice that three icons will appear below the form: One labeled "daConnectivity1", the second labeled "pointBinder1", and the third labeled "bindingExtender1".

If the needed components do not show in the Toolbox, right-click in the Toolbox area, select "Choose Items...", enter **opc labs** into the Filter box, check all boxes next to the displayed components, and press OK.

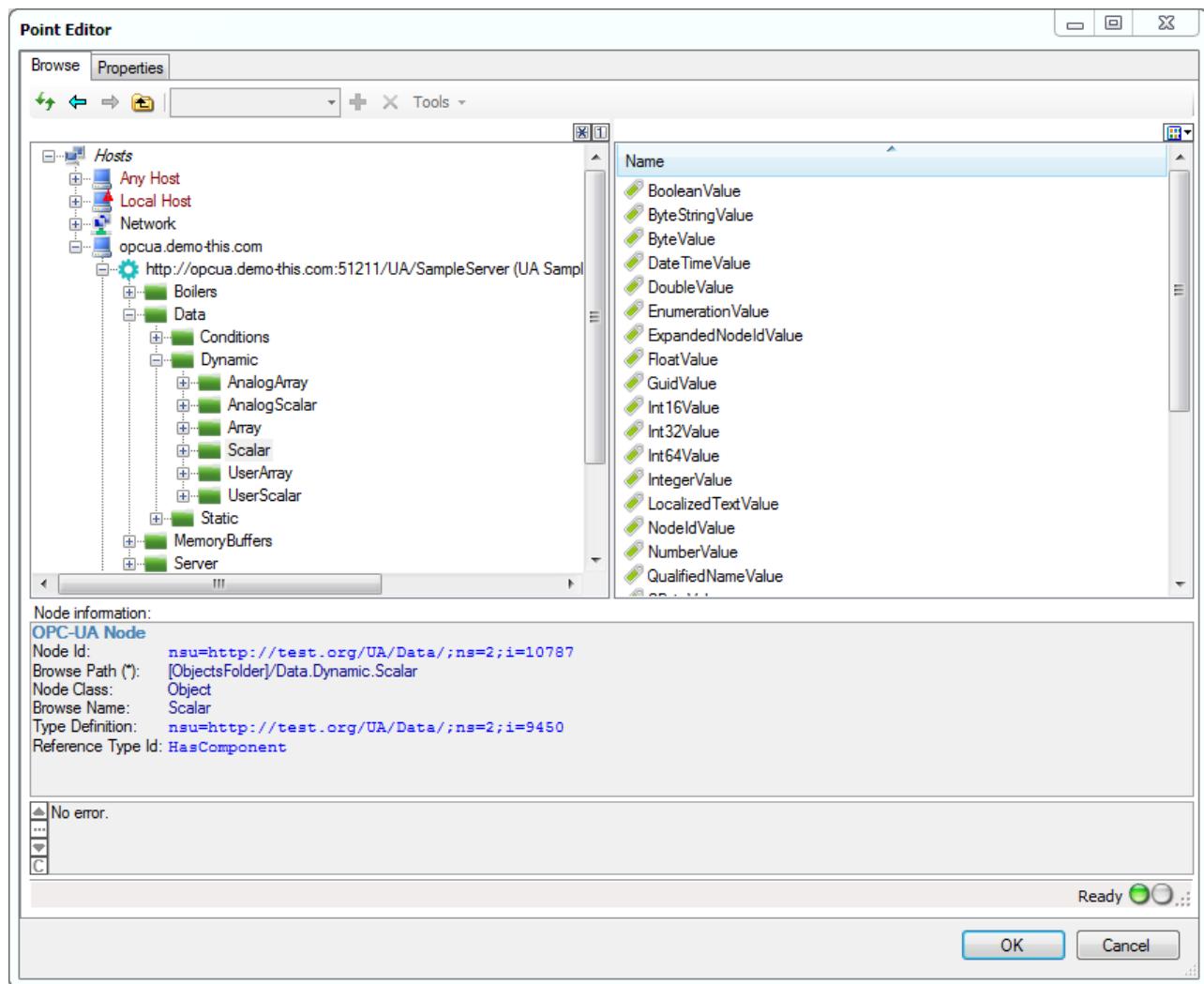
4. Drag a standard TextBox from the Toolbox to the form.
5. Right-click on the text box control on the form, and choose "Bind to Point" command.

Alternatively, you can select the text box control first, and then choose the "Bind to Point" command in the Properties window; for this to work, however, the commands must be displayed in the Properties window.

[Bind to Point...](#) [Edit Live Bindings...](#)  
[Remove Live Bindings](#)

If you do not see these commands, display them by right-clicking in the Properties window, and checking "Commands" in the menu.

- In the "Point Editor" dialog, select Hosts -> opcua.demo-this.com -> http://opcua.demo-this.com:51211/UA/Sample Server (UA Sample Server) -> Data -> Dynamic -> Scalar -> Int32Value, and press OK.



- Build and run your application. You will see live OPC-UA data changing in the text box. It is also possible to verify the effects of live binding without building the application: On the **bindingExtender** component placed on the form, select the "Design Online" command. This will cause Visual Studio to perform the live binding immediately. Use the "Design Offline" command to revert back to normal mode.

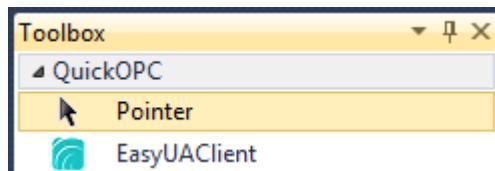
## 2.2.2 Making a first OPC UA application using traditional coding

If you are interested in QuickOPC-UA, your task will most certainly involve reading data from an OPC server. Here are a few steps that illustrate how to achieve that, using Microsoft Visual Studio 2012, 2013 or 2015, and a simple

Windows Forms application in C#. The steps are quite similar if you are using other tools or a different programming language.

We will create a form that will read a float value from the OPC server when loaded, and immediately display the formatted value in the text box on the form.

1. Install QuickOPC-UA. If you are reading this text, chances are that you have already done this.
2. Start Microsoft Visual Studio 2012, 2013 or 2015, and create new Visual C# project, selecting "Windows Forms Application" template. **Make sure that in the upper part of the "New Project" dialog, the target framework is set to ".NET Framework 4.5.2" or later.**
3. Instantiate the EasyUAClient component in the form: Drag the EasyUAClient component from the "QuickOPC" tab of the Toolbox to the form's design surface.



An icon labeled "easyUAClient1" will appear below the form.

If the needed components do not show in the Toolbox, right-click in the Toolbox area, select "Choose Items...", enter **opcLabs** into the Filter box, check all boxes next to the displayed components, and press OK.

Note: This is just shortcut way in Windows Forms to declaring and instantiating the component. In other environments, you can instantiate the component using the 'new' (C#) or 'New' (VB.NET) keyword, or any other way provided by the language or tool you are using.

4. Add a textbox to the form: Drag the TextBox component from the Toolbox to the form's design surface. The text box should appear on the form, and in the Properties window, you will see that it has been given a "textBox1" name.
5. Add handler for the Load event of the form: Select the form by clicking on its design surface (make sure that you do not have any other control selected). Then, in the Properties window, choose the "Events" view (orange lightning icon), and double-click the line with "Load" event (it is under Behavior group). A "Form1\_Load" text in bold font will appear next to the event name, and a new event handler will be created for the Load event. The text editor will open for the code, and the caret will be placed into the Form1\_Load method.

6. Add following code to the beginning of the Form1.cs file:

```
using OpcLabs.EasyOpc.UA;
```

7. Write the event handler implementation. Add the following code to the body of Form1\_Load method:

```
textBox1.Text =  
easyUAClient1.ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",  
"nsu=http://test.org/UA/Data/;i=10853").ToString();
```

8. Build the project.
9. Run the application and observe the results. The text box on the form will be filled with a float value we have read from the OPC server, using a single code line!

If you are targeting an environment different from Windows Forms (such as console application, Web, or WPF), your steps will be similar. In place of Step 4, simply reference the OpcLabs.EasyOpcUA assembly, and create a new instance of **OpcLabs.EasyOpc.UA.EasyUAClient** object.

## 2.2.3 Where do I go from here?

You now have a basic idea of how to use the QuickOPC-UA. There are many additional resources that you can use to gain additional knowledge about the product. Here are some of them:

- Play with the Demo application (accessible from the Start menu or the Launcher application) and explore the various functionalities that the product provides.
- Read the User's Guide.
- Study the Reference documentation.
- Review the additional examples and tools included with the product.

## 2.3 Getting Started with OPC Classic under COM

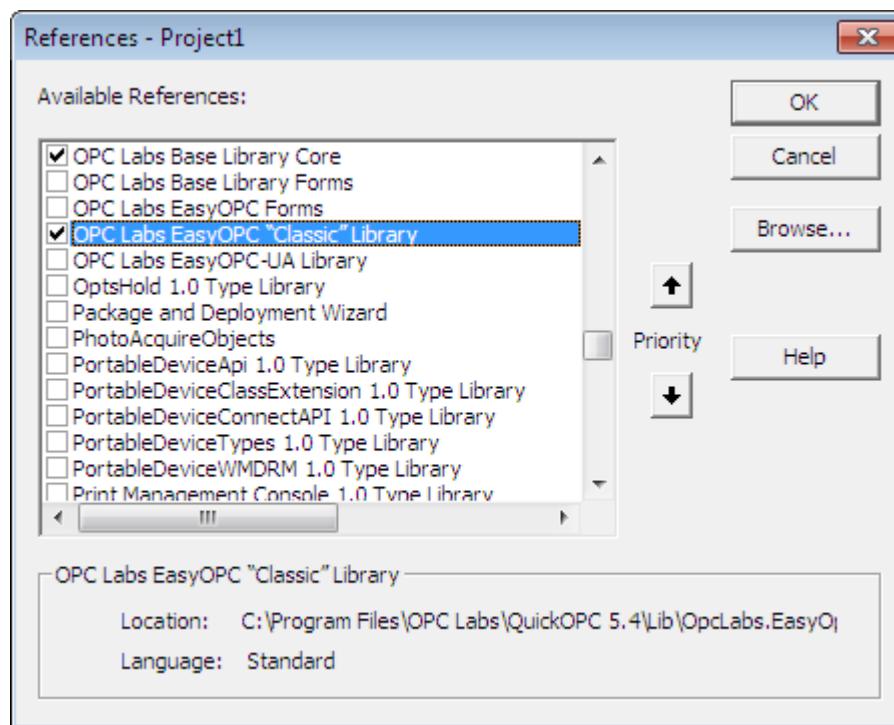
Start here: **Making a first COM application (Section 2.3.1)**.

### 2.3.1 Making a first COM application

If you are interested in QuickOPC-COM, your task will most certainly involve reading data from an OPC server. Here are a few steps that illustrate how to achieve that, using Visual Basic 6.0. The steps are similar if you are using other tools or a different programming language.

We will create a form that will read a float value from the OPC server when loaded, and immediately display the formatted value in the text box on the form.

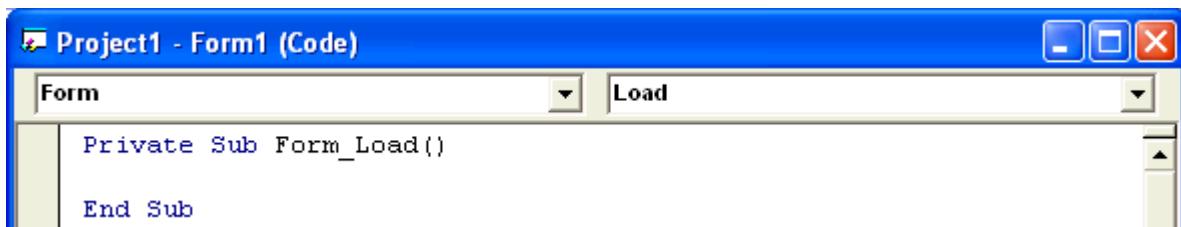
1. Install QuickOPC. If you are reading this text, chances are that you have already done this.
2. Start Visual Basic 6.0, and create new project, selecting "Standard EXE" project type.
3. Reference the EasyOPC-DA component: Select Project -> References from the menu, and look for "OPC Labs EasyOPC "Classic" Type Library" in the list of available references. Check the box next to it, and press OK.



Note: If some of the QuickOPC type libraries are not shown in the References list, use the "Browse..." button

to locate and select them manually. The type libraries are installed under the **Lib** subdirectory under the QuickOPC installation folder. Sometimes, simply restarting the Visual Basic 6.0 helps.

4. Place a Text Box onto the form's surface. Keep the name Text1.
5. Add handler for the Load event of the form: Switch to Code View, and select the Form object and then its Load member. Visual Basic will add the handler's skeleton as follows:



6. Write the event handler implementation. Add the following code to the body of Form\_Load method:

```
' Create EasyOPC-DA component
Dim EasyDAClient As New EasyDAClient

' Read item value and display it
Me.Text1 = EasyDAClient.ReadItemValue("", "OPCLabs.KitServer.2", "Demo.Single")
```

7. Build and run the application by selecting **Run -> Start** from the menu, and observe the results. The text box on the form will be filled with a float value we have read from the OPC server.

If you are using a tool that does not have a referencing mechanism similar to Visual basic, you may need to create the EasyOPC-DA component directly. To do so, use the tool's mechanism to create instance of a COM object, passing it a ProgID string "OpcLabs.EasyOpc.DataAccess.EasyDAClient".

## 2.3.2 Where do I go from here?

You now have a basic idea of how to use the QuickOPC-COM. There are many additional resources that you can use to gain additional knowledge about the product. Here are some of them:

- Play with the Demo application (accessible from the Start menu or the Launcher application) and try obtaining data from the included Simulation OPC server, but also from other OPC servers on your computer or your network.
- Read the User's Guide.
- Study the Reference documentation.
- Review the additional examples and tools included with the product.

## 2.4 Getting Started with OPC UA under COM

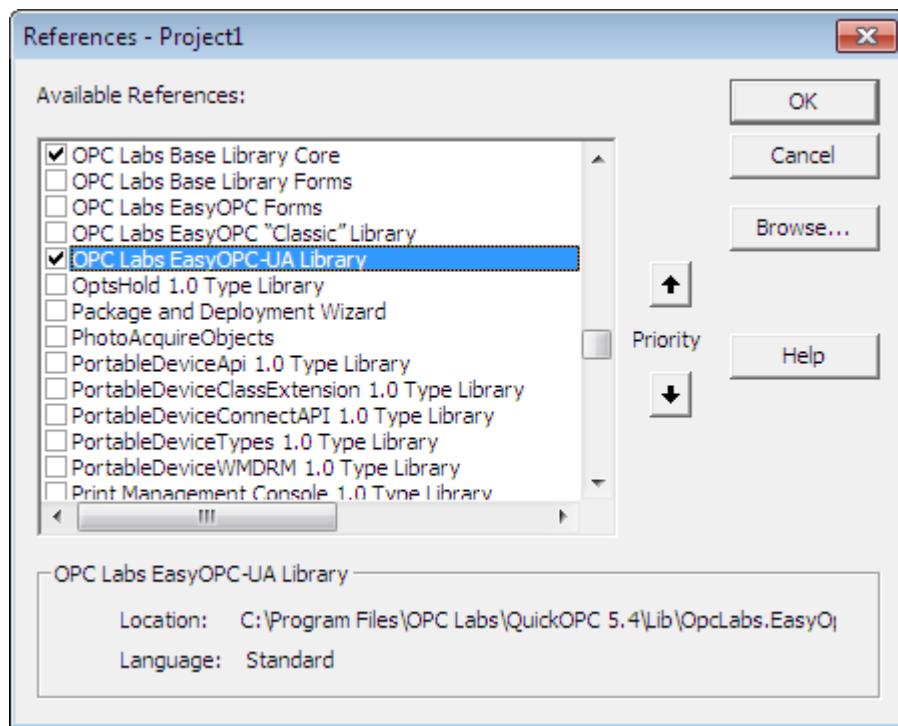
Start here: **Making a first UA application (Section 2.4.1)**.

### 2.4.1 Making a first UA application

If you are interested in QuickOPC-UA, your task will most certainly involve reading data from an OPC UA server. Here are a few steps that illustrate how to achieve that, using Visual Basic 6.0. The steps are similar if you are using other tools or a different programming language.

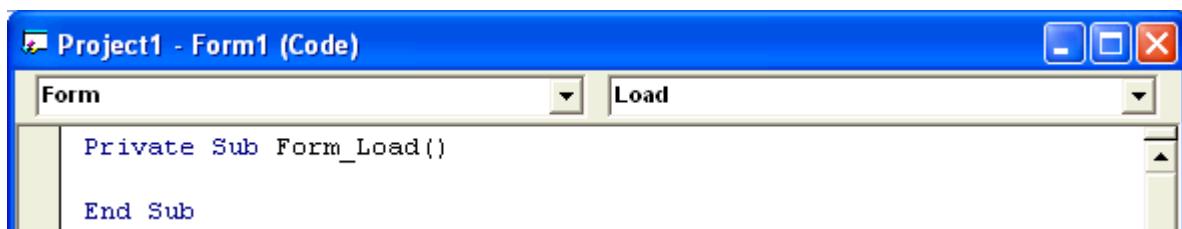
We will create a form that will read a float value from the OPC UA server when loaded, and immediately display the formatted value in the text box on the form.

1. Install QuickOPC. If you are reading this text, chances are that you have already done this.
2. Start Visual Basic 6.0, and create new project, selecting "Standard EXE" project type.
3. Reference the EasyOPC component: Select Project -> References from the menu, and look for "OPC Labs EasyOPC-UA Type Library" in the list of available references. Check the box next to it, and press OK.



Note: If some of the QuickOPC type libraries are not shown in the References list, use the "Browse..." button to locate and select them manually. The type libraries are installed under the **Lib** subdirectory under the QuickOPC installation folder. Sometimes, simply restarting the Visual Basic 6.0 helps.

4. Place a Text Box onto the form's surface. Keep the name Text1.
5. Add handler for the Load event of the form: Switch to Code View, and select the Form object and then its Load member. Visual Basic will add the handler's skeleton as follows:



6. Write the event handler implementation. Add the following code to the body of Form\_Load method:

```
' Create EasyOPC-UA component
Dim Client As New EasyUAClient

' Read node value and display it
```

```
Me.Text1 = Client.ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853")
```

7. Build and run the application by selecting **Run -> Start** from the menu, and observe the results. The text box on the form will be filled with a float value we have read from the OPC UA server.

If you are using a tool that does not have a referencing mechanism similar to Visual basic, you may need to create the EasyOPC-UA component directly. To do so, use the tool's mechanism to create instance of a COM object, passing it a ProgID string "OpcLabs.EasyOpc.UA.EasyUAClient".

## 2.4.2 Where do I go from here?

You now have a basic idea of how to use the QuickOPC-COM. There are many additional resources that you can use to gain additional knowledge about the product. Here are some of them:

- Play with the Demo application (accessible from the Start menu) and explore the various functionalities that the product provides.
- Read the User's Guide.
- Study the Reference documentation.
- Review the additional examples and tools included with the product.

## 3 Installation

QuickOPC comes in two distribution formats:

- Setup program. This is what you get when you directly download the software from the Web site, or obtain it on a distribution media such as CD or flash disk.
- NuGet packages (NuGet is a package manager for development tools such as Visual Studio). The packages are hosted on [www.nuget.org](http://www.nuget.org), and you need to use a NuGet client in order to access them.

The distribution formats are described further below in this chapter.

### 3.1 Common Concerns

#### 3.1.1 Operating Systems

The product is supported on following client operating systems:

- Microsoft Windows 7 (x86 or x64) with Service Pack 1
- Microsoft Windows 8.1 (x86 or x64) with Update
- Microsoft Windows 10 (x86 or x64)

The product is supported on following server operating systems:

- Microsoft Windows Server 2008 R2 (x64) with Service Pack 1
- Microsoft Windows Server 2012 (x64)
- Microsoft Windows Server 2012 R2 (x64)
- Microsoft Windows Server 2016 (x64)

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

#### 3.1.2 Hardware

QuickOPC minimal hardware requirements are the same as those for the operating system and the development tools you are using.

You should have at least 680 MB of free hard disk space before installing QuickOPC.

#### 3.1.3 Included Software

The QuickOPC setup program does not install other software on the target system.

#### 3.1.4 Prerequisites

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

1. Microsoft .NET Framework 4.5.2, 4.6 or 4.6.1.

**Needed when:** Always.

Note that

- .NET Framework 4.6 is included in Windows 10.

## 3.1.5 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, QuickOPC-UA and QuickOPC-COM parts are licensed together.

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 your 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 (GUI or console-based), described further in this document.

## 3.1.6 Related Products

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

## 3.2 Setup Program

### 3.2.1 Running the Setup

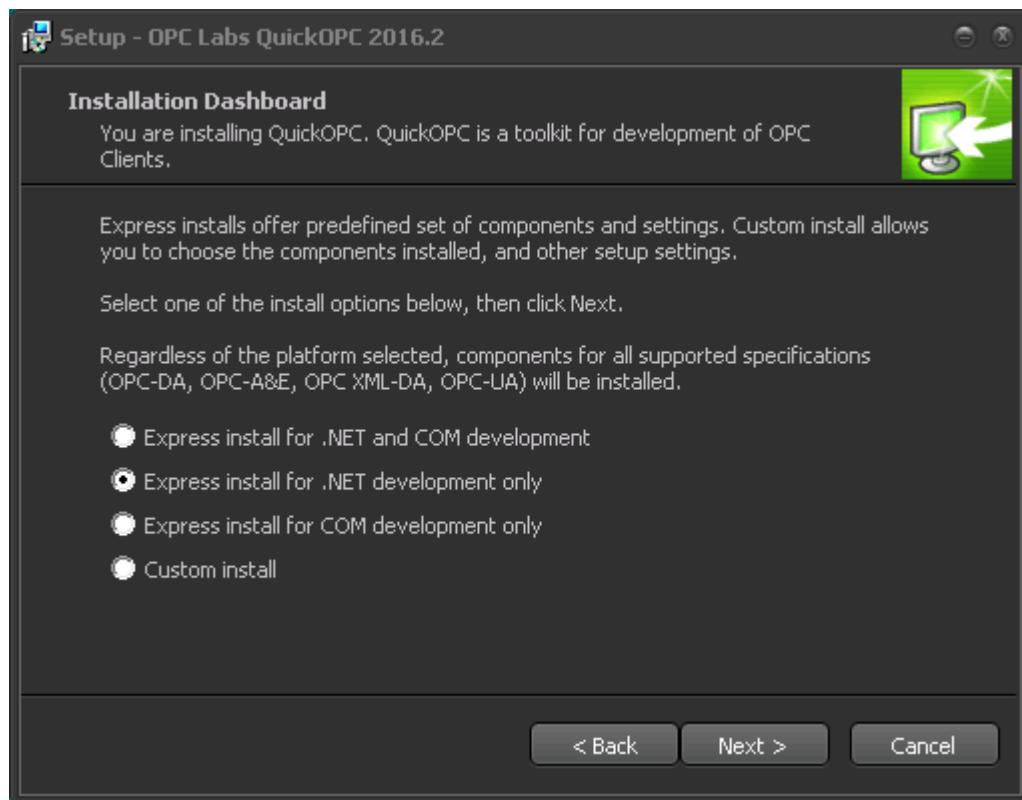
The installation can be started by running the setup program. Just follow the on-screen instructions. The installation program requires that you have administrative privileges to the system.

The installation package file is code-signed, increasing security, improving trust, and providing better download experience from the Web. We use "COMODO RSA Code Signing CA" certificate. The name of signer is "CODE Consulting and Development, s.r.o.". Certain other items in the product (such as the executables of the License Manager and OPC Kit Server, and the Help files, are also code-signed, using the same certificate).

The installation wizard starts with an introductory screen:



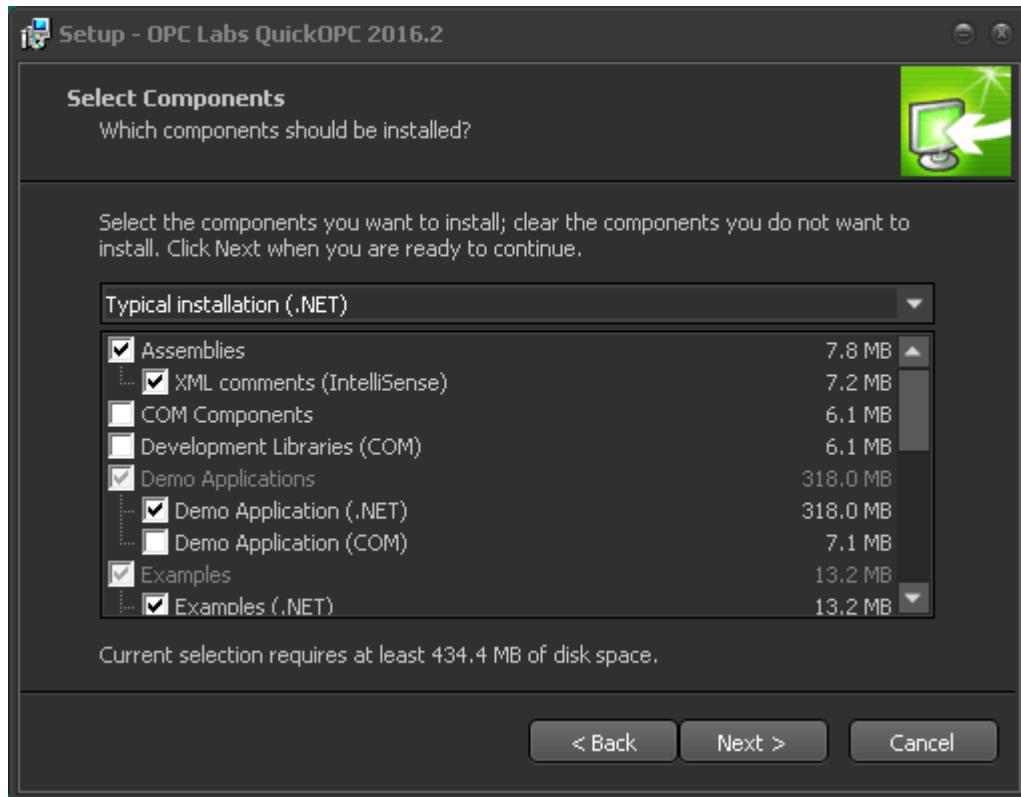
After an introductory screen, the setup wizard offers you the basic installation options:



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.

For "custom install", the component selection wizard page looks like this:



After the installation, the Start menu contains a number of hierarchically organized icons for various parts of the product. There is also a "Product Folder", which opens the directory where all product files reside, making it easier to locate them.

When the installation is finished, it opens the Launcher application. This application is a switchboard that presents the same shortcuts as those available from the Start menu. It serves as a starting point for work with QuickOPC.

The Setup program also places a Launcher application shortcut to the desktop, to the Programs group (Start menu), and to the Quick Launch bar (note that Quick Launch isn't visible by default; to enable it, see <https://support.microsoft.com/en-us/kb/975784>).

When the installation is finished, it also opens the Getting Started section of the documentation. You can also access the documentation and various tools from your Start menu, or the Launcher application that is installed together with the product.

When installing version QuickOPC, it is strongly recommended that you first uninstall any version numbered between 5.00 and 5.31, instead of simply applying the installation over the previous version. The reason for this is that the earlier versions installed the runtime assemblies into the Global Assembly Cache (GAC), whereas versions 5.32 and later do not do this by default. Having older assemblies in the GAC while developing with newer assemblies outside of the GAC is possible in principle, but without extra precautions, it can lead to all kinds of mix-ups and confusion.

The full standalone installer has a German localization. When the Setup programs detects that current user's UI language is German, it automatically switches to this localization.

If the needs of your organization require you to automate the installation of QuickOPC, it is possible to do so, using switches on the command line (see <http://www.jrsoftware.org/ishelp/index.php?topic=setupcmdline>). For additional information, see a related article in the Knowledge Base, **Full installer Types, Components and Tasks**. Note that for production purposes, it is not expected that you will automate the full installer in this way; you should be using the production installer instead (see the Application Deployment section).

## 3.2.2 Uninstallation

The product includes an uninstall utility and registers itself as an installed application in Windows. 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 or the Launcher application.

You can uninstall QuickOPC using following steps (x.yz denotes the version you have installed):

1. Go to **Windows Control Panel** and select "Uninstall a program". In the list of programs, locate "QuickOPC x.yz Build m.n", and double-click on it. Alternatively, from **Windows Start menu**, select All programs -> OPC Labs -> QuickOPC x.y -> Uninstall.
2. Press the "Yes" button when asked whether you are sure to uninstall.

These instructions are also available from the Start menu as a HTML document. To access them, select All programs -> OPC Labs -> QuickOPC x.y -> Uninstall Instructions.

## 3.2.3 Troubleshooting the Setup

In case of non-obvious installation problems, it might be helpful to view a detailed installation log. The installer creates a log file automatically. By default, it is located in the user's temporary directory, and its name starts with "Setup Log" and it includes the date of the installation and a sequence number.

For example, the installation log file location and name might be:

"C:\Users\TestUser\AppData\Local\Temp\Setup Log 2014-11-03 #001.txt"

## 3.3 NuGet Packages

NuGet ([www.nuget.org](http://www.nuget.org)) is the package manager for the Microsoft development platform including .NET. The NuGet client tools provide the ability to produce and consume packages. QuickOPC NuGet packages are hosted in the NuGet Gallery (<https://www.nuget.org/packages>). The NuGet Gallery is the central package repository used by all package authors and consumers.

When you use NuGet to install a package, it copies the library files to your solution and automatically updates your project (add references, change config files, etc.). If you remove a package, NuGet reverses whatever changes it made so that no clutter is left.

Note that NuGet is primarily a tool for resolving build-time dependencies. The amount of functionality that you get through QuickOPC NuGet packages is smaller than what QuickOPC can actually do for you. If you want a full coverage of the features, you would be better off installing the product using the Setup program. Further below you will find a list of differences between the two distribution forms.

The bulk of this documentation assumes that you have installed the product using the Setup program. You need to keep the limited scope of NuGet in mind when interpreting the documentation.

### What is included in the NuGet packages:

- Runtime assemblies for all OPC specifications and programming models.
- User interface components (OPC browsing dialogs and browsing controls for Windows Forms).
- IntelliSense support (XML comments).
- Code Contracts assemblies.
- Examples in LINQPad. This means that whenever you reference one of the QuickOPC NuGet packages in LINQPad, LINQPad will automatically add the QuickOPC examples under its "Samples" tab.

### What is only available from the Setup program:

- Support for COM development (VB6, PHP, Excel, Delphi and similar tools).

- Documentation and Help.
- Visual Studio integration, including Live Binding design-time support (codeless creation of OPC applications).
- Examples and Demo applications, Bonus.
- OPC Data Access simulation server, test tools.
- License Manager utility.

## 3.3.1 List of Packages

Following QuickOPC NuGet packages are available:

- **OpcLabs.QuickOpc**: OPC client components for all environments and project types.
- **OpcLabs.QuickOpc.Forms**: Components that are specific for Windows Forms (can be partially used from WPF as well).

The strings in **bold** are the package IDs.

## 3.3.2 Using Package Manager GUI

This text assumes that you are using Visual Studio 2015. With other tools, the procedures may be different, but should be reasonably similar.

In order to reference a NuGet package from your project using the Package Manager GUI:

1. In the Visual Studio Solution Explorer window, right-click the project, and select **Manage NuGet Packages**.
2. Make sure that "Package source" is set to "nuget.org".
3. Type "QuickOPC" into the Search box. A list of available QuickOPC packages appears.
4. Click on the package you are interested in. Its details appear in the pane next (right) to it.
5. If needed, select a specific version of the package.
6. Press the "Install" button. A "Preview" windows appears, summarizing the changes that are about to be made.
7. Review the changes, and press the "OK" button.
8. The NuGet client will now update your project as needed. Among other things, it will download and copy over the QuickOPC assemblies, and reference them.
9. A "readme.txt" file appears in the editor, with additional information. It is recommended that you read the file.
10. You can now start using the types from the referenced QuickOPC assemblies.

If you do not see the packages in the NuGet Package Manager, make sure your project's framework is set to 4.5.2 or later.

## 3.3.3 Using Package Manager Console

This text assumes that you are using Visual Studio 2015. With other tools, the procedures may be different, but should be reasonably similar.

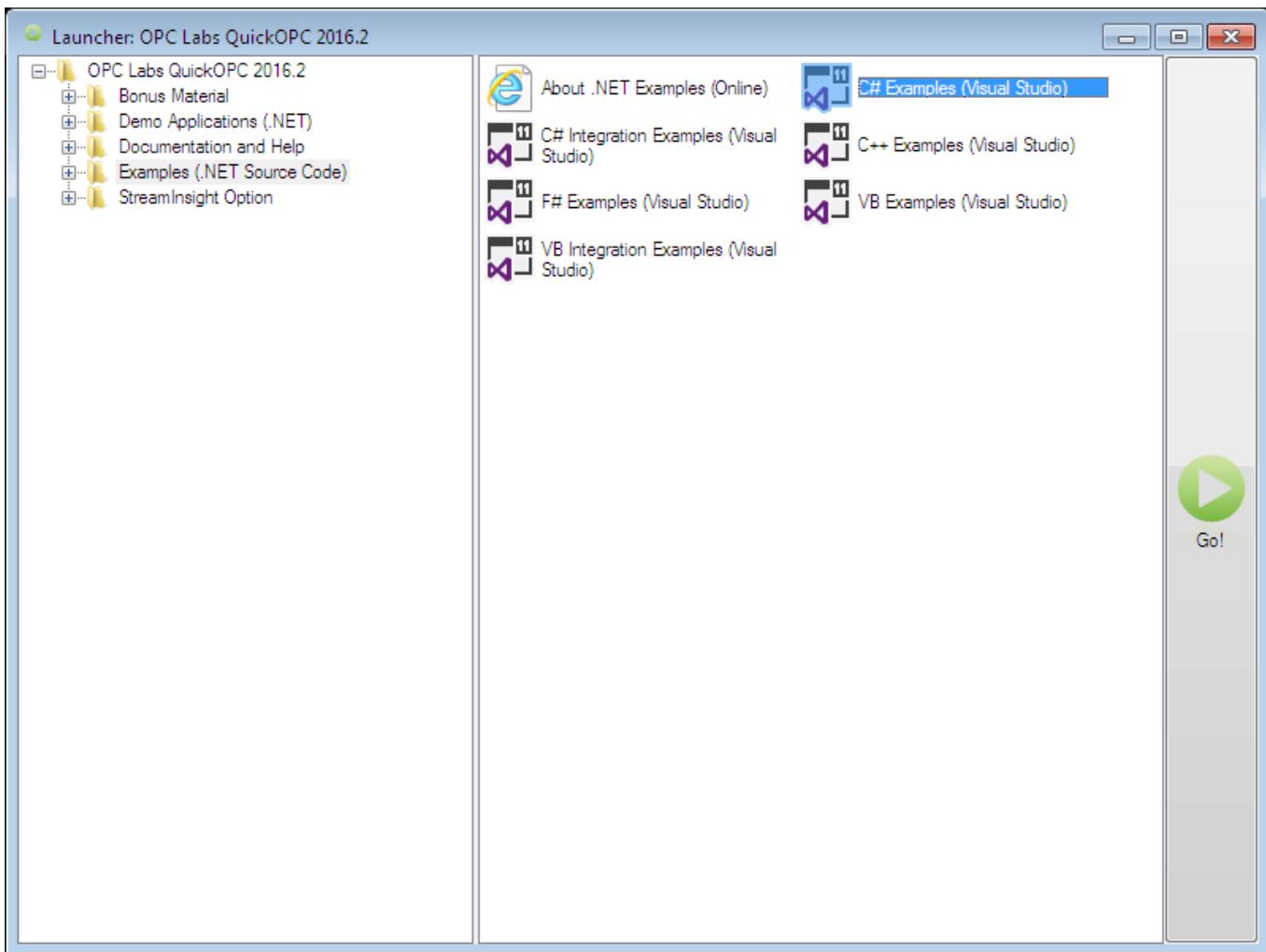
In order to reference a NuGet package from your project using Package Manager Console:

1. In the Visual Studio, select **Tools -> NuGet Package Manager -> Package Manager Console**.
2. Make sure that "Package source" is set to "nuget.org".
3. In the "Default project" drop-down, select the project.
4. Enter command "Install-Package *Id*", where *Id* is the package ID of the package to install (for example, **OpcLabs.QuickOpc**). Press Enter.
5. The NuGet client will now update your project as needed. Among other things, it will download and copy over the QuickOPC assemblies, and reference them.
6. A "readme.txt" file appears in the editor, with additional information. It is recommended that you read the file.
7. You can now start using the types from the referenced QuickOPC assemblies.

## 4 Product Parts

### 4.1 The Launcher application

The Launcher application is a switchboard that presents the same shortcuts as those available from the Start menu. It serves as a starting point for work with QuickOPC. In addition, for users of Windows 8 and later, it allows them to easily navigate the various QuickOPC shortcuts, as the Start menu in Windows 8 and later is essentially broken for applications that install a hierarchical structure of shortcuts.



The Launcher application is also accessible from a desktop shortcut, from the Programs group (Start menu), and from the Quick Launch bar (note that Quick Launch isn't visible by default; to enable it, see <https://support.microsoft.com/en-us/kb/975784>).

You can use the tree on the left side to select a group of actions you are interested in, and then view the actions in the list on the right side. You can launch any action by double-clicking on its icon, or by selecting it first and then pressing the "Go!" button.

### 4.2 Assemblies

 At the core of QuickOPC.NET and QuickOPC-UA 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\NET452** under the installation directory of the product. For easy recognition among other assemblies when used in a larger context, all our assemblies start with "OpcLabs" prefix.

Following assemblies are part of QuickOPC.NET and QuickOPC-UA:

Assembly Name or File	Title	Description
App_Web_OpcLabs.EasyOpcClassicRaw.amd64	EasyOPC "Classic" Raw Library	Low-level OPC "Classic" code, a mixed mode assembly, for 64-bit processes (x64 architecture).
App_Web_OpcLabs.EasyOpcClassicRaw.x86	EasyOPC "Classic" Raw Library	Low-level OPC "Classic" code, a mixed mode assembly, for 32-bit processes (x86 architecture).
OpcLabs.BaseLib	OPC Labs Base Library Core	Supporting code
OpcLabs.BaseLibForms	OPC Labs Base Library Forms	Contains fundamental and common classes for Windows Forms.
OpcLabs.EasyOpcClassic	EasyOPC "Classic" Library	Contains classes that facilitate easy work with various OPC specifications, such as OPC Data Access and OPC Alarms and Events.
OpcLabs.EasyOpcForms	EasyOPC Forms	Contains classes that facilitate easy work with OPC (both "Classic" and Unified Architecture) from Windows Forms applications.
OpcLabs.EasyOpcUA	EasyOPC-UA Library	Contains classes that facilitate easy work with OPC Unified Architecture.

The name of the physical file that contains the assembly is always same as the name of the assembly, with a file extension ".dll".

QuickOPC.NET and QuickOPC-UA components were consciously written to target Microsoft .NET Framework 4.5.2, 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 4.5.2, 4.6 or 4.6.1 of the Microsoft .NET Framework.

The OPC Unified Architecture components are based on version 1.02 of the .NET Stack from OPC Foundation.

For the curious, QuickOPC.NET and QuickOPC-UA have been developed in Microsoft Visual Studio 2015. The layers that directly use COM (such as the OpcLabs.EasyOpcClassic assembly) are written in managed C++ (C++/CLI). More precisely, they are 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#.

There is also a Design subfolder under the Assemblies folder, which contains additional QuickOPC assemblies that are used during design time only. These assemblies are never referenced or used in run-time by your projects.

For development purposes, the assemblies under the Design subfolder are installed into the GAC (Global Assembly Cache) by the setup program. Other (runtime) assemblies are not installed into the GAC by default (the installation program can do it if you select the custom installation, it will then offer you this option near the end of the installation). You do not have to install assemblies into the GAC on the runtime computers, though. We recommend against installing the runtime assemblies into the GAC, unless you have a compelling reason for it.

The somewhat strange naming of certain assemblies, with names starting with "App\_Web\_" prefix, is needed for proper mixed-mode assembly loading under ASP.NET. Do not be confused by the presence of the word "Web" in the

name – the same assemblies are needed outside of Web applications as well.



The .NET assemblies are also necessary when you are using COM components for OPC. They are implemented in .NET, and exposed to the COM worlds using a COM interop.

When specifically selected during an installation (with the "Embedded assemblies" install component, or as part of the Full install), additional assemblies (that are normally embedded within QuickOPC assemblies and thus invisible to the developer) are placed alongside the usual QuickOPC assemblies. These assemblies are needed in certain rare situations, e.g. when registering type libraries for the dependent assemblies for COM interop, or under certain hosting environments or in some development tools.

## 4.2.1 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.

## 4.2.2 ReSharper Annotations

API members are annotated using ReSharper (<http://www.jetbrains.com/resharper/>) attributes, such as the most common [CanBeNull] and [NotNull] attributes. ReSharper users benefit from the annotations, as possible improper usages are recognized and highlighted by ReSharper's code inspection feature.

## 4.2.3 Code Contracts

The API uses Microsoft Code Contracts (<http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx>, <http://research.microsoft.com/en-us/projects/contracts/>) to express coding assumptions. This results in higher consistency in error checking, and the contracts are also explicitly visible in the reference documentation, showing the developer clearly what the expectations about method inputs and outputs are, etc. Developers who want to gain further benefit can enable Code Contracts in their projects, and use the QuickOPC contract assemblies supplied with the product for achieving better code quality.

Assemblies that contain types that are intended to be used by QuickOPC developers are also accompanied by contract assemblies for Code Contracts; names of all such assemblies are made by adding the ".Contracts" postfix to their assembly name, and their file names therefore end with ".Contracts.dll". The contract assemblies should reside in the **"CodeContracts"** subfolder under the Assemblies folder.

Note: The Code Contract assemblies are not installed by the Setup program. Instead, the installer creates a link (to a ZIP file) in the corresponding folder, allowing the developer to download them when needed. The Code Contract assemblies are, however, installed automatically with the QuickOPC NuGet packages.

## 4.3 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.

COM components for OPC "Classic" and OPC UA are implemented in .NET, and exposed to the COM worlds using a COM interop. As such, they are contained in the .NET assemblies described under a corresponding chapter earlier, and installed in a subdirectory called **Assemblies\net452** under the installation directory of the product.

The assemblies become usable from a COM world by registering them using the Regasm.exe (Assembly Registration Tool), which is a part of .NET Framework. The QuickOPC installation program registers the assemblies for COM interop

automatically.

The Assembly Registration tool reads the metadata within an assembly and adds the necessary entries to the registry, which allows COM clients to create .NET Framework classes transparently. Once a class is registered, any COM client can use it as though the class were a COM class. The class is registered only once, when the assembly is installed. Instances of classes within the assembly cannot be created from COM until they are actually registered.

## 4.4 Development Libraries (COM)



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, the information about the objects in the library is made available to the users of the applications and programming tools.

All type libraries (those that are not embedded within other specific .dll or .exe files) are located under the Lib subdirectory in the product installation folder. Type libraries for 64-bit development are located under Lib\x64. Note that the interface exposed by QuickOPC (including the size of integer types) is the same, regardless whether it is for 32-bit or 64-bit development; there are just slight "formal" differences caused by the way the type libraries for 32-bit vs. 64-bit development are generated. Most development can simply use the 32-bit type libraries under the Lib subdirectory.

Note that it is not strictly necessary to use type libraries for COM development. With late binding, it is possible to make use of the COM objects of QuickOPC "as they are". Type libraries are used mainly with early binding.

Whether or not to use the type libraries depends in part on the tool or language you are using. Some allow only late binding (e.g. VBScript), some may allow only early binding, and yet some allow both (e.g. VB6). When your tool allows both options, usually the early binding is better (for performance and type checking reasons), but there are also situations where late binding is still appropriate or necessary. Some of these situations have to do with particular "quirks" present in the tools.

For example, in order to assign atomic values (such as numbers or strings) to a State property of QuickOPC objects, you will need to use late binding in VB6. This is because the property is declared as VARIANT, but VB6 treats it as Object, and does not allow the non-object values be stored into it with early binding.

As described earlier, COM components of QuickOPC are actually created from .NET components, using COM interop tools from .NET Framework. For this reason, there is one COM type library for each assembly that contains the QuickOPC components.

This means that following type libraries are available to you:

Type Library File	Title	Description
OpcLabs.BaseLib.tlb	OPC Labs Base Library Core	Supporting code
OpcLabs.BaseLibForms.tlb	OPC Labs Base Library Forms	Contains fundamental and common classes for Windows Forms.
OpcLabs.EasyOpcClassic.tlb	OPC Labs EasyOPC "Classic" Library	Contains classes that facilitate easy work with OPC Classic.
OpcLabs.EasyOpcForms.tlb	OPC Labs EasyOPC	Contains classes that provide OPC-related (both

	Forms	"Classic and Unified Architecture) user interface.
OpcLabs.EasyOpcUA.tlb	EasyOPC-UA Library	Contains classes that facilitate easy work with OPC Unified Architecture.

This documentation is for QuickOPC version 2016.2. This is known as the *external version* of the product. The *internal version* number is 5.41, and that is also the version of the type libraries.

Each of these libraries is marked with an internal version number of the product, e.g. 5.41. Many COM tools (including e.g. the Visual Basic's References dialog, or OleView), however, display the version number of a type library in a hexadecimal notation, but without giving any indication of it. This may lead to confusion, because e.g. for QuickOPC internal version 5.41, the version gets displayed as 5.29 (a hexadecimal representation of the internal version). Be careful to interpret the version numbers correctly.

## 4.5 Demo Applications

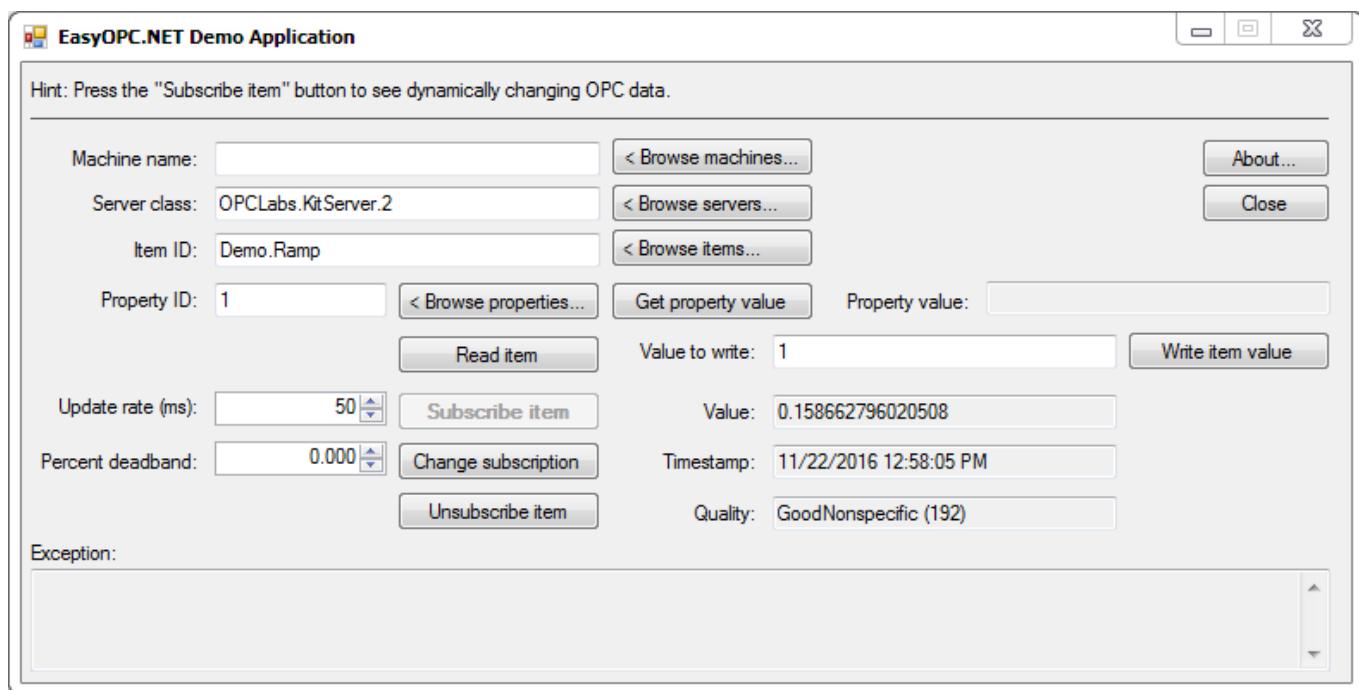
### 4.5.1 .NET Demo Applications

#### 4.5.1.1 .NET Demo Applications for OPC Classic



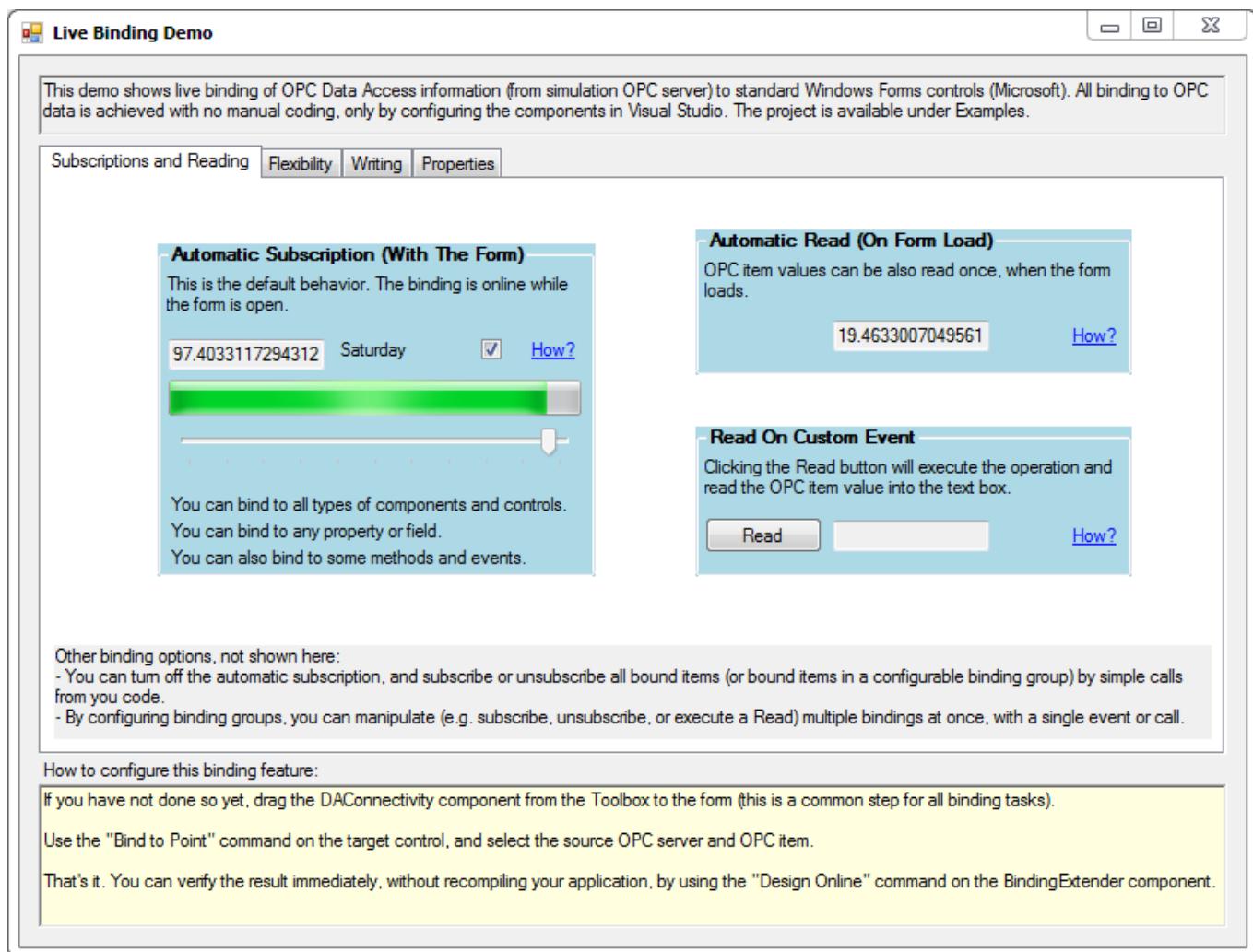
The QuickOPC.NET demo applications are available from the Start menu or the Launcher application.

The basic demo is written using procedural coding model (explained further down in this document).



Note: There is also a similar application with an extra support for OPC XML-DA.

The Live Binding Demo shows live binding of OPC Data Access information (from simulation OPC server) to standard Windows Forms controls (Microsoft). All binding to OPC data is achieved with no manual coding, only by configuring the components in Visual Studio. This demo also serves as a tutorial application for the live binding development model: When you click on any of the "How?" links, the bottom pane displays instructions explaining how to configure that particular feature.



There are also several "integration demos", showing how QuickOPC features can be used in combination with other products that are not part of QuickOPC. Following integration demos are currently available:

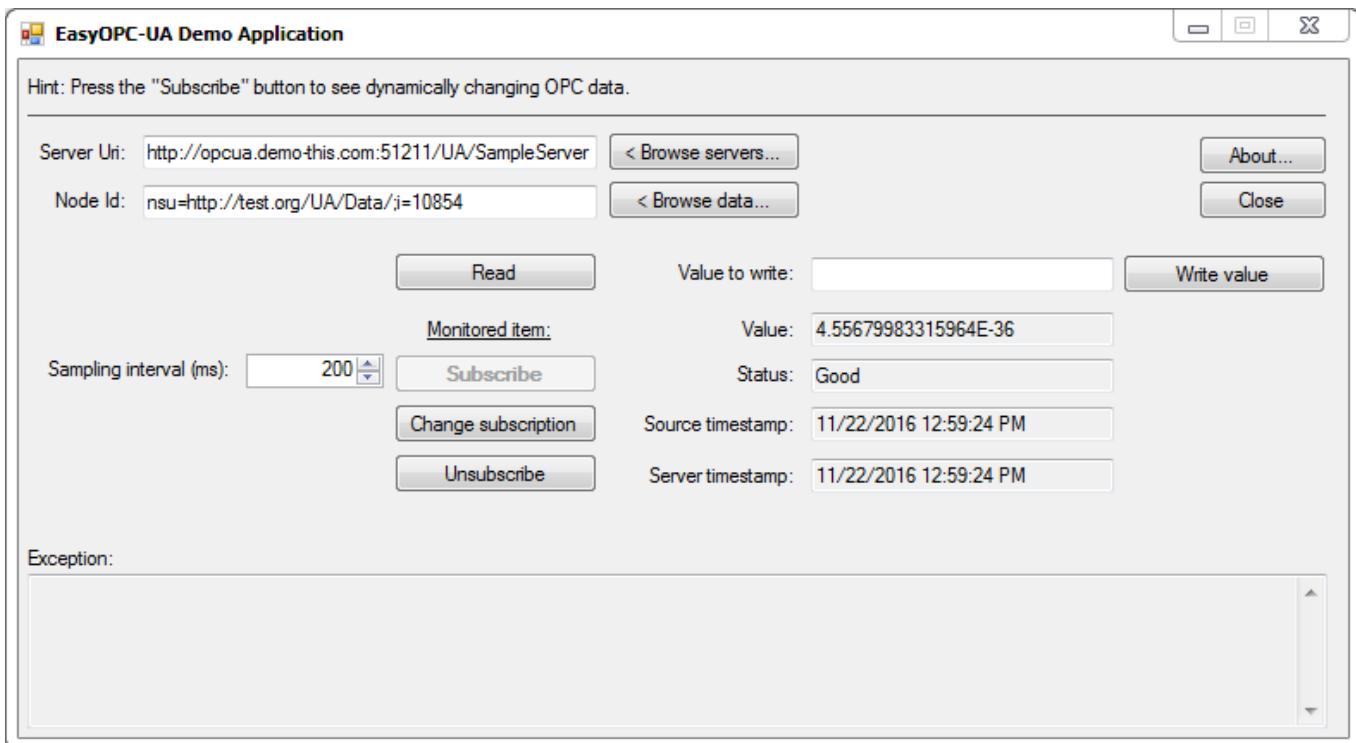
- Demo with Industrial Gadgets .NET
- Demo with Instrumentation Controls
- Demo with Symbol Factory .NET

There is also a similar application that demonstrates the Live Binding capabilities with OPC Unified Architecture.

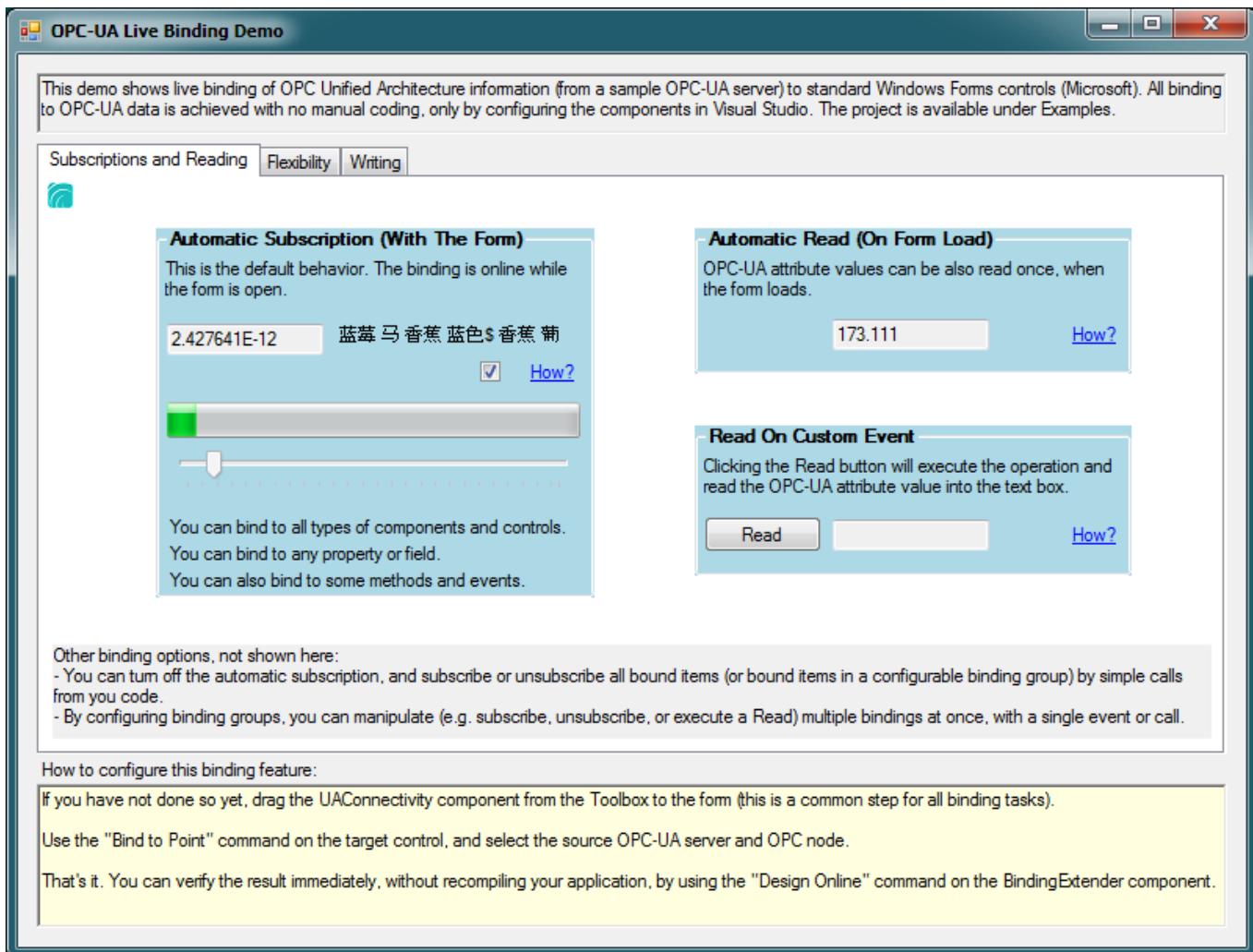
## 4.5.1.2 .NET Demo Applications for OPC UA



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



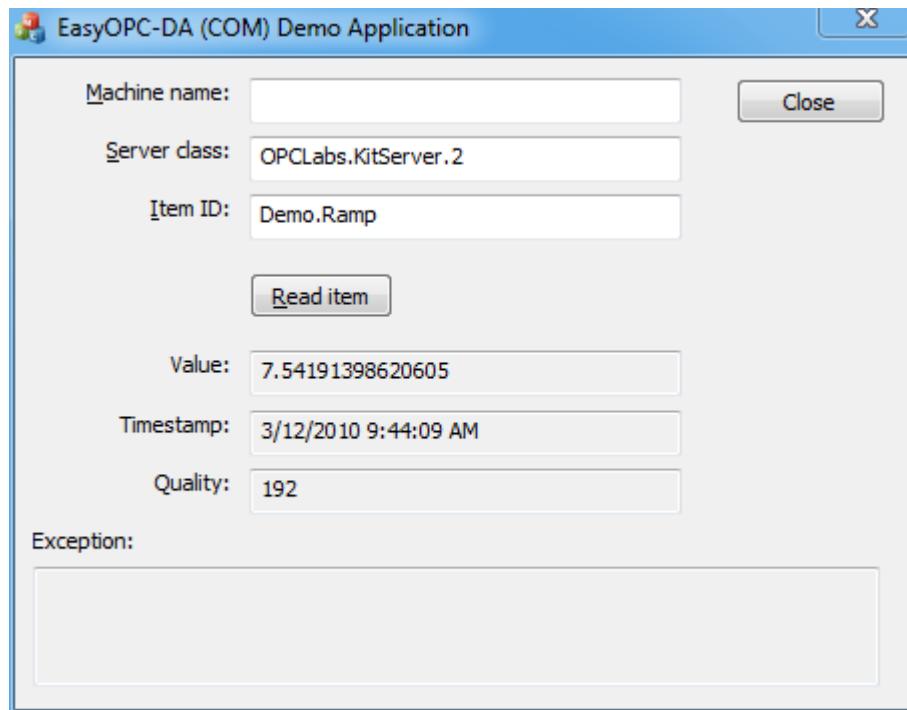
The Live Binding Demo shows live binding of OPC Unified Architecture information (from simulation OPC server) to standard Windows Forms controls (Microsoft). All binding to OPC data is achieved with no manual coding, only by configuring the components in Visual Studio. This demo also serves as a tutorial application for the live binding development model: When you click on any of the “How?” links, the bottom pane displays instructions explaining how to configure that particular feature.



## 4.5.2 COM Demo Applications



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



## 4.6 Examples

Example code, projects and solutions are available from the Start menu or the Launcher application. You can also access them from the file system:

- Examples for COM (QuickOPC-COM and QuickOPC-UA for COM) are in the ExamplesCom subfolder of the installation folder.
- Examples for .NET (QuickOPC.NET and QuickOPC-UA for .NET) are in the ExamplesNet subfolder of the installation folder.

There is a separate **Examples (Section 17)** section in this document that contains list of all examples and their purpose, with chapters for **.NET Examples (Section 17.1)** and **COM Examples (Section 17.2)**. You can access these documentation chapters from the Start menu as well.

### 4.6.1 LINQPad support

LINQPad (<http://www.linqpad.net/>) is a software utility for .NET that allows you to quickly and interactively test out C#, VB.NET and F# code without the need for an IDE. We highly recommend that you use this tool (free edition exists) for experiments and learning.

QuickOPC works well with LINQPad. We have made some special tweaks for it as well, e.g. to ensure a proper expandable display of all types in LINQPad.

QuickOPC standalone installer ships with examples for LINQPad; see the Examples section in this document for details. The NuGet packages include the LINQPad examples as well.

## 4.7 Test Servers

To demonstrate capabilities of QuickOPC, some OPC server is needed. One such server ships with the product, and two are made available remotely.

## 4.7.1 Simulation Server for OPC Classic

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 1: When OPC proxies/stubs are not installed on the system, the installation program installs them together with the simulation server.

Note 2: 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.

## 4.7.2 OPC XML Sample Server

The OPC XML Sample Server is not installed with the product, but is made available on the Internet. You can use it for your explorations and tests. The server's URL is <http://opcxml.demo-this.com/XmlDaSampleServer/Service.asmx>. Our example code also refers to this server, so that the examples can be run without having to set up a server locally.

The server is based on the sample server code from OPC Foundation. It provides some OPC data of its own, and it wraps two OPC COM-based servers (OPC-DA 2.0 and OPC-DA 3.0). It contains a combination of items of various data types, static and dynamic, some readable and some also writeable.

OPC XML does not define a way for server discovery (browsing for servers). For this reason, if you are using user interface components such as the browsing dialogs or controls, you need to specify the server manually, using its URL.

## 4.7.3 OPC UA Sample Server

To demonstrate capabilities of QuickOPC, some OPC server is needed. Most examples use an OPC UA Sample Server provided by OPC Labs (based on OPC Foundation sample server) that is available over Internet. The server can be accessed on "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer" or "http://opcua.demo-this.com:51211/UA/SampleServer" (these are so-called Discovery URIs).

Note that if there is a firewall between the client machine and the server, it needs to have the corresponding ports (in the above example, 51210 and 51211) open for outbound traffic.

The demo application and the examples are designed to connect to the OPC UA Sample Server in its default configuration. In fact, some very simple examples connect to just one OPC node. For example, a dynamically changing double-precision value has node Id "nsu=http://test.org/UA/Data;/i=10854". You can either call the browsing methods to obtain other node IDs together with their descriptions, or you can simply borrow the node IDs used in our examples. There are various other OPC nodes in this server that you can use in your own experiments too.

## 4.7.4 QuickStart Alarm Condition Server (OPC UA)

We run a public Quickstart Alarm Condition Server (OPC UA) that you can use for your explorations and tests. The server is located on [opcua.demo-this.com](http://opcua.demo-this.com), and its endpoint URLs are:

- <opc.tcp://opcua.demo-this.com:62544/Quickstarts/AlarmConditionServer>
- <http://opcua.demo-this.com:62543/Quickstarts/AlarmConditionServer>

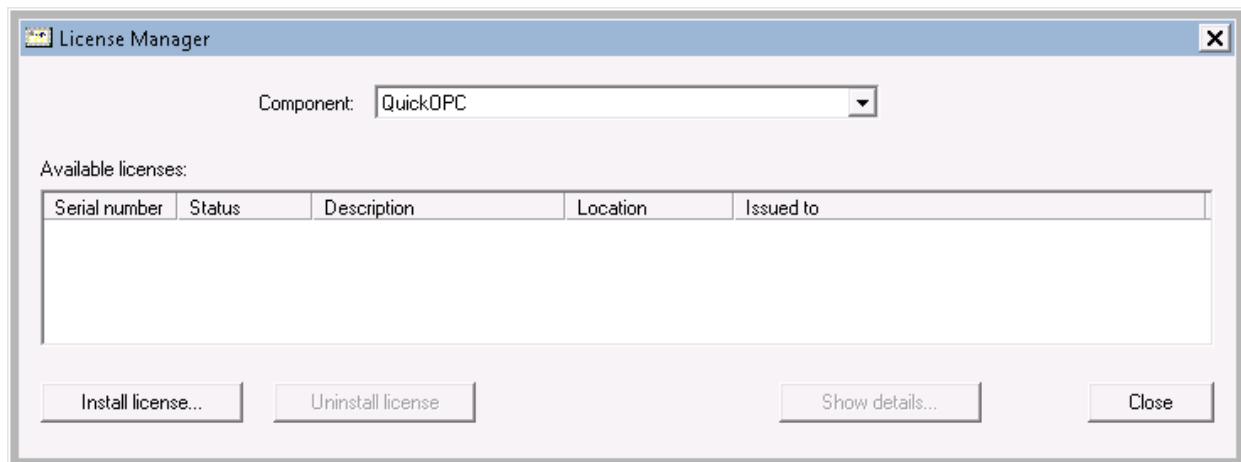
Our example code also refers to this server, so that the examples can be run without having to set up a server locally.

The server is based on the code from OPC Foundation.

## 4.8 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 or the Launcher application), select "QuickOPC" in the component drop-down, 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 elevated privileges to successfully install and uninstall licenses.

### 4.8.1 LMConsole Utility (License Manager Console)

The LMConsole utility is a command-line equivalent of the License Manager, for scripting/automation (e.g. embedded installations), and folks that dislike GUIs in general. The command-line switches are described here, and a brief help is also available from the utility itself.

The utility is installed in the Bin subdirectory under the product folder. Following table summarizes the command-line switches available:

Switch	Description
<b>-u string</b>	Uninstall license
<b>--uninstall string</b>	
<b>-s string</b>	Show license details
<b>--show string</b>	
<b>-q</b>	Query available licenses
<b>--query</b>	
<b>-l</b>	List components
<b>--list</b>	
<b>-i string</b>	Install licenses
<b>--install string</b>	
<b>-c string</b>	Component name
<b>--component string</b>	

--	Ignores the rest of the labeled arguments following this flag.
<b>--ignore_rest</b>	
<b>--version</b>	Displays version information and exits.
<b>-h</b>	Displays usage information and exits.
<b>--help</b>	

The switches are combined to achieve the desired effect. You will typically need following LMConsole commands:

Task	Command
Query available QuickOPC licenses	<b>LMConsole --component QuickOPC --query</b>
Install QuickOPC license (from a file)	<b>LMConsole --component QuickOPC --install fileName</b>
Show QuickOPC license details	<b>LMConsole --component QuickOPC --show Multipurpose</b>
Uninstall QuickOPC license	<b>LMConsole --component QuickOPC --uninstall Multipurpose</b>

## 4.9 Documentation and Help

The documentation consists of following parts:

- User's Guide and Reference (in Visual Studio, and also available online). The reference is formatted according to Visual Studio style and standards.
  - Getting Started. Short step-by-step instructions to create your first project.
  - Examples in .NET.
  - Examples in COM.
  - StreamInsight Option.
- User's Guide (PDF, online). The content is same as above, but the Reference part is not included.
- What's New (online). Contains information about changes made in this and earlier versions.
- Bonus Material (online).



Because the COM components are implemented by exposing the .NET objects of QuickOPC to COM, you may need to refer to QuickOPC reference documentation in order to find information about COM objects and interfaces of QuickOPC as well.

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

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

The QuickOPC help contents integrate Microsoft Visual Studio 2012/2013/2015 Help (Microsoft Help Viewer 2 format). This also means that the reference documentation works as a contextual help; e.g. pressing F1 when the cursor is on some of the QuickOPC types or members invokes directly the help page for that type or member.

For the contextual help to work, you need to have the help preference set properly. In Visual Studio, do this:

Help -> Set Help Preference -> Launch in Help Viewer

(i.e. not Launch in Browser).

## 4.10 Test Tools

QuickOPC ships with certain internal testing tools that might be of interest to advanced users, and were created partially in relation to OPC compliance certification process.

The test tools are not installed by default; you need to select "Custom Install", and then specifically enable them on the "Select Components" page in the installation wizards. After installation, the test tools can be reached through the Start menu or the Launcher application, in the "Test Tools" group.

There is a separate document (online) that contains the list of test tools, their purpose, and instructions on how to operate the tools. When the Test Tools are installed, you can access it from the Start menu as well.

## 5 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.

### 5.1 Typical Usage

QuickOPC.NET and QuickOPC-UA for .NET are 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.



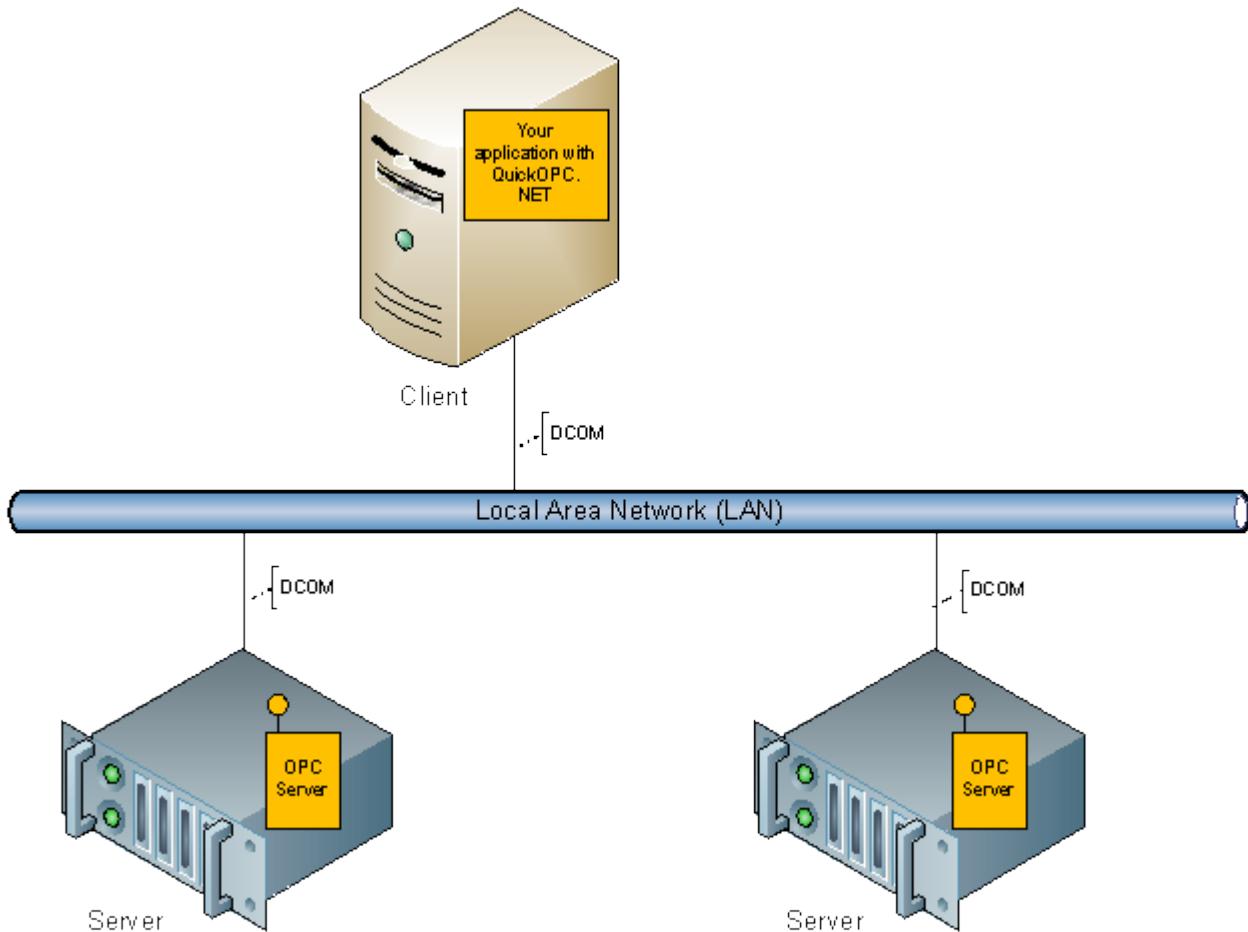
QuickOPC-COM and QuickOPC-UA for COM are suitable for use from within any tool or language based on Microsoft (COM) Automation. There are many different scenarios for it, but some are more common.

#### 5.1.1 OPC Classic 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:

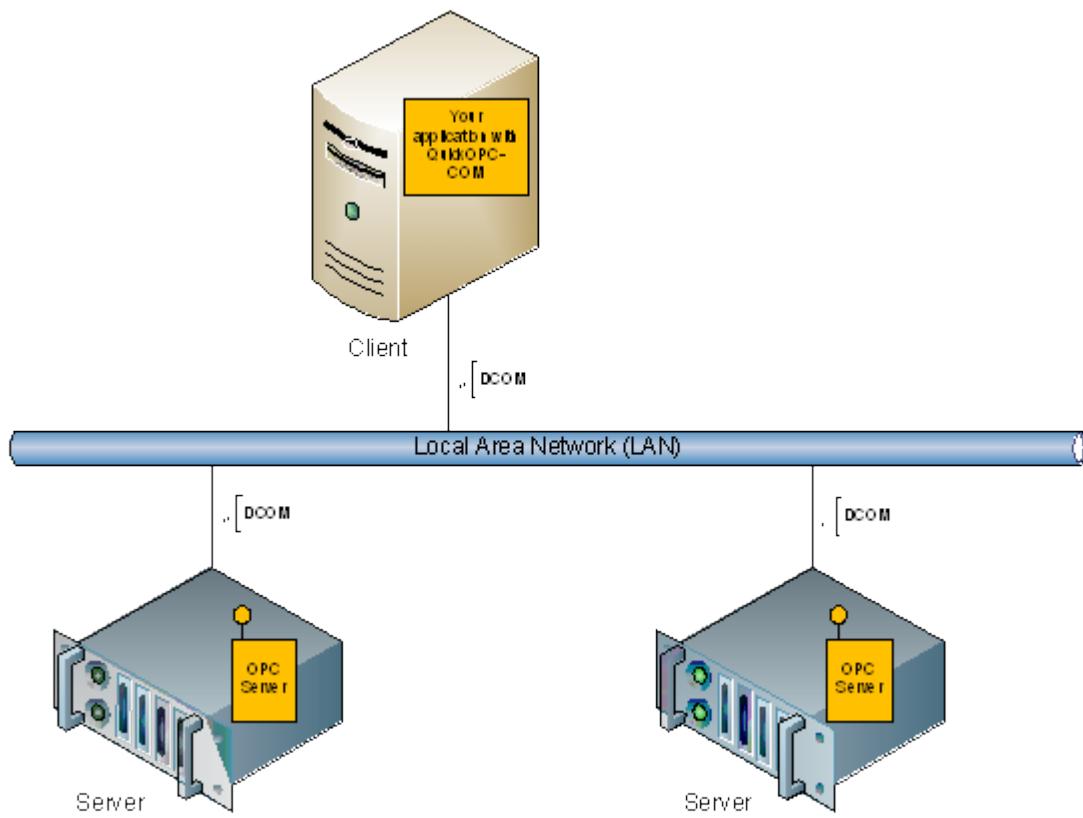


## 5.1.2 OPC Classic or OPC UA thick-client COM applications on LAN



The most typical use of QuickOPC-COM or QuickOPC-UA for 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 (or QuickOPC-UA for 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:



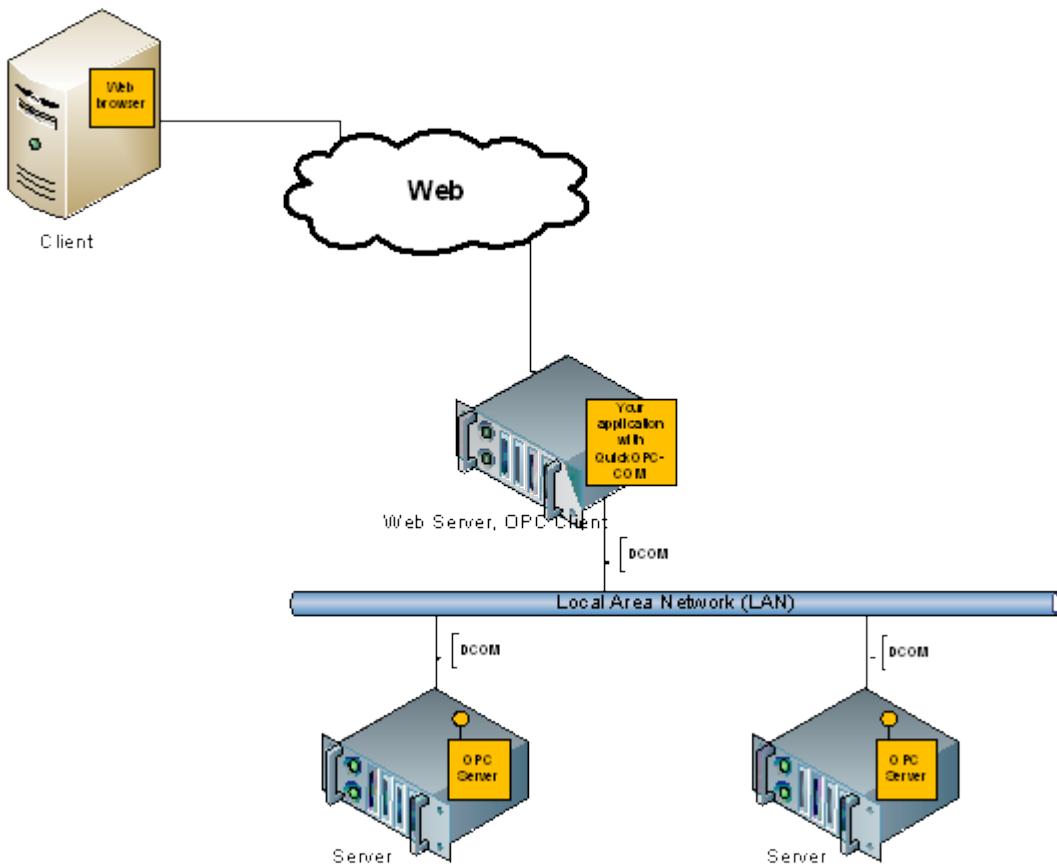
## 5.1.3 OPC Classic Web applications (server side)

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.

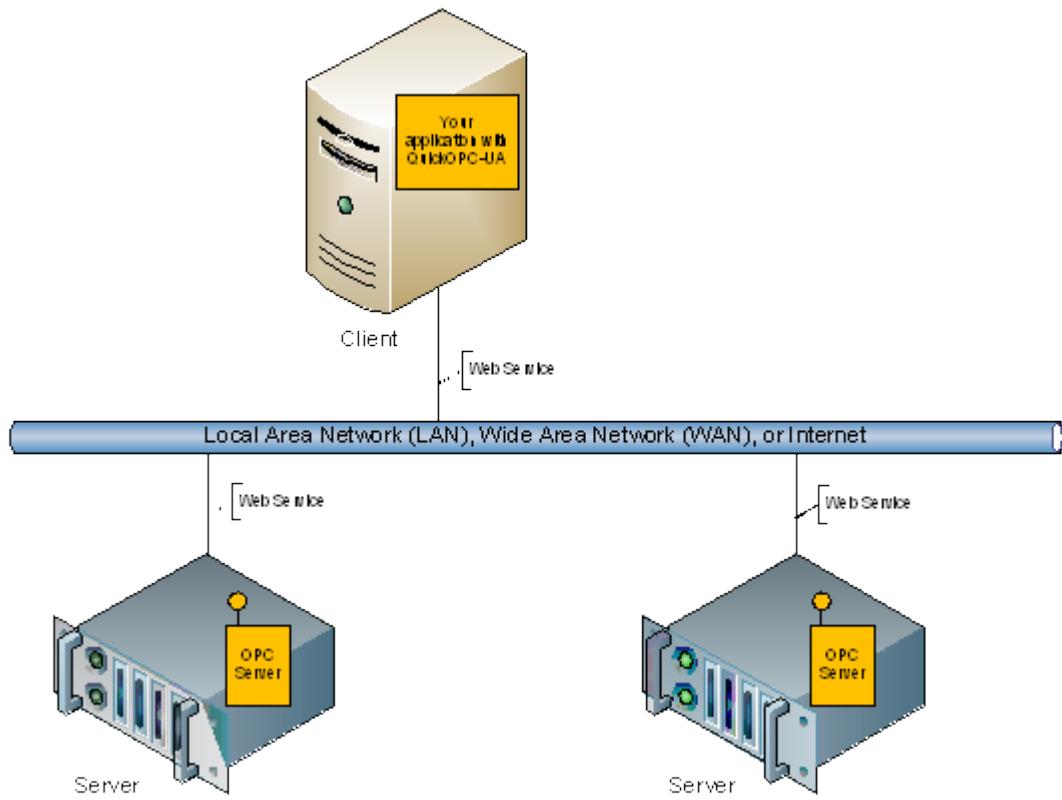
The following picture shows how the individual pieces work together:



## 5.1.4 OPC UA Thick-client applications on LAN, WAN or Internet

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

The following picture shows how the individual pieces work together:

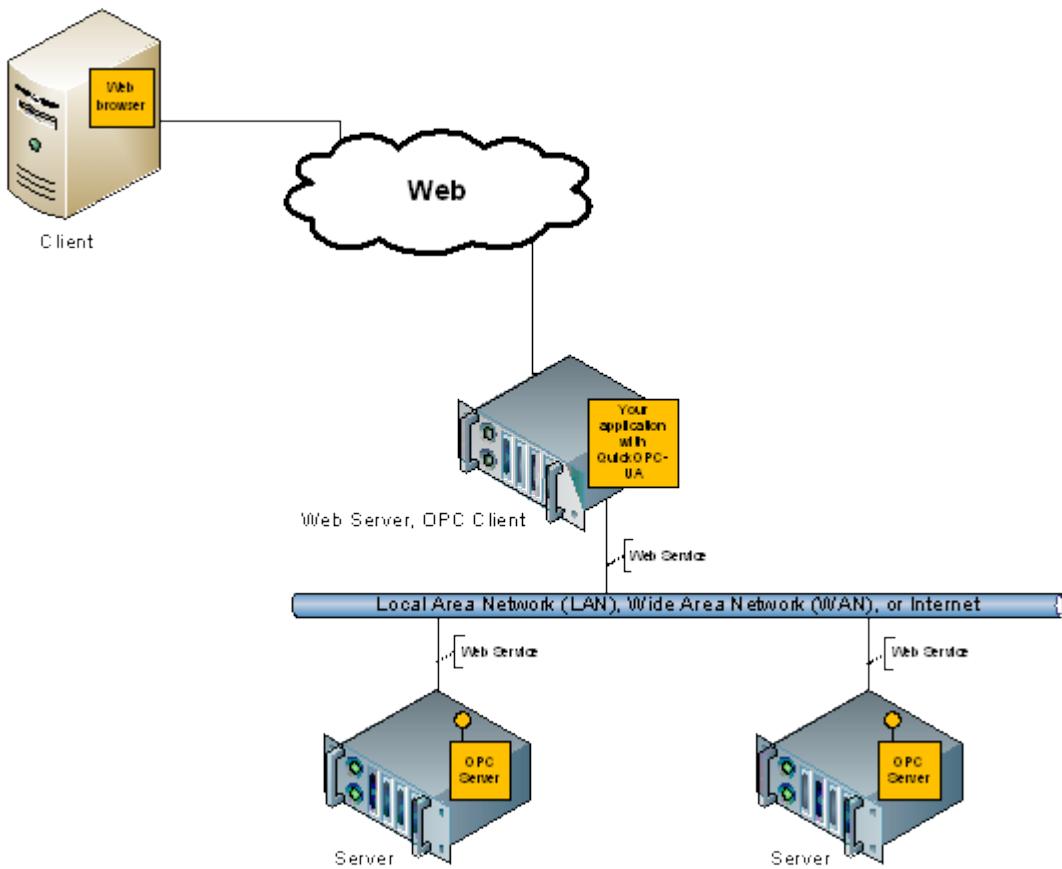


## 5.1.5 OPC UA Web applications (server side)

The other typical use of QuickOPC-UA 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.NET (using C#, Visual Basic.NET, or other .NET language).

The Web application uses the types from QuickOPC-UA 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 means of Web Services. No OPC-related (or indeed, 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:



## 5.2 Development Models

 QuickOPC offers several development models. Each development model is a kind of approach to solving the problem. Some development models involve procedural programming (coding), while others do not require programming in the traditional sense. The various development models are described in the chapters further below.

The following table shows which development models are available for different target platforms:

	.NET	COM
<b>Procedural Coding Model</b>	yes	yes
<b>Live Binding Model</b>	yes	no
<b>Live Mapping Model</b>	yes	no
<b>Reactive Programming Model</b>	yes	no

Multiple different development models can be freely mixed in the same project.

In QuickOPC-COM, only the procedural coding model is available.

The User Interface object belong logically under the Procedural Coding Model, although they are described separately, because they are simply an extension for a specific purpose, and not all developers need to use them.

### 5.2.1 Procedural Coding Model

In the procedural coding development model, you write code that calls methods on QuickOPC objects. You prepare

and pass in the arguments necessary, and get back the results that you can process further.

This is the traditional model, "natural" to software developers. Its advantage is that you have high level of control over what is happening, and you can structure the code the way you like it. The disadvantage is that you may end up writing more code than it is necessary, compared to other models.

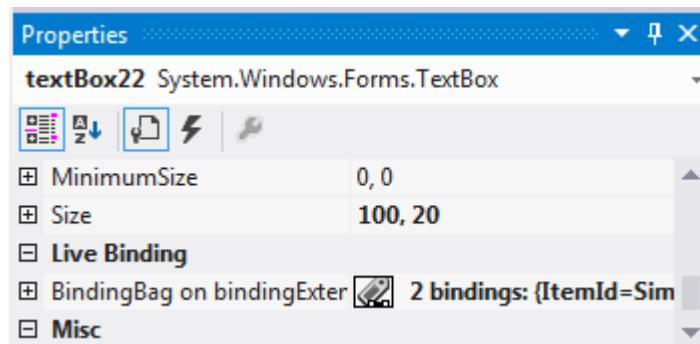
## 5.2.2 Live Binding Model

 With live binding development model, no manual coding is necessary to obtain OPC connectivity. You simply use the Visual Studio Designer to configure bindings between properties of visual or non-visual components, and OPC data. All functionality for OPC reads, writes, subscriptions etc. is provided by the QuickOPC components.

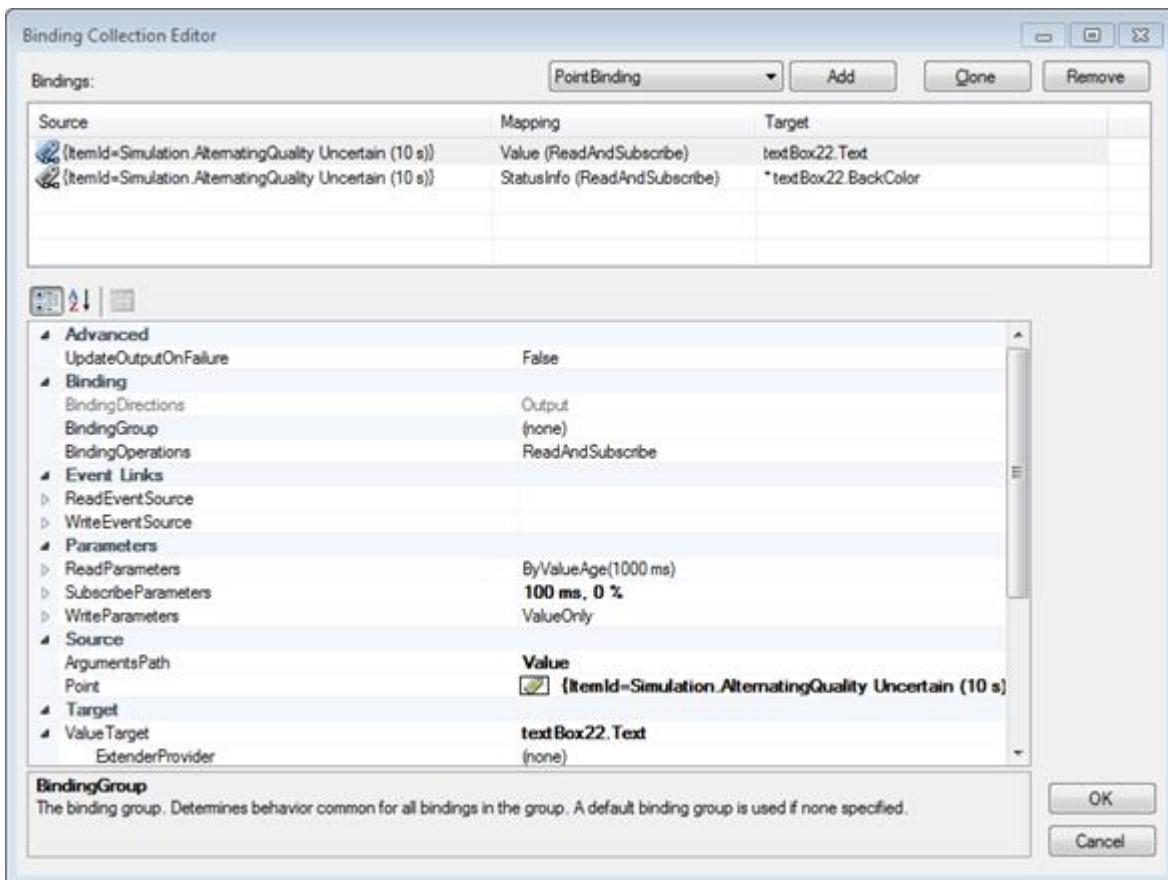
The live binding model is allowed by the [BindingExtender](#), [PointBinder](#) and [DAConnectivity](#) (or [UAConnectivity](#), [CompositeConnectivity](#)) components that you drag from the toolbox to the designer surface. You can then use extender commands in the Properties window, or on the context menu of each component (control), to bind the component's properties to OPC data:



After selecting the "Bind to Point" command, you select the OPC data you want to bind to, and you are done. The currently configured bindings appear as additional entries in the Properties window:



You can also invoke the full-featured binding editor, and configure all aspects on bindings – either on individual component (control), or for a whole container (form) at once:



The Live Binding model is currently available for Windows Forms (or UI-less) containers only. You can use it with OPC Data Access, or with OPC Unified Architecture data.

### 5.2.3 Live Mapping Model



#### Live Mapping Model

The live mapping development model allows you to write objects that correspond logically to a functionality provided by whatever is “behind” the OPC data. For example, if part of your application works with a boiler, it is natural for you to create a boiler object in the code. You then describe how your objects and their members correspond to OPC data – this is done using attributes to annotate your objects. Using Live Mapping methods, you then map your objects to OPC data, and perform OPC operations on them.

For example, when you subscribe to OPC items, incoming OPC item changes can directly set corresponding properties in your objects, without any further coding. You can then focus on writing the code that handles these property changes, abstracting away the fact how they come in.

To give a concrete example, here is a piece of code of a .NET object annotated for live mapping with OPC Data Access, in C#:

#### Live Mapping Annotations

```
[DAType]
class BoilerInputPipe
{
    [DANode]
    public FlowTransmitter FlowTransmitter1 = new FlowTransmitter();

    [DANode]
```

```
public Valve Valve = new Valve();  
  
[DANode, DAItem]  
public bool InAlarm { get; set; };  
}
```

You can use the Liver Mapping model with OPC Data Access, or with OPC Unified Architecture data.

## 5.2.4 Reactive Programming Model

 The reactive programming development model merges the world of Microsoft Reactive Extensions (Rx) for .NET with OPC. The Reactive Extensions (<http://msdn.microsoft.com/en-us/data/gg577609.aspx>) is a library to compose asynchronous and event-based programs using observable collections (data streams) and LINQ-style query operators.

QuickOPC provides OPC-specific classes that implement the Rx interfaces. For example, you can easily create an Rx observable sequence with OPC-DA or OPC-UA item changes, and process it further. Similarly, you can create an Rx observable sequence with OPC-A&E event notifications.

## 5.3 Referencing the Assemblies (.NET)

### 5.3.1 Overview of the Assemblies Available

 Most projects will need:

- “**OPC Labs Base Library Core**” (OpcLabs.BaseLib.dll)
- for OPC Classic: “**OPC Labs EasyOPC “Classic” Library**” (OpcLabs.EasyOpcClassic.dll)
- for OPC UA: “**EasyOPC-UA Library**” (OpcLabs.EasyOpcUA.dll)

Some project will also need other assemblies, such as

- “**OPC Labs Base Library Forms**” (OpcLabs.BaseLibForms.dll)
- “**OPC Labs EasyOPC Forms**” (OpcLabs.EasyOpcForms.dll)

You do not need to (and should not) ever reference the following assemblies:

- **EasyOPC “Classic” Raw Library (x64)** (App\_Web\_OpcLabs.EasyOpcClassicRaw.amd64.dll)
- **EasyOPC “Classic” Raw Library (x86)** (App\_Web\_OpcLabs.EasyOpcClassicRaw.x86.dll)

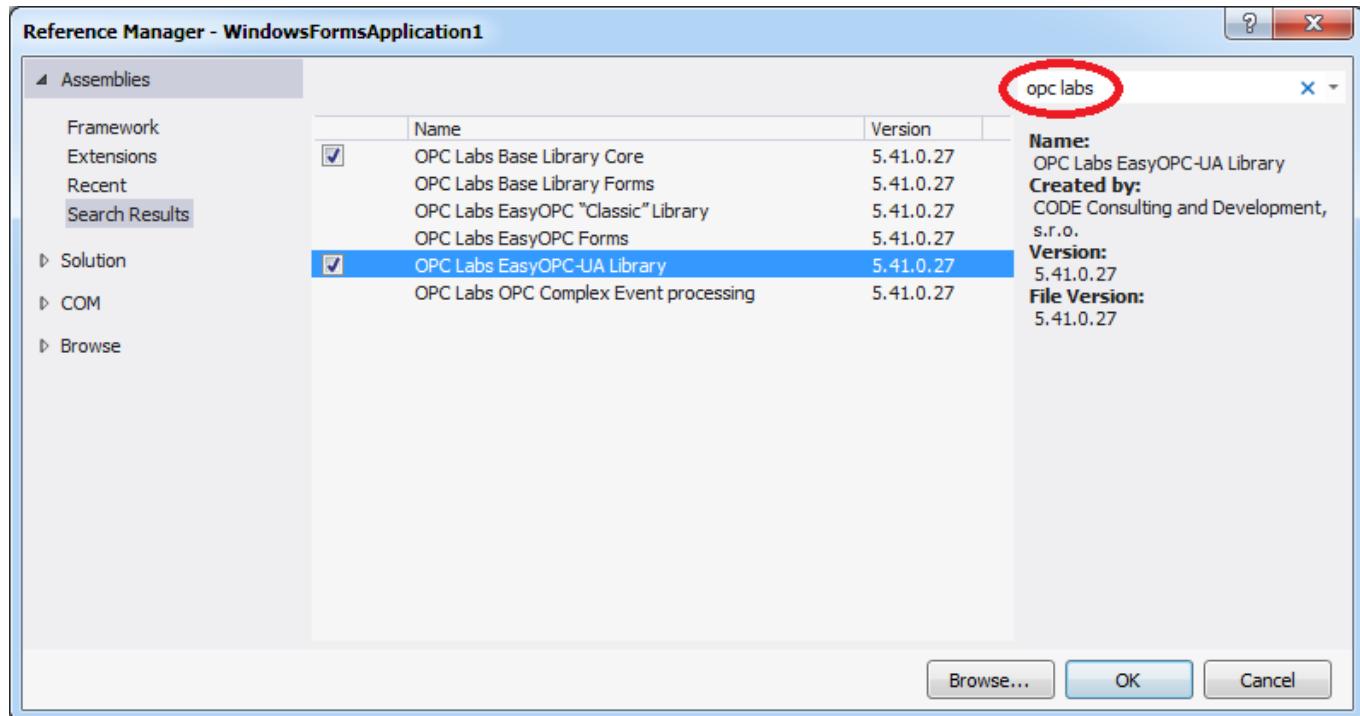
This does not mean that these assemblies are useless or are not used. They are used indirectly, by the relevant assemblies that you do reference, and they will copied over to the target automatically by the build tools.

### 5.3.2 Manual Assembly Referencing

Your application first needs to reference the QuickOPC.NET or QuickOPC-UA 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 "Classic" and QuickOPC-UA assemblies are listed under its Assemblies (or .NET) tab. Select those that you need (see their descriptions in "Product Parts" chapter), and press OK.



You can easily see all QuickOPC assemblies grouped together (as in the picture above), if you type "opc labs" into the search box. You can also sort the list alphabetically by Component Name, by clicking on the corresponding column header; then, scroll to the proper place in the list.

Notes:

- The list content is pulled in background, so you may need to wait a little until all components are listed and you can properly sort them or search for them.
- This documentation is for QuickOPC version 2016.2. This is known as the *external version* of the product. The *internal version* number is 5.41, and that is how the assemblies are listed in the Reference Manager.
- QuickOPC.NET assemblies need the .NET Framework 4.5.2 or later. The dialog filters the components according to the target platform setting of your project. You will not see QuickOPC.NET assemblies listed if your project's target framework is not set to .NET Framework 4.5.2 or later.

## 5.3.3 Assembly Referencing with Visual Studio Toolbox

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 automatically, when you drag the component onto the designer's surface.

## 5.3.4 Assembly Referencing with NuGet

If you have used NuGet to install a QuickOPC package into your project, all necessary assembly references are done for you automatically.

You may, however, sometimes want to \*remove\* certain assembly references later, if you do not need them. For example, the **OpcLabs.QuickOpc** package will automatically reference the assemblies needed for all supported OPC technologies, but if you do not need e.g. OPC "Classic" or OPC Unified Architecture at all, you may remove the

corresponding assembly references from your project.

## 5.3.5 Licensing in Visual Studio

QuickOPC uses, in part, .NET licensing model for component and controls. The internal working of it are largely invisible to you. The visible manifestation of the licensing model in Visual Studio is that the designer creates and maintains a file named **licenses.licx**, which contains a list of the licensed components, together with their assembly and version information. The file is also automatically updated when you upgrade to a newer version of build of QuickOPC. You can safely rely on Visual Studio maintaining this file.

As opposed to other components and controls that use this licensing model in full, QuickOPC license keys do not actually get embedded into your compiled project. The QuickOPC license should always be installed onto the machine separately, using e.h. the License Manager utility of QuickOPC.

## 5.3.6 Namespaces

The QuickOPC "Classic" and QuickOPC-UA class libraries are 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 "Classic" and QuickOPC-UA define types mainly in following namespaces:

Namespace Name	Description
OpcLabs.BaseLib	Contains commonly used fundamental and supporting classes.
OpcLabs.BaseLib.Browsing	Contains classes for generic browsing.
OpcLabs.BaseLib.ComInterop	Provides supporting types for interoperation with the COM technology.
OpcLabs.BaseLib.ComponentModel	Provides classes that are used to implement the run-time and design-time behavior of components and controls.
OpcLabs.BaseLib.Components	Contains concrete generally usable components.
OpcLabs.BaseLib.Data	Contains classes used for data manipulations.
OpcLabs.BaseLib.Forms.Browsing	Contains classes for browsing in Windows Forms.
OpcLabs.BaseLib.Generic	Contains classes that use generics.
OpcLabs.BaseLib.Graphs	Contains classes that work with graphs.
OpcLabs.BaseLib.LiveBinding	Contains components used for live binding.
OpcLabs.BaseLib.LiveMapping	Contains classes used for live mapping of sources to targets.
OpcLabs.BaseLib.Network	Contains classes for networking.
OpcLabs.BaseLib.OperationModel	Contains classes that are used to pass arguments to operations and obtain the operation results.
OpcLabs.EasyOpc	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, OPC Alarms and Events, or OPC Unified Architecture).

Namespace Name	Description
OpcLabs.EasyOpc.AlarmsAndEvents	Contains classes that facilitate easy work with OPC Alarms and Events.
OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace	Contains classes for OPC Alarms & Events address space.
OpcLabs.EasyOpc.AlarmsAndEvents.Engine	Contains classes that are used by the OPC Alarms & Events engine.
OpcLabs.EasyOpc.AlarmsAndEvents.Extensions	Contains extension classes that facilitate easy work with OPC Alarms and Events.
OpcLabs.EasyOpc.AlarmsAndEvents.Forms.Browsing	Contains classes for OPC Alarms & Events browsing in Windows Forms.
OpcLabs.EasyOpc.AlarmsAndEvents.OperationModel	Contains classes that are used to pass arguments to OPC Alarms & Events operations and obtain the operation results.
OpcLabs.EasyOpc.AlarmsAndEvents.Reactive	Contains reactive extensions (Rx) classes for OPC Alarms & Events.
OpcLabs.EasyOpc.DataAccess	Contains classes that facilitate easy work with OPC Data Access.
OpcLabs.EasyOpc.DataAccess.AddressSpace	Contains classes for OPC Data Access address space.
OpcLabs.EasyOpc.DataAccess.Engine	Contains classes that are used by the OPC Data Access engine.
OpcLabs.EasyOpc.DataAccess.Extensions	Contains extension classes that facilitate easy work with OPC Data Access.
OpcLabs.EasyOpc.DataAccess.Forms.Browsing	Contains classes for OPC Data Access browsing in Windows Forms.
OpcLabs.EasyOpc.DataAccess.Generic	Contains classes for OPC Data Access that use generics.
OpcLabs.EasyOpc.DataAccess.LiveBinding	Contains components used for OPC Data Access live binding.
OpcLabs.EasyOpc.DataAccess.LiveMapping	Contains classes used for live mapping of OPC Data Access sources to targets.
OpcLabs.EasyOpc.DataAccess.OperationModel	Contains classes that are used to pass arguments to OPC Data Access operations and obtain the operation results.
OpcLabs.EasyOpc.DataAccess.Reactive	Contains reactive extensions (Rx) classes for OPC Data Access.
OpcLabs.EasyOpc.Engine	Contains classes that are used by the engine for OPC "Classic" specifications (COM/DCOM, and XML based).
OpcLabs.EasyOpc.Forms.Browsing	Contains classes for browsing using OPC "Classic" specifications in Windows Forms.
OpcLabs.EasyOpc.LiveMapping	Contains classes used for live mapping of OPC "Classic" sources to targets.
OpcLabs.EasyOpc.OperationModel	Contains classes that are used to pass arguments to OPC

Namespace Name	Description
	"Classic" (COM/DCOM and XML based) operations and obtain the operation results.
OpcLabs.EasyOpc.SwtbExtenderReplacement	Contains classes that serve as a replacement for Software Toolbox Extender ( <a href="http://www.opcextender.net">www.opcextender.net</a> ) component.
OpcLabs.EasyOpc.UA	Contains classes that facilitate easy work with OPC Unified Architecture.
OpcLabs.EasyOpc.UA.AddressSpace	Contains classes for OPC Unified Architecture (OPC-UA) address space.
OpcLabs.EasyOpc.UA.AlarmsAndConditions	Contains classes for OPC Unified Architecture (OPC-UA) Alarms & Conditions.
OpcLabs.EasyOpc.UA.Discovery	Contains classes for OPC Unified Architecture (OPC-UA) discovery.
OpcLabs.EasyOpc.UA.Engine	Contains classes that are used by the engine for OPC Unified Architecture specification (OPC-UA).
OpcLabs.EasyOpc.UA.Extensions	Contains extension classes that facilitate easy work with OPC Unified Architecture (OPC-UA).
OpcLabs.EasyOpc.UA.Filtering	Provides classes and interfaces that provide filtering capabilities in OPC Unified Architecture.
OpcLabs.EasyOpc.UA.Forms.Browsing	Contains classes that facilitate OPC Unified Architecture (OPC-UA) browsing from Windows Forms applications.
OpcLabs.EasyOpc.UA.Generic	Contains classes for OPC Unified Architecture (OPC-UA) that use generics.
OpcLabs.EasyOpc.UA.Graphs	Contains classes for graph-related operations in OPC-UA.
OpcLabs.EasyOpc.UA.LiveBinding	Contains components used for OPC Unified Architecture (OPC-UA) live binding.
OpcLabs.EasyOpc.UA.LiveMapping	Contains classes used for live mapping of OPC Unified Architecture (OPC-UA) sources to targets.
OpcLabs.EasyOpc.UA.OperationModel	Contains classes that are used to pass arguments to OPC Unified Architecture (OPC-UA) operations and obtain the operation results.
OpcLabs.EasyOpc.UA.Parsing	Contains parsing classes for OPC Unified Architecture (OPC-UA).
OpcLabs.EasyOpc.UA.Reactive	Contains reactive extensions (Rx) classes for OPC Unified Architecture (OPC-UA).

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.

## 5.4.1 COM Components for OPC Classic

In order to gain access to all COM objects of QuickOPC "Classic", you need types described in the following type libraries:

Type Library Name	LIBID	Note
OPC Labs Base Library Core	ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9	
OPC Labs Base Library Forms	A0D7CA1E-7D8C-4D31-8ECB-84929E77E331	only for User Interface objects
OPC Labs EasyOPC "Classic" Library	1F165598-2F77-41C8-A9F9-EAF00C943F9F	
OPC Labs EasyOPC Forms	2C654FA0-6CD6-496D-A64E-CE2D2925F388	only for User Interface objects

For your convenience, we have listed below examples of source code reference statements for the "OPC Labs EasyOPC "Classic" Library" in various languages and tools.

Language/tool	Statement
C++	#import"libid:1F165598-2F77-41C8-A9F9-EAF00C943F9F" // OpcLabs.EasyOpcClassic
WSH	<referenceguid="{1F165598-2F77-41C8-A9F9-EAF00C943F9F}" /> <!--OpcLabs.EasyOpcClassic-->
ASP	<!--METADATA TYPE="TypeLib" NAME="OPC Labs Easy "Classic" Library" UUID="{1F165598-2F77-41C8-A9F9-EAF00C943F9F}" -->

## 5.4.2 COM Components for OPC UA

In order to gain access to all COM objects of QuickOPC-UA, you need types described in the following type libraries:

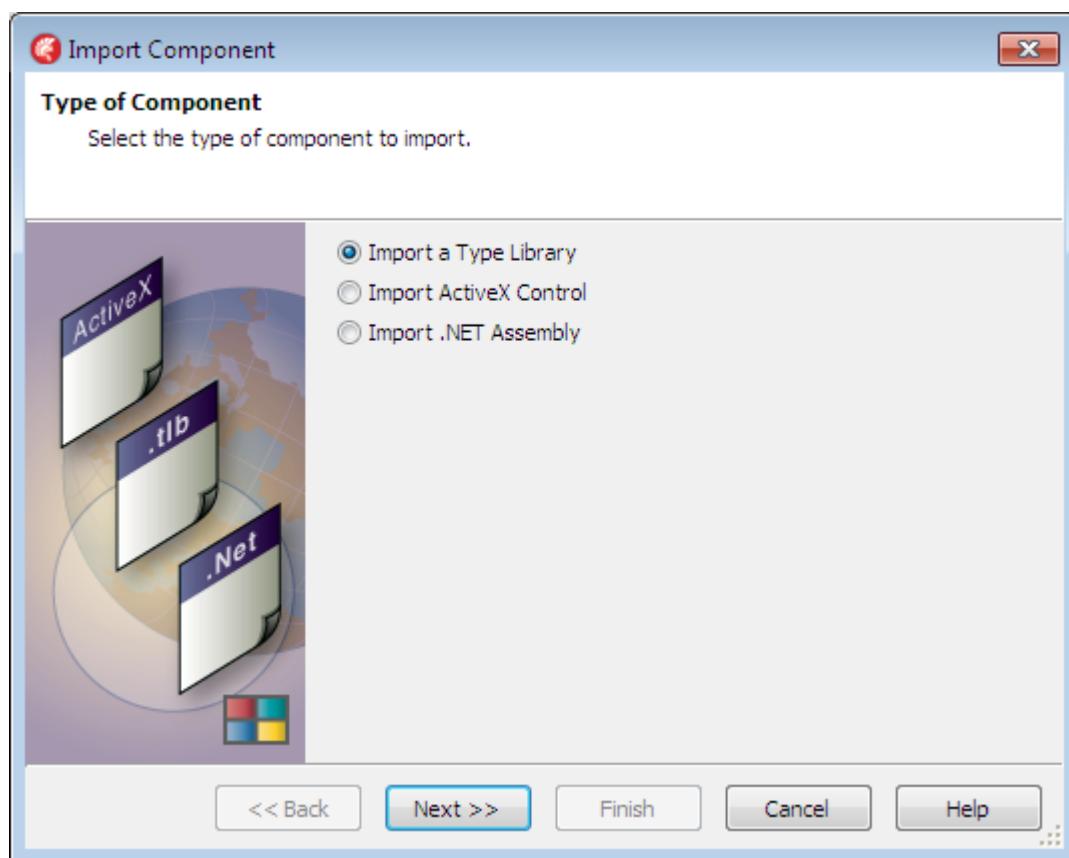
Type Library Name	LIBID	Note
OPC Labs Base Library Core	ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9	
OPC Labs Base Library Forms	A0D7CA1E-7D8C-4D31-8ECB-84929E77E331	only for User Interface objects
OPC Labs EasyOPC "Classic" Library	1F165598-2F77-41C8-A9F9-EAF00C943F9F	
OPC Labs EasyOPC-UA Library	E15CAAE0-617E-49C6-BB42-B521F9DF3983	
OPC Labs EasyOPC Forms	2C654FA0-6CD6-496D-A64E-CE2D2925F388	only for User Interface objects

## 5.4.3 Importing Type Libraries to Delphi

Use the steps below to import QuickOPC-UA for COM type libraries to Delphi XE7, Delphi XE8 or Delphi 10 Seattle.

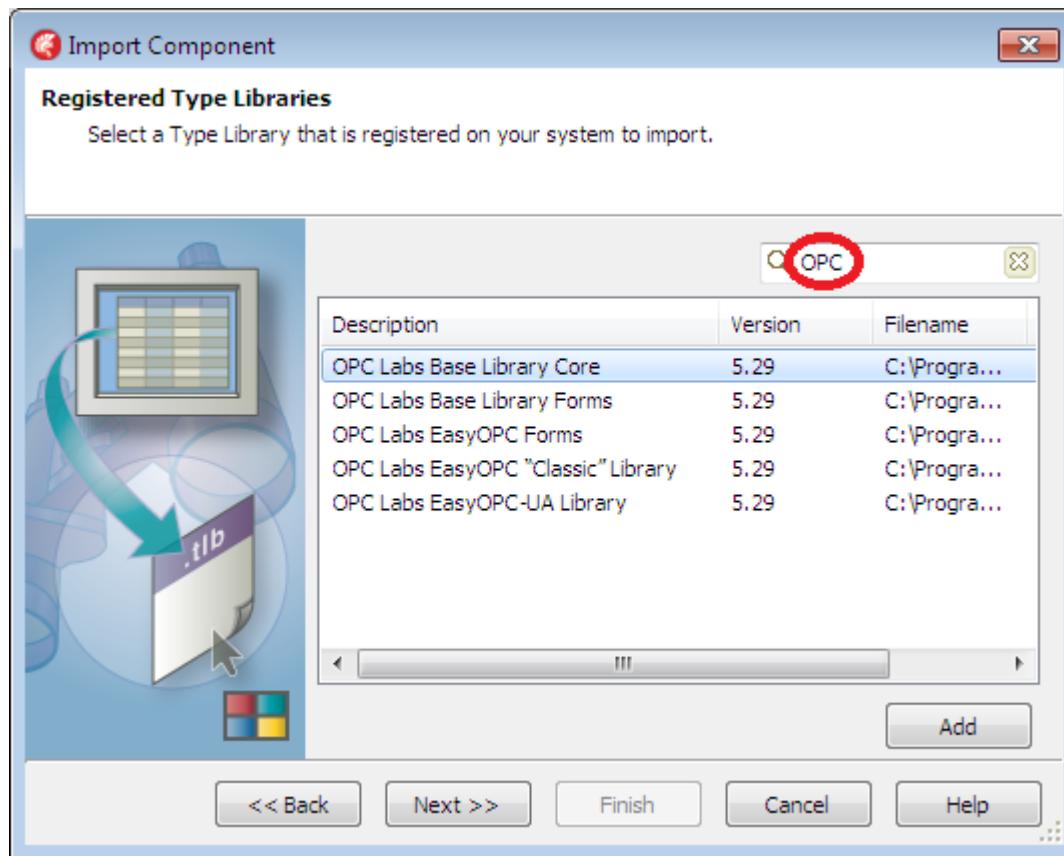
Note, however, that for the purpose of our examples, we already ship Delphi (XE7) components created from importing QuickOPC type libraries together with the examples. It may be faster and safer to simply reuse them. They are located in the "Imports" subdirectory under the Delphi examples directory  
**(ExamplesCom/OP/DelphiXE7/Imports** relatively from the QuickOPC installation folder).

1. In Delphi, select command **Component -> Import Component**. The first page of the "Import Component" wizard appears, titled "Type of Component":



Select "Import a Type Library" and press the Next button.

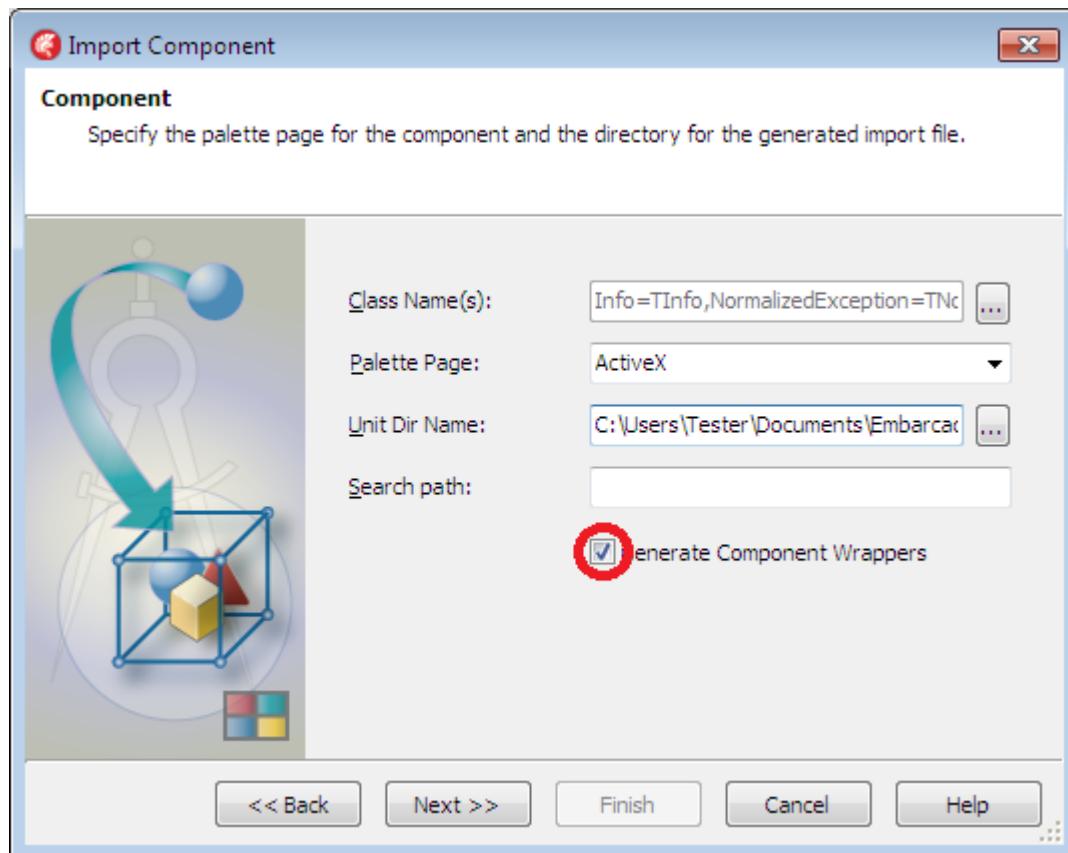
2. The page "Registered Type Libraries" appears:



You may need to resize the form to make it larger, and/or drag the column header dividers to make the columns wider.

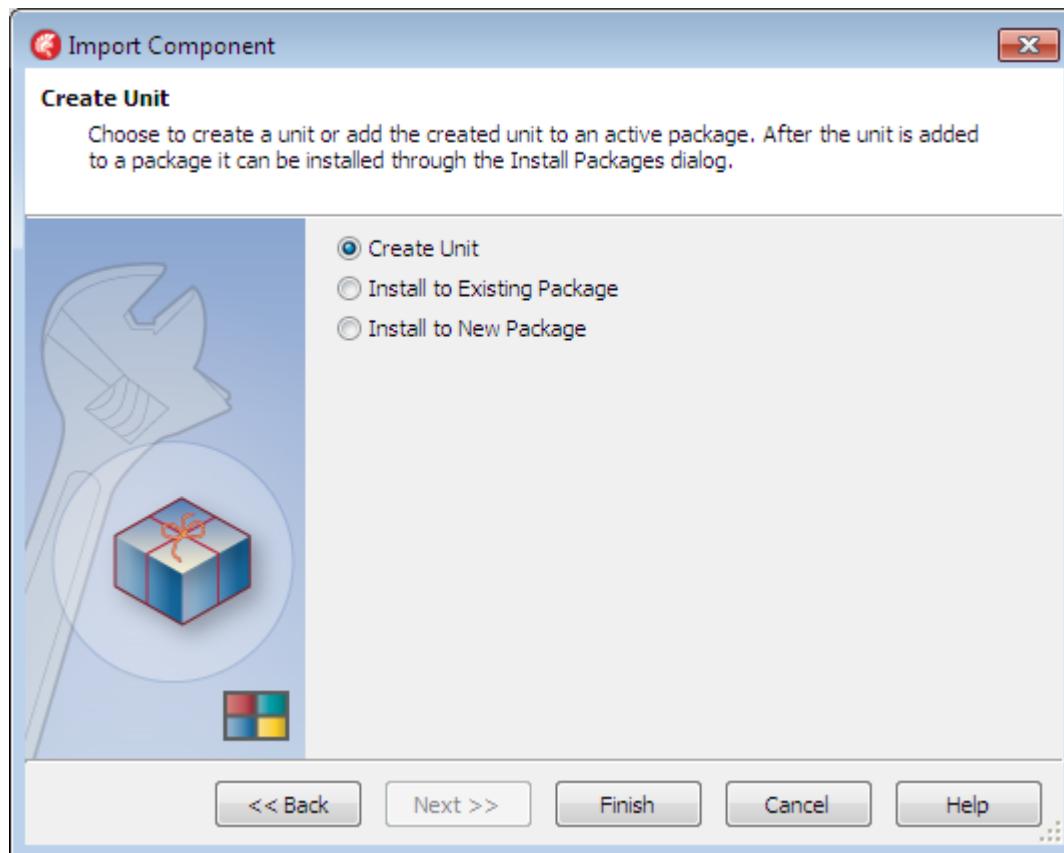
Type "OPC" into the search box, in order to narrow down the list of available type libraries. Select the type library you want to import, and press the Next button.

3. The page "Component" appears:



Check the "Generate Component Wrappers" box, and optionally modify the other settings as needed. Then, press the "Next button".

4. The page "Create Unit" appears:



Select the option you need (typically, you can leave the "Create Unit" choice selected), and press the Finish button.

5. If you need to import another type library, repeat the process from Step 1.

Note that importing a type library may also silently regenerate (and overwrite) an existing imported component that the type library depends on, but then without the "Generate Component Wrappers" option – potentially messing things up. To avoid this, you can either

- a. make backup copies of existing components before importing any new type library, and restore the overwritten components afterwards, if needed; or
- b. import the type libraries in the reverse order of layering – start with those that have "Forms" in their name, because they are at the top of the hierarchy, proceed to the remaining "EasyOpc..." type libraries, and import the "BaseLib..." as the last; or
- c. if possible, simply use the pre-imported components that come with our examples, as described above.

## 5.5 Conventions

With the help of various conventions, we try to make it easier to find and discover the information needed, and understand the components' internal consistency.

### 5.5.1 Date and Time Conventions

All OPC-related properties that are of `DateTime` type (and are consistently in UTC) have corresponding properties with a "`Local`" postfix, which contain the same value but expressed in local time. For example, `DAVtq.Timestamp` contains a timestamp in UTC, whereas its `DAVtq.TimestampLocal` contains the same in local time.

## 5.5.2 Naming Conventions

### 5.5.2.1 Common Naming Conventions

In OPC-UA, the name part “**MonitoredItem**” is used wherever the type or member applies to both Data Access and Alarms and Conditions, whereas the names that use the term “**DataChange**” apply to Data Access only.

### 5.5.2.2 Type Names

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.
- Types which are specific to OPC Unified Architecture start with **UA** (or **EasyUA**) prefix. Types that are shared among multiple OPC specifications do not have these prefixes.

Note that the second and third conventions work in addition and together with the fact that there are separate **DataAccess**, **AlarmsAndEvents**, and **UA** 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.

### 5.5.2.3 Interface Names in COM



As opposed to .NET, every object access in COM is solely made over its interfaces. In COM, after the object is instantiated, you never “see” the object as such – only the interfaces it has. Therefore, at least one additional interface (besides the COM-defined IUnknown and IDispatch) is needed on each .NET exposed to COM.

We consistently use following conventions in QuickOPC-COM and QuickOPC-UA for COM:

- Incoming interfaces have a form of [XXXX](#), where **XXXX** is the distinguishing part, e.g. [EasyUAClient](#), for the [EasyUAClient](#) object.
- Outgoing interface (for events) have a form of [DXXXXEvents](#), where **XXXX** is the distinguishing part, e.g. [DEasyUAClientEvents](#), for the [EasyUAClient](#) object.

The incoming interface are dual interfaces. The outgoing interfaces are dispatch-only interfaces, hence the use of ‘D’ as a prefix.

### 5.5.2.4 ProgIDs (COM)

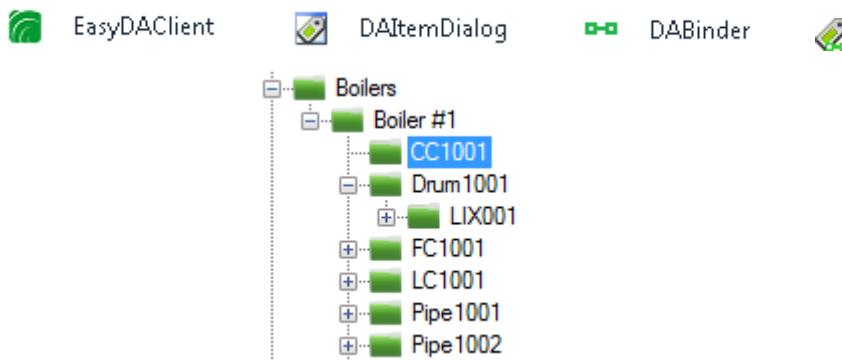


In QuickOPC-COM and QuickOPC-UA for COM, all objects have ProgIDs that are identical to their qualified type names in .NET. For example, the main EasyUAClient component resides in the [OpcLabs.EasyOpc.UA](#) namespace, and its fully qualified name and therefore also its ProgID is “[OpcLabs.EasyOpc.UA.EasyOpcUA](#)”. The main EasyDAClient component resides in the [OpcLabs.EasyOpc.DataAccess](#) namespace, and its fully qualified name and therefore also its ProgID is “[OpcLabs.EasyOpc.DataAccess.EasyOpcDA](#)”. This convention makes it easy to instantiate COM objects in QuickOPC, based on their .NET counterparts.

## 5.5.3 Coloring Conventions

QuickOPC maintains visual consistency by using specific colors for certain functionality areas. This is typically seen in various icons, images and bitmaps.

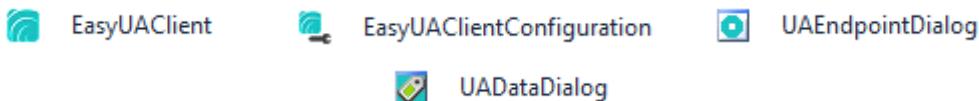
Green color is used for OPC Data Access. For example:



Orange color is used for OPC Alarms&Events. For example:



Turquoise color is used for OPC Unified Architecture. Because OPC-UA supports multiple functionalities, the turquoise color is sometimes used in combination with other colors, such as green for Data, or orange for Events. For example:

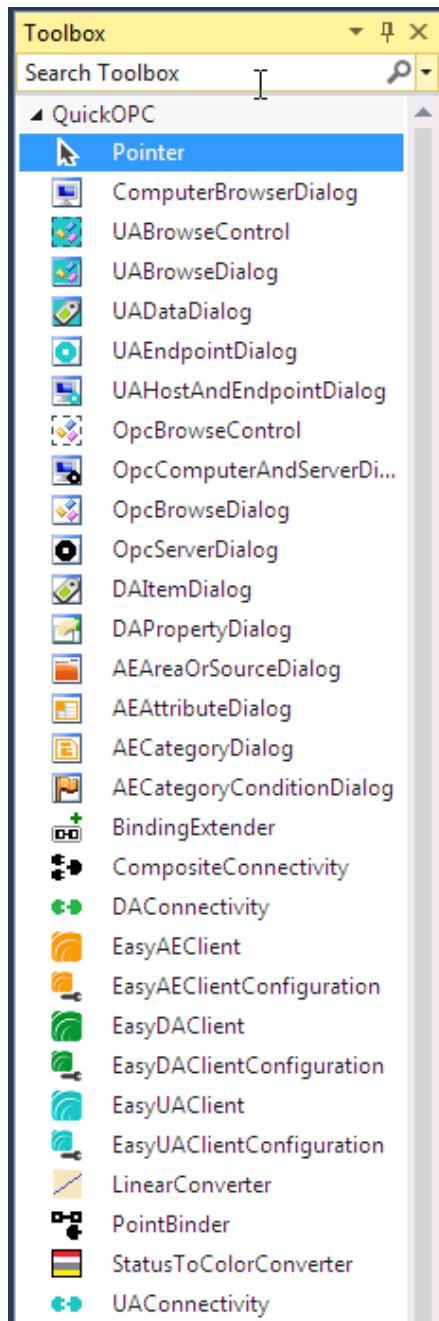


## 5.6 Components and Objects

 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.

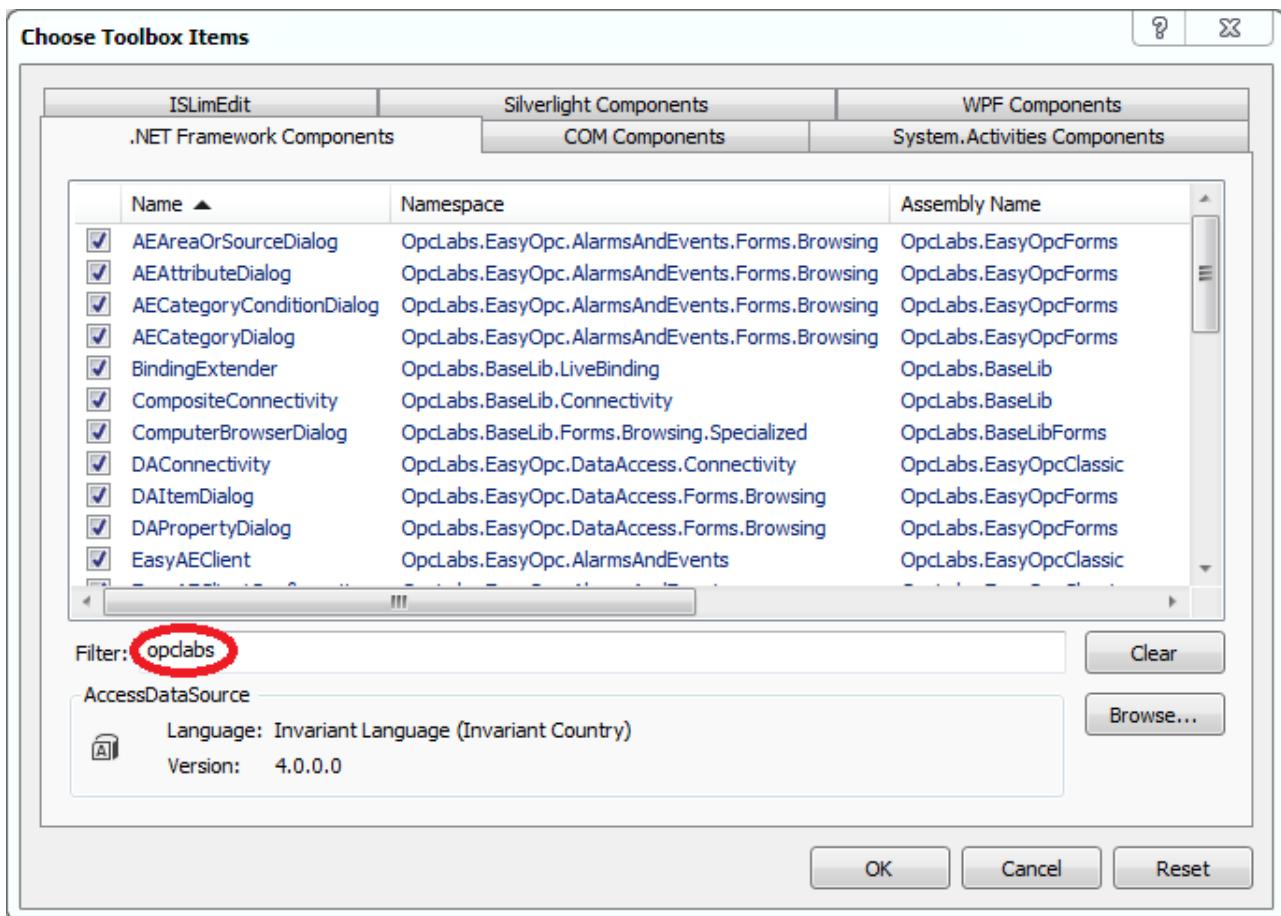
If you are working with an IDE (Integrated Development Environment) such as Visual Studio, the QuickOPC components appear in its Toolbox (see example picture below). Note that, however, most QuickOPC objects can be used directly from the code, and that the Toolbox only offers the components for certain environments, such as Windows Forms; in other environments, you can still use QuickOPC, but you will instantiate the components directly. Also note that there are many more objects to QuickOPC than just those classified as designer components and shown on the Toolbox.

The QuickOPC components in the Toolbox look similar to this:



If, for some reason, some or all of the QuickOPC components do not appear in the toolbox, here are several hints to troubleshoot the problem:

- Make sure you have a document open in the editor that allows placing components to it – such as the Windows form (alternatively, "Show All" can be selected in the Toolbox).
- Make sure that your project targets the right platform. Commonly, if you do not see the right components, it means that your project is not targeting the .NET Framework version 4.5.2 or later.
- Perform a "Reset Toolbox" command (right-click in the toolbox area first), and restart Visual Studio.
- If everything else fails, you can add the QuickOPC components to the Toolbox manually (you only need to do this once): Right-click on the Toolbox, and select "Choose Items...".



In the "Choose Toolbox Items" dialog, select ".NET Framework Components" tab, and type "opclabs" into the Filter field.

Then, check all boxes next to the displayed components, and press OK. The component icons 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 (usually, a Windows Form) in the designer, or when you check "Show All" in the Toolbox context menu.

You can create a new toolbox tab named "QuickOPC", and move the component icons into this tab.



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.

## 5.6.1 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 [EasyAECClient](#) object.
- For OPC Unified Architecture, it is the [EasyUAClient](#) 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 [EasyAECClient](#) or [EasyUAClient](#) object are similar.

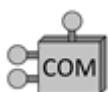
In order to be able to use the [EasyDAClient](#), [EasyAECClient](#) or [EasyUAClient](#) object, you need to instantiate it first.



In QuickOPC "Classic" and QuickOPC-UA, there are two methods that you can use:

- Write the code that creates a new instance of the object, using an 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](#), [EasyAECClient](#) and [EasyUAClient](#) objects are 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, in Console applications 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.

Note: In addition, there is a static [Create\(\)](#) method on each [EasyXXClient](#) class. This methods works the same as parameterless constructor. For example (in C#), [EasyUAClient.Create\(\)](#) is the same as `new EasyUAClient()`.



For QuickOPC-COM and QuickOPC-UA for COM, the following table contains information needed to instantiate the objects.

Class Name	CLSID	ProgID
<a href="#">EasyAECClient</a>	3643545B-221F-4960-BF47-8A4DDEC81A67	OpcLabs.EasyOpc.AlarmsAndEvents.EasyOpcAE
<a href="#">EasyDAClient</a>	6B0B5307-BCB6-4953-A832-BFCF952F7561	OpcLabs.EasyOpc.DataAccess.EasyOpcDA
<a href="#">EasyUAClient</a>	54AFB0EA-9809-4D1D-AFBE-0EC164C59A45	OpcLabs.EasyOpc.UA.EasyOpcUA

As described in the "Naming Conventions", in QuickOPC-COM and QuickOPC-UA for COM, all objects have ProgIDs that are identical to their qualified type names in .NET. For example, the ProgID of the main [EasyUAClient](#) is therefore "OpcLabs.EasyOpc.UA.EasyOpcUA".

For COM development, CLSIDs of all objects and IIDs of all interfaces can also be found in the Reference documentation (look for the [GuidAttribute](#) on the classes and interfaces).

## 5.6.1.1 Interfaces and Extension Methods

In QuickOPC.NET, the main methods constituting the functionality of [EasyDAClient](#), [EasyAECClient](#), and [EasyUAClient](#) components are actually implementations of [IEasyDAClient](#), [IEasyAECClient](#), and [IEasyUAClient](#) interfaces, respectively. The remaining methods and method overloads, which simply build upon the core interface methods, are implemented as extension methods on the interface. Also, at many places, arguments and properties that accept one of the [IEasyXXClient](#) interfaces, instead of a concrete component.

In most languages (certainly in C# and Visual Basic), this design leads to the same syntax as if all the methods were

implemented directly on the core concrete object. The approach that we have chosen allows to supply a different implementation of the component's interface where needed, and is currently used for simulation purposes in the browsing controls and dialogs.

## 5.6.1.2 Isolated Clients

Normally, instances of a client object (such as [EasyDAClient](#), [EasyAECClient](#), [EasyUAClient](#)) act commonly, sharing the same connections to the target OPC servers, and also many common parameters. This way, you can create a large number of these instances and work with them in an easy way, without having to worry about negative effects on the target. If, for example, you create two instances of an [EasyDAClient](#) object, and subscribe to some OPC items in each of these instances, only one connection will be created to the OPC server.

This approach works well for most applications, and it allows easy coding in scenarios such as Web development, where each page request may require OPC operations, and the requests may be coming in quick succession and in large numbers, and even processed in parallel.

In some cases, however, you may want to have a dedicated connection to the OPC server, or have more control over the parameters of the connection. In such case, you can set the [Isolated](#) property of the [EasyXXClient](#) object to 'true'. By doing so, operations invoked on this instance of the client object will work with their own connection to the target OPC server. We say that the client (object) is *isolated*.

If you create more such isolated client objects, each of them will work with its own connection. Obviously you need to be more careful with this approach, in order to keep the load on the target OPC server in reasonable limits.

## 5.6.1.3 Shared Instance

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

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

If you plan to use events on the [EasyXXClient](#) component, the shared 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 shared instance for library code (if you are developing a Class Library project), 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 expect.

In QuickOPC-COM and QuickOPC-UA for COM, you can access the shared instance through the [EasyXXClientConfiguration](#) object.

## 5.6.2 User Interface Objects



In QuickOPC "Classic" and QuickOPC-UA, 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.



In QuickOPC-COM and QuickOPC-UA for COM, you can use some of the user interface objects (namely, the dialogs – but not the controls) as well. Create the objects as any other QuickOPC objects on the COM platforms. The convention described earlier (with computational objects) always applies, and therefore the ProgIDs of the user interface objects are the same as the namespace-qualified names of corresponding .NET types.

## 5.7 Connection-less 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 connection-less (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](#), [EasyAECClient](#) or [EasyUAClient](#) 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).

## 5.8 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 (or, [ReadValue](#) method on the [EasyUAClient](#) object). There is also a corresponding method named [ReadMultipleItemValues](#) on the [EasyDAClient](#) object (or, [ReadMultipleValues](#) on the [EasyUAClient](#) object) 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.

## 5.9 Event Pull Mechanism

*Event pull mechanism* is an alternative to the traditional delivery mechanism of notifications through event handlers. Setting up event handlers, and properly handling events, is difficult or impossible in some (especially COM-based) tools (e.g. Python) or in some environments (e.g. applications running under Web server). In COM, unless taken care of already by the tool, the developer must also assure that (in an STA model) Windows messages are being pumped by the application.

Note: This feature should only be used when the traditional event delivery approach is not feasible. It is unlikely that

you will need to use the event pull mechanism in C# or VB.NET (except maybe for server-side of certain Web apps and Web services).

*Event pull mechanism* replaces the event handlers by methods that allow the developer to obtain a next available notification from an internally provided queue. Events can then be handled by a loop that calls the “pull” method and processes the notifications. In order to maintain reasonable performance, a so-called long poll approach is used: The method only returns after a notification becomes available, or a specified timeout elapses. Note that the event pull concept is strictly confined to how the QuickOPC exchanges notifications with the enclosing application, and is not related to how QuickOPC communicates with OPC servers.

For each event **XXXX**, by convention, two methods for event pull mechanism are provided: A [PullXXXX](#) method, which allows to retrieve a single notification, and a [PullMultipleXXXX](#) method, which allows to retrieve multiple notifications, if they are available. Although currently is no significant performance difference between the two methods, it is recommended that your code uses the [PullMultipleXXXX](#) methods, because it may bring performance benefits in the future.

The following table shows the traditional events, and their corresponding methods for event pull:

Type	Event	Pull method	PullMultiple method
<a href="#">EasyDAClient</a>	<a href="#">ItemChanged</a>	<a href="#">PullItemChanged</a>	<a href="#">PullMultipleItemChanges</a>
<a href="#">EasyAECClient</a>	<a href="#">Notification</a>	<a href="#">PullNotification</a>	<a href="#">PullMultipleNotifications</a>
<a href="#">EasyUAClient</a>	<a href="#">DataChangeNotification</a>	<a href="#">PullDataChangeNotification</a>	<a href="#">PullMultipleDataChangeNotifications</a>
	<a href="#">EventNotification</a>	<a href="#">PullEventNotification</a>	<a href="#">PullMultipleEventNotifications</a>
	<a href="#">ServerConditionChanged</a>	<a href="#">PullServerConditionChanged</a>	<a href="#">PullMultipleServerConditionChanges</a>

Each [PullXXXX](#) or [PullMultipleXXXX](#) method takes an argument that specifies a timeout (in milliseconds). When a notification is available, the methods returns it immediately, and return a non-null object containing the event arguments (or an array of such event arguments, for [PullMultipleXXXX](#)). If no notifications are available during the specified period (timeout), the methods returns after the period elapses, but the return value is a null reference.

After your code receives a non-null event arguments object, it can process it as it would do in a traditional event handler.

You will probably be calling the [PullXXXX](#) or [PullMultipleXXXX](#) method in a loop with some termination condition. Specify a timeout that will allow the loop to react to the termination condition quickly enough. You should not, however, use very low (or zero) timeouts in such a loop, because that would lead to unnecessary high CPU consumption.

In order to use the event pull mechanism, your code needs to set up a queue with a fixed capacity upfront. This is done by setting a corresponding property in the [EasyXXClient](#) object. By convention, for an event named **XXXX**, the property name is [PullXXXXQueueCapacity](#). For example, set the [EasyUAClient.PullDataChangeNotificationQueueCapacity](#) property to allocate a queue for OPC UA data change notifications.

If you do not set the queue capacity to a positive (non-zero) value, and attempt to use the event pull mechanism, the [PullXXXX](#) or [PullMultipleXXXX](#) method will throw an [InvalidOperationException](#).

Examples for OPC Data Access:

## JScript

```
// This example shows how to subscribe to item changes and obtain the events by
// pulling them.

var Client = new ActiveXObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient");
// In order to use event pull, you must set a non-zero queue capacity upfront.
```

```
Client.PullItemChangedQueueCapacity = 1000;

WScript.Echo("Subscribing item changes...");
Client.SubscribeItem("", "OPCLabs.KitServer.2", "Simulation.Random", 1000);

WScript.Echo("Processing item changes for 1 minute...");
var endTime = new Date().getTime() + 60*1000
do {
    var EventArgs = Client.PullItemChanged(2*1000);
    if (EventArgs !== null) {
        // Handle the notification event
        WScript.Echo(EventArgs);
    }
} while(new Date().getTime() < endTime);

WScript.Echo("Unsubscribing item changes...");
Client.UnsubscribeAllItems();

WScript.Echo("Finished.");
```

## Object Pascal

---

```
// This example shows how to subscribe to item changes and obtain the events by
// pulling them.

class procedure PullItemChanged.Main;
var
  Client: TEasyDAClient;
  EndTick: Cardinal;
  EventArgs: _EasyDAItemChangedEventArgs;
begin
  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);
  // In order to use event pull, you must set a non-zero queue capacity upfront.
  Client.PullItemChangedQueueCapacity := 1000;

  WriteLn('Subscribing item changes...');
  Client.SubscribeItem('', 'OPCLabs.KitServer.2', 'Simulation.Random', 1000);

  WriteLn('Processing item changes for 1 minute...');
  EndTick := Ticks + 60*1000;
  while Ticks < EndTick do
  begin
    EventArgs := Client.PullItemChanged(2*1000);
    if EventArgs <> nil then
      // Handle the notification event
      WriteLn(EventArgs.ToString());
  end;

  WriteLn('Unsubscribing item changes...');
  Client.UnsubscribeAllItems();

  WriteLn('Finished.');
end;
```

## PHP

---

```
// This example shows how to subscribe to item changes and obtain the events by
// pulling them.

$Client = new COM("OpcLabs.EasyOpc.DataAccess.EasyDAClient");
```

```
// In order to use event pull, you must set a non-zero queue capacity upfront.  
$Client->PullItemChangedQueueCapacity = 1000;  
  
print "Subscribing item changes...\n";  
$Client->SubscribeItem("", "OPCLabs.KitServer.2", "Simulation.Random", 1000);  
  
print "Processing item changed events for 1 minute...\n";  
$endTime = time() + 60;  
do {  
    $EventArgs = $Client->PullItemChanged(2*1000);  
    if (!is_null($EventArgs)) {  
        // Handle the notification event  
        print $EventArgs->ToString();  
        print "\n";  
    }  
} while (time() < $endTime);
```

## Python

```
# This example shows how to subscribe to item changes and obtain the events by  
pulling them.  
  
import time  
import win32com.client  
  
# Instantiate the client object  
client = win32com.client.Dispatch('OpcLabs.EasyOpc.DataAccess.EasyDAClient')  
# In order to use event pull, you must set a non-zero queue capacity upfront.  
client.PullItemChangedQueueCapacity = 1000  
  
print('Subscribing item changes...')  
client.SubscribeItem('', 'OPCLabs.KitServer.2', 'Simulation.Random', 1000)  
  
print('Processing item changes for 1 minute...')  
endTime = time.time() + 60  
while time.time() < endTime:  
    eventArgs = client.PullItemChanged(2*1000)  
    if eventArgs is not None:  
        # Handle the notification event  
        print(eventArgs)  
  
print('Unsubscribing item changes...')  
client.UnsubscribeAllItems()  
  
print('Finished.')
```

## VBScript

```
Rem This example shows how to subscribe to item changes and obtain the events by  
pulling them.
```

### Option Explicit

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")  
' In order to use event pull, you must set a non-zero queue capacity upfront.  
Client.PullItemChangedQueueCapacity = 1000  
  
WScript.Echo "Subscribing item changes..."  
Client.SubscribeItem "", "OPCLabs.KitServer.2", "Simulation.Random", 1000  
  
WScript.Echo "Processing item changes for 1 minute..."
```

```
Dim endTime: endTime = Now() + 60*(1/24/60/60)
Do
    Dim EventArgs: Set EventArgs = Client.PullItemChanged(2*1000)
    If Not (EventArgs Is Nothing) Then
        ' Handle the notification event
        WScript.Echo EventArgs
    End If
Loop While Now() < endTime

WScript.Echo "Unsubscribing item changes..."
Client.UnsubscribeAllItems

WScript.Echo "Finished."
```

Examples for OPC Alarms&Events:

## Object Pascal

---

```
// This example shows how to subscribe to events and obtain the notification events
// by pulling them.

class procedure PullNotification.Main;
var
    Client: TEasyAEClient;
    EndTick: Cardinal;
    EventArgs: _EasyAENotificationEventArgs;
    Handle: Integer;
    ServerDescriptor: _ServerDescriptor;
    State: OleVariant;
    SubscriptionParameters: _AESubscriptionParameters;
begin
    ServerDescriptor := CoServerDescriptor.Create;
    ServerDescriptor.ServerClass := 'OPCLabs.KitEventServer.2';

    // Instantiate the client object
    Client := TEasyAEClient.Create(nil);
    // In order to use event pull, you must set a non-zero queue capacity upfront.
    Client.PullNotificationQueueCapacity := 1000;

    WriteLn('Subscribing events...');
    SubscriptionParameters := CoAESubscriptionParameters.Create;
    SubscriptionParameters.NotificationRate := 1000;
    Handle := Client.SubscribeEvents(ServerDescriptor, SubscriptionParameters, true,
    State);

    WriteLn('Processing event notifications for 1 minute...');
    EndTick := Ticks + 60*1000;
    while Ticks < EndTick do
    begin
        EventArgs := Client.PullNotification(2*1000);
        if EventArgs <> nil then
            // Handle the notification event
            WriteLn(EventArgs.ToString());
    end;

    WriteLn('Unsubscribing events...');
    Client.UnsubscribeEvents(Handle);

    WriteLn('Finished.');
end;
```

## VBScript

Rem This example shows how to subscribe to events and obtain the notification events by pulling them.

```
Option Explicit

Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
' In order to use event pull, you must set a non-zero queue capacity upfront.
Client.PullNotificationQueueCapacity = 1000

WScript.Echo "Subscribing events..."
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = Client.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

WScript.Echo "Processing event notifications for 1 minute..."
Dim endTime: endTime = Now() + 60*(1/24/60/60)
Do
    Dim EventArgs: Set EventArgs = Client.PullNotification(2*1000)
    If Not (EventArgs Is Nothing) Then
        ' Handle the notification event
        WScript.Echo EventArgs
    End If
Loop While Now() < endTime

WScript.Echo "Unsubscribing events..."
Client.UnsubscribeEvents handle

WScript.Echo "Finished."
```

Examples for OPC UA (Data):

## C#

```
// This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.
using OpcLabs.EasyOpc.UA;
using OpcLabs.EasyOpc.UA.OperationModel;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class PullDataChangeNotification
        {
            public static void Main()
            {
                // Instantiate the client object
                // In order to use event pull, you must set a non-zero queue
capacity upfront.
                var easyUAClient = new EasyUAClient {
```

```
PullDataChangeNotificationQueueCapacity = 1000 };

        Console.WriteLine("Subscribing...");
        easyUAClient.SubscribeDataChange(
            "http://opcua.demo-this.com:51211/UA/SampleServer", // or
"opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
            "nsu=http://test.org/UA/Data/;i=10853",
            1000);

        Console.WriteLine("Processing data change events for 1 minute...");
        int endTick = Environment.TickCount + 60 * 1000;
        do
        {
            EasyUADataChangeNotificationEventArgs eventArgs =
easyUAClient.PullDataChangeNotification(2 * 1000);
            if (eventArgs != null)
                // Handle the notification event
                Console.WriteLine(eventArgs);
        } while (Environment.TickCount < endTick);
    }
}
}
```

## VB.NET

---

```
' This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class PullDataChangeNotification
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()
            ' In order to use event pull, you must set a non-zero queue capacity
upfront.
            easyUAClient.PullDataChangeNotificationQueueCapacity = 1000

            Console.WriteLine("Subscribing...")
            easyUAClient.SubscribeDataChange(
                "http://opcua.demo-this.com:51211/UA/SampleServer",
                "nsu=http://test.org/UA/Data/;i=10853",
                1000) ' or "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"

            Console.WriteLine("Processing data change events for 1 minute...")
            Dim endTick As Integer = Environment.TickCount + 60 * 1000
            Do
                Dim eventArgs As EasyUADataChangeNotificationEventArgs =
easyUAClient.PullDataChangeNotification(2 * 1000)
                If Not eventArgs Is Nothing Then
                    ' Handle the notification event
                    Console.WriteLine(eventArgs)
                End If
            Loop While Environment.TickCount < endTick
        End Sub
    End Class
End Namespace
```

## Free Pascal

---

```
// This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.

class procedure PullDataChangeNotification.Main;
var
  Client: EasyUAClient;
  EndTick: Cardinal;
  EventArgs: _EasyUADataChangeNotificationEventArgs;
begin
  // Instantiate the client object
  Client := CoEasyUAClient.Create;
  // In order to use event pull, you must set a non-zero queue capacity upfront.
  Client.PullDataChangeNotificationQueueCapacity := 1000;

  WriteLn('Subscribing...');

  Client.SubscribeDataChange(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10853',
    1000);

  WriteLn('Processing data change events for 1 minute...');

  EndTick := GetTickCount + 60*1000;
  while GetTickCount < EndTick do
  begin
    EventArgs := Client.PullDataChangeNotification(2*1000);
    if EventArgs <> nil then
      // Handle the notification event
      WriteLn(EventArgs.ToString);
  end;

  WriteLn('Unsubscribing...');

  Client.UnsubscribeAllMonitoredItems;

  WriteLn('Finished.');
end;
```

## JScript

```
// This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.

var Client = new ActiveXObject("OpcLabs.EasyOpc.UA.EasyUAClient");
// In order to use event pull, you must set a non-zero queue capacity upfront.
Client.PullDataChangeNotificationQueueCapacity = 1000;

WScript.Echo("Subscribing...");

Client.SubscribeDataChange("http://opcua.demo-this.com:51211/UA/SampleServer",
  "nsu=http://test.org/UA/Data/;i=10853", 1000);

WScript.Echo("Processing data change events for 1 minute...");

var endTime = new Date().getTime() + 60*1000
do {
  var EventArgs = Client.PullDataChangeNotification(2*1000);
  if (EventArgs !== null) {
    // Handle the notification event
    WScript.Echo(EventArgs);
  }
} while(new Date().getTime() < endTime);
```

## Object Pascal

```
// This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.

class procedure PullDataChangeNotification.Main;
var
  Client: TEasyUAClient;
  EndTick: Cardinal;
  EventArgs: _EasyUADataChangeNotificationEventArgs;
begin
  // Instantiate the client object
  Client := TEasyUAClient.Create(nil);
  // In order to use event pull, you must set a non-zero queue capacity upfront.
  Client.PullDataChangeNotificationQueueCapacity := 1000;

  WriteLn('Subscribing...');

  Client.SubscribeDataChange(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10853',
    1000);

  WriteLn('Processing data change events for 1 minute...');

  EndTick := Ticks + 60*1000;
  while Ticks < EndTick do
  begin
    EventArgs := Client.PullDataChangeNotification(2*1000);
    if EventArgs <> nil then
      // Handle the notification event
      WriteLn(EventArgs.ToString());
  end;

  WriteLn('Unsubscribing...');

  Client.UnsubscribeAllMonitoredItems;

  WriteLn('Finished.');
end;
```

## PHP

```
// This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.

$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");
// In order to use event pull, you must set a non-zero queue capacity upfront.
$Client->PullDataChangeNotificationQueueCapacity = 1000;

print "Subscribing...\n";
$Client->SubscribeDataChange("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853", 1000);

print "Processing data change events for 1 minute...\n";
$endTime = time() + 60;
do {
  $EventArgs = $Client->PullDataChangeNotification(2*1000);
  if (!is_null($EventArgs)) {
    // Handle the notification event
    print $EventArgs->ToString();
    print "\n";
  }
} while (time() < $endTime);
```

## Python

```
# This example shows how to subscribe to changes of a single monitored item, pull
events, and display each change.

import time
import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.UA.EasyUAClient')
# In order to use event pull, you must set a non-zero queue capacity upfront.
client.PullDataChangeNotificationQueueCapacity = 1000

print('Subscribing...')
client.SubscribeDataChange('http://opcua.demo-this.com:51211/UA/SampleServer',
                           'nsu=http://test.org/UA/Data/;i=10853', 1000)

print('Processing data change events for 1 minute...')
endTime = time.time() + 60
while time.time() < endTime:
    eventArgs = client.PullDataChangeNotification(2*1000)
    if eventArgs is not None:
        # Handle the notification event
        print(eventArgs)
```

## VBScript

Rem This example shows how to subscribe to changes of a single monitored item, pull events, and display each change.

Option Explicit

```
' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
' In order to use event pull, you must set a non-zero queue capacity upfront.
Client.PullDataChangeNotificationQueueCapacity = 1000

WScript.Echo "Subscribing..."
Client.SubscribeDataChange "http://opcua.demo-this.com:51211/UA/SampleServer",
                           "nsu=http://test.org/UA/Data/;i=10853", 1000

WScript.Echo "Processing data change events for 1 minute..."
Dim endTime: endTime = Now() + 60*(1/24/60/60)
Do
    Dim EventArgs: Set EventArgs = Client.PullDataChangeNotification(2*1000)
    If Not (EventArgs Is Nothing) Then
        ' Handle the notification event
        WScript.Echo EventArgs
    End If
Loop While Now() < endTime
```

Examples for OPC UA (Alarms & Conditions):

## Free Pascal

```
// This example shows how to subscribe to event notifications, pull events, and
display each incoming event.

class procedure PullEventNotification.Main;
var
  Client: EasyUAClient;
  EndTick: Cardinal;
```

```
EventArgs: _EasyUAEVENTIFICATIONEventArgs;
begin
    // Instantiate the client object and hook events
    Client := CoEasyUAClient.Create;
    // In order to use event pull, you must set a non-zero queue capacity upfront.
    Client.PullEventNotificationQueueCapacity := 1000;

    WriteLn('Subscribing...');

    Client.SubscribeEvent(
        'opc.tcp://opcua.demo-this.com:62544/Quickstarts/AlarmConditionServer',
        'nsu=http://opcfoundation.org/UA/;i=2253', // UAObjectIds.Server
        1000);

    WriteLn('Processing event notifications for 30 seconds...');

    EndTick := GetTickCount + 60*1000;
    while GetTickCount < EndTick do
    begin
        EventArgs := Client.PullEventNotification(2*1000);
        if EventArgs <> nil then
            // Handle the notification event
            WriteLn(EventArgs.ToString);
    end;

    WriteLn('Unsubscribing...');

    Client.UnsubscribeAllMonitoredItems;

    WriteLn('Finished.');
end;
```

## Object Pascal

```
// This example shows how to subscribe to event notifications, pull events, and
// display each incoming event.

class procedure PullEventNotification.Main;
var
    Client: TEasyUAClient;
    EndTick: Cardinal;
    EventArgs: _EasyUAEVENTIFICATIONEventArgs;
begin
    // Instantiate the client object and hook events
    Client := TEasyUAClient.Create(nil);
    // In order to use event pull, you must set a non-zero queue capacity upfront.
    Client.PullEventNotificationQueueCapacity := 1000;

    WriteLn('Subscribing...');

    Client.SubscribeEvent(
        'opc.tcp://opcua.demo-this.com:62544/Quickstarts/AlarmConditionServer',
        'nsu=http://opcfoundation.org/UA/;i=2253', // UAObjectIds.Server
        1000);

    WriteLn('Processing event notifications for 30 seconds...');

    EndTick := Ticks + 60*1000;
    while Ticks < EndTick do
    begin
        EventArgs := Client.PullEventNotification(2*1000);
        if EventArgs <> nil then
            // Handle the notification event
            WriteLn(EventArgs.ToString);
    end;
```

```
WriteLn('Unsubscribing...');  
Client.UnsubscribeAllMonitoredItems;  
  
WriteLn('Finished.');//  
end;
```

## VBScript

---

Rem This example shows how to subscribe to event notifications, pull events, and display each incoming event.

```
Option Explicit  
  
Const UAObjectIds_Server = "nsu=http://opcfoundation.org/UA/;i=2253"  
  
' Instantiate the client object and hook events  
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")  
' In order to use event pull, you must set a non-zero queue capacity upfront.  
Client.PullEventNotificationQueueCapacity = 1000  
  
WScript.Echo "Subscribing..."  
Client.SubscribeEvent "opc.tcp://opcua.demo-  
this.com:62544/Quickstarts/AlarmConditionServer", UAObjectIds_Server, 1000  
  
WScript.Echo "Processing event notifications for 30 seconds..."  
Dim endTime: endTime = Now() + 30*(1/24/60/60)  
Do  
    Dim EventArgs: Set EventArgs = Client.PullEventNotification(2*1000)  
    If Not (EventArgs Is Nothing) Then  
        ' Handle the notification event  
        WScript.Echo EventArgs  
    End If  
Loop While Now() < endTime
```

Besides the usual [ArgumentException](#) (when your code passes in an improper argument), the [InvalidOperationException](#) described above (which also indicates a bug in the code), and system exceptions such [OutOfMemoryException](#), the [PullXXXX](#) and [PullMultipleXXXX](#) methods can only throw one type of exception, and that is [ProcedureCallException](#). The current implementation does not actually throw this exception type, but future implementations may do so, and we recommend that your code is written in such a way that is ready for that. The [ProcedureCallException](#) indicates a failure in the event delivery mechanism (and a likely loss of one or more notifications).

Make sure you always specify a queue capacity that will provide a “buffer” of sufficient length for a difference between the rate of incoming notifications, and the rate your code can consume them. Besides an indication through the event log output (currently in OPC UA only), there is no way to detect queue overflows.

## 5.10 Error Handling

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

- Errors returned by system calls when performing OPC-related operations.
- In OPC Classic, errors returned by COM/DCOM infrastructure, including RPC errors.
- In OPC UA, errors returned by communication infrastructure, including Web service errors.
- Error returned by OPC-UA Stack or SDK.
- Network-related errors.
- Errors reported by the OPC server you are connecting to.
- Errors detected by the QuickOPC library.

In general, you cannot reliably prevent all 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 "Classic" and QuickOPC-UA each define one new type of exception, called [OpcException](#) (for OPC "Classic"), or [UAException](#) (for OPC-UA), derived from the [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 [UAException](#), or by examining the [ErrorCode](#) property. In most scenarios, however, your code will be handling all [OpcException](#)-s or [UAException](#)-s in the same way, independent of the inner exception or error code.

The [InnerException](#) can be:

- [COMException](#), especially common in OPC Classic.
- [UAStatusCodeException](#), when the OPC-UA status code is not as expected.
- [UAClientEngineException](#), for errors that originates in the OPC-UA client engine and not in the OPC service (or OPC-UA SDK).
- [UAServiceException](#), when a UA defined error occurs.
- [LicenseException](#), if there is a problem with component's license in QuickOPC-UA.
- An exception from the lower level systems, such as .NET Framework, WCF, OPC-UA stack or SDK.

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](#) or [UAException](#) 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 "Classic" and QuickOPC-UA operations, [OpcException](#) or [UAException](#) 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](#)).

## 5.10.1 Errors and Multiple-Element Operations

Some methods on the main [EasyDAClient](#) or [EasyUAClient](#) 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](#) or [UAException](#),

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](#) or [UAException](#) 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.

## 5.10.2 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](#) or [UAException](#), 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.

## 5.11 Helper Types

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

### 5.11.1 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. In OPC UA, an exception to this rule is for the maximum age of value (for reading), which is a floating point number in order to allow a finer precision.

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 “Classic” and QuickOPC-UA, 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.

### 5.11.2 Data Objects

The data objects hold the actual data made available by the OPC technology.

#### 5.11.2.1 Quality in OPC Classic

OPC Classic represents a quality of a data value by several bit-coded fields. QuickOPC 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:

DAQuality		
property <a href="#">DAQualityChoice</a> <a href="#">QualityChoiceBitField</a> { get; } property <a href="#">bool</a> <a href="#">IsBad</a> { get; } property <a href="#">bool</a> <a href="#">IsGood</a> { get; } property <a href="#">bool</a> <a href="#">IsUncertain</a> { get; }	<a href="#">SubStatus</a>	property <a href="#">DALimitChoice</a> <a href="#">LimitBitField</a> { get; set; }
property <a href="#">DASatusChoice</a> <a href="#">StatusBitField</a> { get; set; } <a href="#">SetQualityAndSubStatus</a> (...)		

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 specification “borrows” the quality type from OPC Data Access, and therefore the [DAQuality](#) type is used with OPC Alarms and Events as well.

Constants for [DAQuality](#) values are contained in the [DAQualities](#) enumeration.

## 5.11.2.2 Value, Timestamp and Quality (VTQ) in OPC Classic

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 a 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.

## 5.11.2.3 Variant Type (VarType) in OPC Classic

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 [VarTypes](#) enumeration. For example, [VarTypes.I2](#) denotes a 16-bit signed integer, [VarTypes.R4](#) denotes a 32-bit float, and [VarTypes.BStr](#) denotes a string. For arrays of values, use constants named [VarTypes.ArrayOfXXXX](#) (e.g. [VarTypes.ArrayOfI4](#)), or (if the element type is determined in the run time) the [VarType.MakeArrayType](#) method.

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 [VarTypes](#) enumeration. For example, [I2](#) denotes a 16-bit signed integer, [R4](#) denotes a 32-bit float, and [BStr](#) denotes a string. For arrays of values, use logical 'or' to combine the element type with the [Array](#) 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.

## 5.11.2.4 OPC UA Status Code

OPC represents a result of a service or operation in a status code, which consists of several bit-coded fields. QuickOPC-UA encapsulates the status code in a [UAStatusCode](#) class. The bit fields in status code are inter-dependent, making it a bit complicated to encode or decode it. The [UAStatusCode](#) 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 status code to a string.

The following table attempts to depict the elements of [UAStatusCode](#) and their relations, together with related properties and methods:

UAStatusCode						
property InternalValue						
property CodeBits	property FlagBits					
property Severity	property Condition	property StructureChanged	property SemanticsChanged	property InfoType property HasDataValueInfo	if HasDataValueInfo == true: property LimitInfo	property Overflow

The [CodeBits](#) part represents the numeric value of an error or condition, whereas the [FlagBits](#) part provides more information on the meaning of the status code.

QuickOPC-UA provides a [UACodeBits](#) class that contains constants for errors and conditions defined by the OPC-UA specifications. You can compare the resulting [CodeBits](#) to the constants if your code needs to look for a specific error or condition.

The semantics of some parts is dependent on [InfoType](#), and therefore the [LimitInfo](#) and [Overflow](#) properties can only be accessed if the [InfoType](#) has a corresponding value, which is indicated by the value of [HasDataValueInfo](#) property.

## 5.11.2.5 OPC UA Attribute Data

The combination of data value, source and serve timestamps and status code is common in OPC-UA. This combination is returned e.g. when an attribute of an OPC node is read. Note that according to OPC specification, the actual data value is only valid with certain status codes (typically, when the status is Good).

QuickOPC-UA has a [UAAttributeData](#) object for this combination. The object provides access to individual elements of this 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 attribute data, 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 status), and a null reference case is handled as well.

## 5.11.3 Argument Objects

*Argument objects* are "holders" that contain all arguments necessary to perform certain operation. QuickOPC has method overrides that accept individual arguments, but also method overrides that accept argument object as an input. The argument object can be reused with other operations without having to specify individual arguments again, and it can also participate in multiple-operand methods where an array of argument objects is accepted on input, and an array of result objects is returned on output.

All argument objects derive from [OperationArguments](#).

## 5.11.4 Result Objects

*Result objects* are returned by methods that work on multiple elements simultaneously such as [EasyDAClient.ReadMultipleItems](#), or [EasyUAClient.ReadMultiple](#). 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 (as operations on other elements might have succeeded).

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), [DAVtqResult](#) (contains value, timestamp, and quality combination for OPC "Classic"), or [UAAttributeDataResult](#) (contains value, timestamps, and status code combination for OPC-UA).

The [OperationResult](#) object (and therefore all result objects derived from it as well) has a [Diagnostics](#) property, which contains a collection of warning and informational diagnostics entries gathered, performing the requested operation. You can use this collection to retrieve additional information about the outcome of the operation.

The additional [OperationResult.DiagnosticsSummary](#) string contains a textual summary of diagnostics information, one message per line.

Note that diagnostics information in the [OperationResult](#) is gathered and available both in case of success, and in case of a failure.

Currently, the diagnostics information is only available for operations on OPC Unified Architecture (OPC-UA), i.e. not for OPC "Classic".

## 5.11.5 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 for OPC Classic:

- [ServerElement](#) object contains information gathered about an OPC Classic (OPC COM or OPC XML) server.
- [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.
- [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.

There are following types of element objects for OPC UA:

- [UAApplicationElement](#) object contains information gathered about an OPC-UA application (OPC-UA server).
- [UANodeElement](#) object contains information gathered about an OPC node (in OPC server's address space). The node can represent e.g. a branch in the organizing tree of nodes, or a process value, or a property.

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.

## 5.11.6 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 for OPC Classic:

- [ServerDescriptor](#) contains information necessary to identify and connect to an OPC Classic server, such as the server's ProgID.
- [AENodeDescriptor](#) contains information necessary to identify a node (area or source) in OPC A&E address space, such as Node Id (a string).
- [DANodeDescriptor](#) contains information necessary to identify a node (branch or leaf) in OPC DA address space, such as Node Id (a string).
- [DAItemDescriptor](#) contains information necessary to identify an OPC item, such as its Item Id, together with additional information such as the data type requested.
- [DAPropertyDescriptor](#) contains information necessary to identify an OPC property, such as its Property Id (or a qualified name, in case of OPC XML).

There are following types of descriptor objects for OPC UA:

- [UAEndpointDescriptor](#) contains information necessary to identify and connect to an OPC server, such as the server's URL.
- [UANodeDescriptor](#) contains information necessary to identify an OPC node in the server, such as the UANodeId that holds the text of its expanded node Id.

Note: For OPC-UA, the meaning of these objects is described further below in chapter "Identifying Information in OPC-UA".

In a [ServerDescriptor](#), the primary means of server identification is the [ServerClass](#) property, which contains a string that can hold either ProgID of the server, or its {CLSID}. There are also additional [ServerDescriptor](#) properties, all related to the [ServerClass](#) property, allowing to individually retrieve or modify parts of the server class designation. The [ProgId](#) property contains a ProgID of the server (and is an empty string if the ProgID is not given). The [Clsid](#) property contains the CLSID of the server in a form of a [Guid](#) (and is [Guid.Empty](#) if no CLSID for the server is given). The [ClssidString](#) property contains the CLSID string of the server (and is an empty string if no CLSID for the server is given). The [ObjectId](#) property contains the object ID, i.e. a ProgID, {CLSID}, or ProgID/{CLSID}.

Each [ServerDescriptor](#) is also represented in a string form that uses the URL syntax (e.g.

"<opcda://127.0.0.1/Kepware.KEPServerEX.V5/{B3AF0BF6-4C0C-4804-A122-6F3B160F4397}>"). For more information about these "OPC URLs", see <http://www.opclabs.com/resources/developer-blog/1086-a-partial-story-of-opc-urls>.



If you have received an element object (e.g. from browsing methods or common OPC dialogs), you can convert it to a descriptor object. In OPC Classic, 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. In OPC UA, there is a constructor for [UAEndpointDescriptor](#) object that accepts [UAApplicationElement](#) as an input, and there is a constructor for [UANodeDescriptor](#) object that accepts [UANodeElement](#) as an input, too.

Nodes and items in OPC address space can be specified using their node Id (or item Id), or using so-called browse paths.

The item Id (or node Id) is a string that uniquely addresses an item (or a node) in OPC address space. Its syntax is determined by the OPC server. It usually consists of the names of nodes used to reach that node from the root of the OPC address space, together with some decorating characters and separators. This is not a requirement, however, and the OPC server may even use item Ids that do not resemble the node names at all. Unless you have detailed knowledge of the particular OPC server you are connecting to, you should not attempt to create item Id strings yourself, from various parts – they should always come from the server. Item Ids (node Ids) are specified by the [ItemId](#) property of the [DANodeDescriptor](#) or [DAItemDescriptor](#) object. Strings convert implicitly to [DANodeDescriptor](#) or [DAItemDescriptor](#) containing the input string as the value of [ItemId](#) property, and you can therefore simply use the string item Ids in place of node and item descriptors.

## 5.11.7 Browse Paths

An alternative way (to using a node ID) of specifying a node in OPC address space is using a browse path. The browse path is a sequence of node names, starting from a known location (e.g. the root of the OPC address space). The browse path can also be expressed as a string, where the individual node names are separated by delimiters (e.g. "/", a slash).

Browse paths can be absolute (starting from a specified node), or relative.

The advantage of the browse paths is that you can construct them yourself, with just knowledge of the node names, without knowing the full node IDs. In the end, each item must be addressed by its node ID, and QuickOPC will resolve the browse path to a node ID as necessary.

### 5.11.7.1 Browse Paths for OPC Classic

In OPC Classic, absolute browse paths always start at the root of the OPC address space, because the root is the only "pre-defined" or well-known node.

An absolute browse path for OPC Classic is contained in a [BrowsePath](#) object. Browse paths are commonly specified by the [BrowsePath](#) property of the [DANodeDescriptor](#) or [DAItemDescriptor](#) object.

If a non-null ItemID is specified in the descriptor, QuickOPC will use this item Id and ignore the browse path. If ItemID is null, QuickOPC will attempt to resolve the browse path contained in the [BrowsePath](#) property of the descriptor. Either item id, or browse path (or both) must be specified.

Besides the ability to get around usage of full node IDs, the other reason to use browse paths is for OPC browsing with OPC Servers that only support OPC Data Access 1.0 specification. Such OPC servers cannot start the browsing at a node in the address given just by its Item ID; the node must always be reached by browsing from the root level. If the ID of such node is already known, QuickOPC takes care of supplying the proper browse path automatically, but this cannot be always done. If, however, a browse path is given (which can be done by using the [DANodeElement](#) that is the output of the browsing to construct the [DANodeDescriptor](#) that is the input of further browsing), the browsing can proceed normally.

### 5.11.7.2 OPC Classic Browse Path Format

In OPC Classic, the slash at the beginning of the browse path denotes an absolute path that starts from the root of the OPC address space. The string form of a browse path is a concept on the client side (QuickOPC), and does not appear in OPC specifications.

A browse path can be represented by a string; in such case, it can express either a relative or absolute browse path (the [BrowsePath](#) object cannot hold relative browse paths).

As mentioned above, the format of the browse path string is such that the individual node names are separated by slashes (""). The slash at the beginning of the browse path denotes an absolute path that starts from the root of the OPC address space. In addition, '.' denotes a current level, and '..' denotes a parent level, similarly to the conventions used in Windows file system.

Because the node names can contain any characters, we need to consider situations in which the node name itself contains a slash ('/') or a dot ('.'). In such case, in the string format of browse paths, these characters are escaped by preceding them with an ampersand ('&'). An ampersand itself needs to be escaped, too.

### 5.11.7.3 Browse Paths in OPC-UA

In OPC Unified Architecture, each element of the browse path is not a simple string, but instead, it is a structured object ([UABrowsePathElement](#)) which contains more information that allows identifying the child node.

The OPC-UA browse path element contains a qualified name of the child node (a name qualified with a namespace URI), in the [UABrowsePathElement.TargetName](#) property. This allows ensuring the uniqueness of browse path elements in complex OPC-UA servers, such as when the address space is aggregated from multiple sources.

Because in OPC-UA there may be various references that lead from the parent node to a child node, the OPC-UA browse paths need to specify which type of reference should be used. This is done by the [UABrowsePathElement.ReferenceType](#) property; in addition, it is possible to specify whether the reference should be followed in reverse direction ([UABrowsePathElement.ReferencesInverse](#)), and whether subtypes of the given reference type should also be considered.

An absolute browse path for OPC-UA is contained in a [UABrowsePath](#) object. The starting node of an OPC-UA browse path is contained in the [UABrowsePath.StartingNodeId](#) property. Browse paths are commonly specified by the [BrowsePath](#) property of the [UANodeDescriptor](#) object.

The [UABrowsePath](#) can be a null browse path, in case its [StartingNodeId](#) is a null node ID. A null browse path is used to specify that no browse path is known or given. Note that passing a null browse path object is different from passing a null reference to [UABrowsePath](#) (which is not allowed in most places).

If a non-null [NodeId](#) is specified in the descriptor, QuickOPC will use this node Id and ignore the browse path. If [NodeId](#) is null, QuickOPC will attempt to resolve the browse path contained in the [BrowsePath](#) property of the descriptor. Either node ID, or browse path (or both) must be specified.

## 5.11.7.4 OPC-UA Browse Path Format

QuickOPC-UA uses the relative browse path string format recommended by the OPC specification, but extends it by an ability to specify the starting node ID (and therefore introducing the absolute browse paths), and by an ability to specify the namespaces not by an index into a namespace table, but by a namespace URI. The precise syntax of these extensions is beyond the current scope of this document, and the description below is just a close approximation.

An OPC-UA browse path can be represented by a string; in such case, it can express either a relative or absolute browse path (the [UABrowsePath](#) object cannot hold relative browse paths).

As mentioned above, the format of the browse path string is such that the individual elements (usually, node names) are separated by delimiters. In OPC-UA, the delimiter actually precedes every element, i.e. there is a delimiter also at the beginning of the relative browse path. Different delimiters are available for specifying various reference types. Following delimiters are available:

Delimiter	Description
/	Follow any subtype of HierarchicalReferences.
.	Follow any subtype of Aggregates reference type.
< <b>referenceSpecifier</b> >	Follow the specified reference type. A '#' placed in front of the reference type name indicates that subtypes should not be followed. A '!' in front of the reference type name is used to indicate that the inverse reference should be followed.

If the OPC-UA browse path does not start with any of the above delimiters, it denotes an absolute path that starts from the node specified at the beginning of the string. For example, "[Objects]/Boilers" is an example of such absolute OPC-UA browse path, whereas "/Boilers" in OPC-UA is a relative browse path.

The string form of an OPC-UA browse path is a concept on the client side (QuickOPC), and the browse paths strings are not directly processed by OPC-UA servers.

Because the node names can contain any characters, we need to consider situations in which the node name itself contains a slash ('/') or a dot ('.'). In such case, in the string format of browse paths, these characters are escaped by preceding them with an ampersand ('&'). An ampersand itself needs to be escaped, too.

Parsing an absolute browse path:

## VBScript

```
Rem Parses an absolute OPC-UA browse path and displays its starting node and elements.

Option Explicit

Dim BrowsePathParser: Set BrowsePathParser =
CreateObject("OpcLabs.EasyOpc.UA.Parsing.UABrowsePathParser")
Dim BrowsePath: Set BrowsePath = BrowsePathParser.Parse([
[ObjectsFolder]/Data/Static/UserScalar"])

' Display results
WScript.Echo "StartingNodeId: " & BrowsePath.StartingNodeId

WScript.Echo "Elements:"
Dim BrowsePathElement: For Each BrowsePathElement In BrowsePath.Elements
    WScript.Echo BrowsePathElement
Next
```

Attempting to parse an absolute browse path:

## VBScript

```
Rem Attempts to parses an absolute OPC-UA browse path and displays its starting node and elements.

Option Explicit

Dim BrowsePathParser: Set BrowsePathParser =
CreateObject("OpcLabs.EasyOpc.UA.Parsing.UABrowsePathParser")
Dim BrowsePath
Dim StringParsingError: Set StringParsingError = BrowsePathParser.TryParse([
[ObjectsFolder]/Data/Static/UserScalar"], BrowsePath)

' Display results
If Not (StringParsingError Is Nothing) Then
    WScript.Echo "*** Error: " & StringParsingError
    WScript.Quit
End If

WScript.Echo "StartingNodeId: " & BrowsePath.StartingNodeId

WScript.Echo "Elements:"
Dim BrowsePathElement: For Each BrowsePathElement In BrowsePath.Elements
    WScript.Echo BrowsePathElement
Next
```

Parsing a relative browse path:

## VBScript

```
Rem Parses a relative OPC-UA browse path and displays its elements.

Option Explicit

Dim BrowsePathParser: Set BrowsePathParser =
```

```
CreateObject("OpcLabs.EasyOpc.UA.Parsing.UABrowsePathParser")
Dim BrowsePathElements: Set BrowsePathElements =
BrowsePathParser.ParseRelative("/Data.Dynamic.Scalar.CycleComplete")

' Display results
Dim BrowsePathElement: For Each BrowsePathElement In BrowsePathElements
    WScript.Echo BrowsePathElement
Next
```

Attempting to parse a relative browse path:

## VBScript

```
Rem Attempts to parse a relative OPC-UA browse path and displays its elements.

Option Explicit

Dim BrowsePathElements: Set BrowsePathElements =
CreateObject("OpcLabs.EasyOpc.UA.AddressSpace.UABrowsePathElementCollection")

Dim BrowsePathParser: Set BrowsePathParser =
CreateObject("OpcLabs.EasyOpc.UA.Parsing.UABrowsePathParser")
Dim StringParsingError: Set StringParsingError =
BrowsePathParser.TryParseRelative("/Data.Dynamic.Scalar.CycleComplete",
BrowsePathElements)

' Display results
If Not (StringParsingError Is Nothing) Then
    WScript.Echo "*** Error: " & StringParsingError
    WScript.Quit
End If

Dim BrowsePathElement: For Each BrowsePathElement In BrowsePathElements
    WScript.Echo BrowsePathElement
Next
```

## 5.11.8 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 on their details, see the “Setting Parameters” and “Advanced Topics” chapters, and also the Reference documentation.

The parameters of all [EasyXXClient](#) objects are consistently organized into three groups:

- [SharedParameters](#) (static parameters that are always shared among object instances),
- [InstanceParameters](#) (parameters that can always be set independently for each object instance), and
- [AdaptableParameters](#) (parameters that are normally shared, but can be made specific to an instance if the [Isolated](#) property is set to ‘[true](#)’).

An instance of any [EasyXXClient](#) object that is not isolated (i.e. has its [Isolated](#) property equal to ‘[false](#)’, which is the default) actually uses following settings:

- Shared parameters from the static property [EasyXXClient.SharedParameters](#),
- adaptable parameters from the static property [EasyXXClient.AdaptableParameters](#), and
- instance parameters from its instance property [EasyXXClient.InstanceParameters](#).

An isolated instance (i.e. an instance which has its [Isolated](#) property explicitly set to ‘[true](#)’) of any [EasyXXClient](#) object

actually uses following settings:

- Shared parameters from the static property `EasyXXClient.SharedParameters`,
- adaptable parameters from its instance property `EasyXXClient.IsolatedParameters`, and
- instance parameters from its instance property `EasyXXClient.InstanceParameters`.



In QuickOPC.NET, there are sometimes useful conversions and implicit conversions that allow you to easily construct parameter objects. For example, the `DAGroupParameters` object has a `FromInt32` static method, and a corresponding implicit conversion operator, that allow it be constructed from an integer that represents the requested update rate (in milliseconds). This means that in C# and many other languages, you can simply use an integer update rate at all places where `DAGroupParameters` object is expected, if you are fine with specifying just the update rate and keeping the other properties of `DAGroupParameters` at their defaults.

All parameter and policy objects (and their constituents) consistently have a static `Default` property. This allows for a cleaner code where an arguments is needed, but you want to supply a default (there is no need to call the default constructor).

All parameter objects consistently derive from a `Parameters` base class. This class has a `StandardName` property (a string). The standard name identifies certain well-known combination of parameter values. When all the parameters correspond to a well-known combination, the `StandardName` property contains a symbolic name of such combination; otherwise, it contains an empty string. You can also set the `StandardName` property yourself, and the parameter values will change to reflect the name.

Some other objects that have similar nature also derive from the `Parameters` base class.

## 5.11.9 Utility Classes



The `VarTypeUtilities` static class provides a `FromType` method that allows you to determine the COM-based type `VarType` (used in OPC operations) that corresponds to a given .NET Type.

The `DAUtilities` static class provides methods that allow you to determine whether a given data type (`VARIANT`), percent deadband, update rate or value age are valid values in OPC Data Access.

The `AEUtilities` static class provides methods that allow you to determine whether given data type (`VarType`), event severity, event type filter, or notification rate are valid values in OPC Alarms&Events.

## 5.12 Identifying information in OPC XML

QuickOPC.NET supports the OPC XML-DA specification transparently, with the same objects that are used for COM-based OPC Data Access. The two specifications are quite similar, and for most part, developers will not see much difference. Accesses to OPC-DA and OPC XML-DA can be freely mixed, even in the same method call. This transparency has been achieved by several generalizations in the object model and API.

OPC XML-DA specification identifies certain entities differently. The specifics of those are explained further below.

### 5.12.1 Servers

In OPC XML, a client connects to a server with the use of the server URL. QuickOPC uses the `ServerDescriptor` class for identifying servers, and you can construct a `ServerDescriptor` by passing it a string that contains the server URL. The URL is also accessible in the `UrlString` property of the `ServerDescriptor`, and there is an implicit conversion available from a (URL) string to a `ServerDescriptor`. This means that at most places in QuickOPC API, you can specify an OPC XML server simply by using its URL.

URLs of OPC XML servers can start with "http:", "https:", or "opc.xmlda" (which is internally treated as "http:").

For compatibility between OPC COM and OPC XML, the `ServerDescriptor` (which, for COM, contained mainly the

machine name and server class in the past) is generalized to be able to describe both COM and XML based servers. The primary information in the [ServerDescriptor](#) is its [UrlString](#). For OPC COM servers, the URL is composed in such a way that it contains the original [Location](#) (i.e. machine name) and [ServerClass](#) properties. For OPC XML servers, their URL can be used as the [UrlString](#) directly.

Note: We use the term "location" when we need to describe either a "machine name" (used in COM) or a "host name" (used with the Web, and OPC XML).

The [ServerDescriptor](#) also contains an additional [NetworkSecurity](#) property. This object can optionally specify the network credentials used when connecting to an OPC XML server.

The [ServerElement](#) (returned when you browse for OPC servers) has a [UrlString](#) property that can be used to connect to OPC XML servers.

## 5.12.2 Nodes and Items

OPC XML-DA does not identify OPC items by a single string as OPC COM. Instead, it uses an additional string, and only the combination of the two identifies an item in an OPC XML-DA server. The original string that roughly corresponds to OPC item ID is called "ItemName" in the OPC XML-DA specification; in QuickOPC, we use the existing [ItemId](#) property for it. The additional string is called "ItemPath" in the OPC XML-DA specification; in QuickOPC, we have introduced a [NodePath](#) property for it. This property appears both in the [DANodeDescriptor](#) and [DANodeElement](#). When you use the information received from the OPC XML browsing, both [ItemId](#) and [NodePath](#) are filled in the [DANodeElement](#), and you can easily convert it to a [DANodeDescriptor](#) and it will "just work". If you are getting the nodes (items) from elsewhere, you need to create a [DANodeDescriptor](#) that contains both these strings (at least in general case; some OPC XML-DA server do not use them both).

## 5.12.3 Properties

OPC XML-DA identifies properties by a string (a XML qualified name), instead of the numerical code used in OPC COM. For this a reason, a [DAPropertyDescriptor](#) and a [DAPropertyElement](#) have an additional [QualifiedName](#) property. When the component returns the [DAPropertyElement](#) from browsing, it fill is the information it knows about. When you pass the [DAPropertyDescriptor](#) to the component, you can fill in either [PropertyId](#), or [QualifiedName](#), or both. The component will use whatever is available and at the same time usable by the underlying technology. For the most common case, standard (well-known) properties can always be identified using their numerical ID, and the component will look up their qualified name automatically.

The [DAPropertyElement](#) (returned when you browse for OPC properties) contains an additional [ItemPath](#) property for OPC XML. For properties coming from OPC XML servers, if the property can be accessed as an item as well, the property is filled in with the string that together with the [ItemId](#) identifies the OPC item that corresponds to the OPC property.

## 5.13 Identifying Information in OPC-UA

These chapters explain how the most commonly used entities in OPC Unified Architecture, such as servers, nodes, and attributes, are identified in the application code.

Note: Some entities in OPC Unified Architecture are identified by URIs and URLs. .NET Framework has a [System.Uri](#) class that provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI. It is generally recommended that programs use this class for URI manipulation internally, instead of working with plain strings containing URLs. In OPC Unified Architecture, however, the URIs and URLs are used in such a way that they do not always have to fully conform to the specifications. For this reason, URI and URL arguments and properties in QuickOPC-UA are also strings and not [System.Uri](#) (and are consistently named so they contain "[UriString](#)" or "[UrlString](#)").

## 5.13.1 Server Endpoints

A Server Endpoint is a physical address available on a network that allows clients to access one or more services provided by a server. Server endpoint is specified by its URL string.

Besides standard URLs e.g. with HTTP scheme, OPC Unified Architecture uses its own schemes, such as "opc.tcp". Examples of server endpoint URL strings are:

- <http://opcua.demo-this.com:51211/UA/SampleServer>
- <opc.tcp:// opcua.demo-this.com:51210/UA/SampleServer>
- <http://localhost:51211/UA/SampleServer>
- <opc.tcp://localhost:51210/UA/SampleServer>

Note: For backward compatibility, it is also possible to use "opc.http:" in place of "http:" scheme in OPC UA endpoint descriptors (for OPC Unified Architecture over SOAP/HTTP). You should not, however, use "opc.http:" in new projects.

In QuickOPC-UA, server endpoint corresponds to a [UAEndpointDescriptor](#) object that you create and pass to various methods. An implicit conversion from a string containing the URL of the endpoint exists. It is therefore common to specify [UAEndpointDescriptor](#)-s simply as strings, relying on the implicit conversion.

A single OPC-UA server may have multiple endpoints with the same functionality, but using different protocols, or different security settings, and expose the list of its own endpoint using discovery services. By default, the component will attempt to interpret the server endpoint you have specified as server's discovery endpoint, and automatically select the best endpoint.

How precisely the endpoint is selected is determined by Endpoint Selection Policy ([UAEndpointSelectionPolicy](#) object) that you can parameterize in the [SessionParameters](#) property on the [EasyUAClient](#) object.

## 5.13.2 Node IDs

Node ID is an identifier for a node in an OPC server's address space.

OPC Unified Architecture allows the OPC server to choose one or more types of node IDs for representation of its nodes. Node IDs can be numeric (a 32-bit integer), string, a GUID (globally unique identifier, 128 bits), or opaque (a binary data blob).

QuickOPC-UA works with so-called expanded node IDs, which contain the node's identifier together with a complete namespace URI. A single server may contain nodes in multiple namespaces, and the server maintains a namespace table where each namespace can be identified by an index. During a communication session with an OPC client, nodes are identified using indices into the namespace table. The namespace table contents and order of elements may, however, change between sessions.

QuickOPC-UA does not expose node IDs with indices into a namespace table alone. Instead, it always passes around the complete URI string of a namespace with a node ID. Because this expanded form of a node ID is used, a Node ID in QuickOPC-UA is stable between sessions, and can also be persistent (stored) and later used without a complexity of additional index remapping.

Depending on the type of Node ID, the expanded node ID string may have one of the following forms:

- nsu=*namespaceUri*;i=*integer*
- nsu=*namespaceUri*;s=*string*
- nsu=*namespaceUri*;g=*guid*
- nsu=*namespaceUri*;b=*base64string*

In QuickOPC-UA, an expanded node ID corresponds to [UANodeId](#) object that you create and pass to various methods. An implicit conversion from a string containing the expanded text of Node ID exists. It is therefore common to specify [UANodeId](#)-s simply as strings, relying on the implicit conversion. Also note that [UANodeId](#) is usually a part of [UANodeDescriptor](#) and can be (implicitly) converted to it easily.

When the [UANodeId](#) object contains no data, it is considered a null Node Id, and its [IsNull](#) property is true. Beware that this is different from having a null reference to a [UANodeId](#) (which, in most places, is not allowed).

Note: During browsing, the OPC server provides so-called browse names (browse IDs), which can be combined into browse paths. The browse names (browse IDs) usually "look" more user-friendly and you may be tempted to think that they would be more appropriate for node identification. The purpose of browse names (browse IDs) is, however, different, and using them alone for node identification would be inefficient.

## 5.13.2.1 Namespace indices in Node Ids

During most communication between the client and the server, OPC Unified Architecture uses namespace indices, which are integer numbers, to distinguish between namespaces. The namespaces indices reference a namespace table made available by the server, and only the namespace table actually contains the namespace URLs. The order of entries in the server's namespace table can change between sessions. For this reason, namespace indices should not be persisted, as they alone are not generally sufficient to identify a namespace and thus a node.

There are, however, situations where you may want to include the namespace index in the expanded node Id string:

- For optimization. If the namespace index is given, it may be slightly faster to reference the node.
- Sometimes the party you are working with does not fully understand the consequences of using a namespace index, and only gives you node Ids with the namespace indices, without the namespace URLs.
- For situations like this, QuickOPC-UA allows the use of "ns=..." notation for a namespace index inside the expanded node ID strings, and has two additional provisions:
- When expanded node ID strings are provided out of QuickOPC-UA components, they contain BOTH the namespace URI, and the namespace index. For example, the browsing may return following expanded node ID string to you:

nsu=http://test.org/UA/Data/;ns=2;i=10219

- When QuickOPC-UA receives an expanded node ID string on input, it uses the namespace URI specified in the string by the "nsu=..." notation, if present. If the namespace URI is not present, a namespace index, specified by the "ns=..." notation (or defaulted to zero), is used. If both the namespace URI and the namespace index are present, semantically it always behaves the same as if only the namespace URI was specified, but the namespace index may be used for performance optimization. This means that the expanded node ID strings provided out of the component always refer to nodes through the namespace URI, even if they (in addition) contain the namespace index.

Note that namespace index 0 (zero) is reserved, and is not usually specified inside the expanded node ID strings.

## 5.13.2.2 Standard Node IDs

QuickOPC-UA provides static classes with standard Node IDs defined by OPC

Foundation: [UADatatypes](#), [UAMethods](#), [UAObject](#), [UAObjectTypes](#), [UAResourceTypes](#), [UAVariables](#), and [UAVariableTypes](#). You can use properties of these static classes to easily specify "well-known" nodes you need to refer, for example, the "ObjectsFolder" node which is a common starting node for browsing.

There are following classes with standard Node IDs for OPC-UA:

- [UADatatypes](#): A class that declares constants for all Data Types in the Model Design. This class contains definitions for nodes such as [BaseDatatype](#), [Number](#), [Integer](#), [Boolean](#), and so on.
- [UAMethods](#): A class that declares constants for all MethodIds. in the Model Design. This class contains definitions for method nodes for condition handling, status machine handling, and others.
- [UAObject](#): A class that declares constants for Objects in the Model Design. This class contains definitions for folder nodes such as [DataTypesFolder](#), [EventTypesFolder](#), [ObjectsFolder](#), [ObjectTypesFolder](#), [ReferenceTypesFolder](#), [RootFolder](#), [Server](#), [TypesFolder](#), [VariableTypesFolder](#), [ViewsFolder](#), and others.
- [UAObjectTypes](#): A class that declares constants for Object Types in the Model Design. This class contains definitions for nodes such as [BaseObjectType](#), [FolderType](#), and many other object types.
- [UAResourceTypes](#): A class that declares constants for Reference Types in the Model Design. This class contains definitions for nodes such as [Aggregates](#), [HasChild](#), [HasComponent](#), [HasProperty](#), [HierarchicalReferences](#), [Organizes](#), and more.
- [UAVariables](#): A class that declares constants for all Variables in the Model Design.
- [UAVariableTypes](#): A class that declares constants for all Variable Types in the Model Design.

When you need to pass a standard node to any method, simply use the corresponding property from the static class, such as [UAObject](#).[ObjectsFolder](#).

## 5.13.4 Attribute IDs

Attribute is a primitive characteristic of a Node. All Attributes are defined by OPC UA, and may not be defined by clients or servers (i.e. the attribute set is not vendor-extensible). Attributes are the only elements in the address space permitted to have data values.

In most cases, you will probably be working with the [Value](#) attribute, which contains the actual value of a variable. If you need to address a different attribute, use the [UAAttributeld](#) enumeration, which contains constants for all attribute IDs available in OPC UA address space.

[UAAttributeld](#) can be given as [Attributeld](#) property of the [UAAttributeArguments](#) object (and derived objects, such as [UAReadArguments](#), [UAWriteArguments](#), [UAWriteValueArguments](#), and [EasyUAMonitoredItemArguments](#)). You can then pass this object to a corresponding method on the main [EasyUAClient](#) object.

## 5.13.5 Index Range Lists

OPC Unified Architecture has support for single- and multidimensional arrays in data values. In addition, it is also possible to work with (read, write, subscribe to) only a subset of an array, so that the whole (potentially large) array does not have to be transferred between the OPC server and OPC client.

QuickOPC-UA uses Index Range Lists ([UAIndexRangeList](#) objects) to control which parts of array value should be accessed. An index range list can be used to identify the whole array, a single element of a structure or an array, or a single range of indexes for arrays. Index Range List is basically a list of individual Index Ranges ([UAIndexRange](#) objects), each containing a range for a single dimension of an array.

There is an implicit conversion from an integer ([Int32](#)) to a [UAIndexRange](#), so if your language (such as C#) supports implicit conversions, an index range containing just a single element can be written simply as the element index, without explicitly constructing the [UAIndexRange](#) object.

Besides constructing the [UAIndexRangeList](#) from individual [UAIndexRange](#) objects, it is also possible to specify the index range list by parsing a string, using static [UAIndexRangeList.TryParse](#) method.

Examples of index range strings:

- 5:20
- 10:11,1:2
- 1,4:7,90

Dimensions are separated by commas. A minimum and maximum value, separated by a colon, denotes a range of index. If the colon is missing, only a single for an index is selected. An empty string denotes the whole array.

One-dimensional index range lists can be easily constructed using the [UAIndexRangeList.OneDimension](#) static method, passing it either the single index, or minimum and maximum indices.

[UAIndexRangeList](#) can be given as [IndexRangeList](#) property of the [UAAttributeArguments](#) object (and derived objects, such as [UAReadArguments](#), [UAWriteArguments](#), [UAWriteValueArguments](#), and [EasyUAMonitoredItemArguments](#)). You can then pass this object to a corresponding method on the main [EasyUAClient](#) object.

### C#

```
// This example shows how to read a range of values from an array.
using OpcLabs.EasyOpc.UA;
using System;
using OpcLabs.EasyOpc.UA.OperationModel;

namespace UADocExamples
{
    namespace _UAIndexRangeList
```

```
{  
    class Usage  
    {  
        public static void ReadValue()  
        {  
            // Instantiate the client object  
            var easyUAClient = new EasyUAClient();  
  
            // Obtain the value, indicating that just the elements 2 to 4 should  
            // be returned  
            object value = easyUAClient.ReadValue(  
                new UAReadArguments(  
                    "http://opcua.demo-this.com:51211/UA/SampleServer", // or  
                    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"  
                    "nsu=http://test.org/UA/Data/;i=10305",  
                    UAIndexRangeList.OneDimension(2, 4)));  
  
            // Cast to typed array  
            var arrayValue = (Int32[]) value;  
  
            // Display results  
            for (int i = 0; i < 3; i++)  
                Console.WriteLine("arrayValue[{0}]: {1}", i, arrayValue[i]);  
        }  
    }  
}
```

## VB.NET

```
' This example shows how to read a range of values from an array.  
Imports OpcLabs.EasyOpc.UA  
Imports OpcLabs.EasyOpc.OperationModel  
  
Namespace _UAIndexRangeList  
    Friend Class Usage  
        Public Shared Sub ReadValue()  
            ' Instantiate the client object  
            Dim easyUAClient = New EasyUAClient()  
  
            ' Obtain the value, indicating that just the elements 2 to 4 should be  
            ' returned  
            Dim value As Object = easyUAClient.ReadValue(  
                New UAReadArguments(  
                    "http://opcua.demo-this.com:51211/UA/SampleServer",  
                    "nsu=http://test.org/UA/Data/;i=10305",  
                    UAIndexRangeList.OneDimension(2, 4))  
            ' or "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"  
  
            ' Cast to typed array  
            Dim arrayValue = DirectCast(value, Int32())  
  
            ' Display results  
            For i = 0 To 2  
                Console.WriteLine("arrayValue[{0}]: {1}", i, arrayValue(i))  
            Next  
        End Sub  
    End Class  
End Namespace
```

## 5.14 OPC-UA Security

OPC Unified Architecture is designed as secure solution from the ground up. The result is a solid, secure infrastructure that can be, at the same time, complex to understand, and sometimes difficult to deploy and configure properly.

Security in OPC UA is a broad subject that we cannot and will not cover in detail here. It is recommended that you study available materials. Here are some useful links that can get you started:

- Secure communication with IEC 62541 OPC UA: [https://opcfoundation.org/wp-content/uploads/2014/05/OPC-UA\\_Security\\_EN.pdf](https://opcfoundation.org/wp-content/uploads/2014/05/OPC-UA_Security_EN.pdf)
- The OPC UA Security Model For Administrators: [https://opcfoundation.org/wp-content/uploads/2014/05/OPC-UA\\_Security\\_Model\\_for\\_Administrators\\_V1.00.pdf](https://opcfoundation.org/wp-content/uploads/2014/05/OPC-UA_Security_Model_for_Administrators_V1.00.pdf)

QuickOPC-UA attempts to simplify the security configuration where possible, but a reasonable level of understanding OPC-UA security is still needed.

### 5.14.1 Security in Endpoint Selection

As explained earlier, a single OPC-UA server may have multiple endpoints with the same functionality, but using different protocols, or different security settings (for example, an endpoint that provide message security may exist alongside with endpoint that does not provide message security).

The [MessageSecurityPreference](#) property of the [UAEndpointSelectionPolicy](#) tells the component whether endpoints that provide message security are preferred for selection. It has three possible values:

- **Negative**: The endpoints without message security will be given a major preference. This is the default setting, for good interoperability.
- **None**: The endpoint security level provided by the server will be followed without change.
- **Positive**: The endpoints with message security will be given a major preference.

The [AllowedMessageSecurityModes](#) property of the [UAEndpointSelectionPolicy](#) gives you a possibility to completely enable or disable certain classes of endpoints from selection, based on their security mode. You can combine the value of this property from following flags:

- **SecurityNone**: No security is applied.
- **SecuritySign**: All messages are signed but not encrypted.
- **SecuritySignAndEncrypt**: All messages are signed and encrypted.

There are also pre-defined symbolic values for combinations of these flags, such as **All** (which is the default setting).

The above described mechanism allows you to describe the intended capabilities of the endpoint, but without specifying its configuration precisely. This is the recommended approach, because it has a built-in flexibility to cover changes in server configuration or version, protocol improvements etc. An alternative approach (described below) allows more precision.

The [UAEndpointSelectionPolicy.SecurityPolicyUriString](#) property, when not an empty string (empty string is the default), specifies the required security policy URI string of the endpoint. This allows the developer to select the precise security policy, when so required. The static [UASecurityPolicyUriStrings](#) class contains strings that can be used to specify the security policy URI.

### 5.14.2 Trusting Server Instance Certificate

In a secured OPC-UA solution, the OPC server identifies itself to the OPC client using its instance certificate. The OPC client authenticates the certificate and checks whether the communication with that OPC server is authorized. Typically, the process of checking other party's instance certificate is achieved through use of certificate trust lists,

accessible programmatically and by means of UA Configuration Tool (installed with our product, and with OPC-UA downloads provided by OPC Foundation).

In QuickOPC-UA, you can control how server instance certificates are trusted using Certificate Acceptance Policy ([UACertificateAcceptancePolicy](#) object) that you can parameterize in the EngineParameters property on the [EasyUAClient](#) object.

The [TrustedEndpointUrlStrings](#) property contains an array of endpoint URLs that are always trusted, without regard for the certificate provided. By default, this list contains endpoint URLs of the demo OPC UA Sample Server on the Internet ([opcua.demo-this.com](http://opcua.demo-this.com)) and the local OPC UA Sample Server, meaning that you will always be able to use the sample servers from QuickOPC-UA, even if configuration of its instance certificate, or configuration of trusted certificate list for QuickOPC-UA has done been done properly. In a fully secure configuration, this list should be empty.

The [AcceptAnyCertificate](#) property determines whether the client accepts any server certificate, even if a certificate validation error occurs. Setting this property to [true](#) effectively bypasses an important security feature in OPC Unified Architecture. Use it only for testing and development purposes, or if your application does not require the server certificate check.

The type and location of trusted peers certificate store can be controlled by [EasyUAClient.SharedParameters.EngineParameters.TrustedPeersCertificateStore](#) property. By default, QuickOPC-UA uses a standard directory-based store defined by OPC Foundation ("UA Applications").

The [AllowUserAcceptCertificate](#) property determines whether the interactive user can be prompted to and accept a server certificate that has failed other validation checks. This setting has effect only when the current process is running in user interactive mode.

Some finer details regarding the server instance checking process can be controlled by a configuration file, if you correspondingly configure the [EasyUAClient.SharedParameters.EngineParameters.ConfigurationSources](#). For more information, see "Application Configuration" chapter under "Advanced Topics".

## 5.14.3 Providing Client Instance Certificate

Similarly to server instance certificate checking described above, in a secured OPC-UA solution, the OPC client identifies itself to the OPC server using its own instance certificate. The OPC server authenticates the certificate and checks whether the communication with that OPC client is authorized. Typically, the process of checking other party's instance certificate is achieved through use of certificate trust lists, accessible programmatically and by means of UA Configuration Tool (installed with our product, and with OPC-UA downloads provided by OPC Foundation).

Since with QuickOPC-UA you are developing an OPC-UA client application, this client application must therefore typically provide its instance certificate to the OPC servers it is connecting to. QuickOPC-UA attempts to make this process easier by automatically selecting parameters for the certificate, and if the certificate does not exist yet, by creating it and storing into the appropriate certificate store on the computer. Depending on your deployment scenario, you may also create the certificate and put it into the certificate store by some other means, and in that case, you would simply let QuickOPC-UA use that certificate.

### VBScript

---

```
Rem This example demonstrates how to set the application name for the client
certificate.
```

```
Option Explicit

' The configuration object allows access to static behavior.
Dim ClientConfiguration: Set ClientConfiguration =
CreateObject("OpcLabs.EasyOpc.UA.EasyUAClientConfiguration")
WScript.ConnectObject ClientConfiguration, "ClientConfiguration_"
```

```
' Set the application name, which determines the subject of the client certificate.  
' Note that this only works once in each host process.  
ClientConfiguration.SharedParameters.EngineParameters.ApplicationName = "QuickOPC -  
VBScript example application"  
  
' Do something - invoke an OPC read, to trigger some loggable entries.  
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")  
Dim value: value = Client.ReadValue("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853")  
  
' The certificate will be located or created in a directory similar to:  
' C:\Users\All Users\OPC Foundation\CertificateStores\UA Applications\certs\  
' and its subject will be as given by the application name.  
  
WScript.Echo "Processing log entry events for 10 seconds..."  
WScript.Sleep 10*1000  
  
' Event handler for the LogEntry event.  
' Print the loggable entry containing client certificate parameters.  
Sub ClientConfiguration_LogEntry(Sender, e)  
    If e.EventId = 151 Then WScript.Echo e  
End Sub
```

If you let QuickOPC-UA create the certificate and add it to the store, it should be noted that it is a security-sensitive operation that would typically be allowed only to administrators. QuickOPC-UA will attempt to do this if the certificate is not found, which may be when the application is first started on the computer, but if your application is not running with privileges required for this operation, the certificate will not be added.

In order to allow your application be run under usual privileges, and make its behavior more predictable, it is recommended that you assure the generation of the certificate during the application deployment, when administrative privileges are usually required anyway. One way to do this is call the static method `EasyUAClient.Install` during the deployment process. For more information, see the "Application Deployment" chapter.

The type and location of client instance certificate store can be controlled by properties in `EasyUAClient.SharedParameters.EngineParameters`, such as the `ApplicationCertificateStore` property. By default, QuickOPC-UA uses a standard directory-based store defined by OPC Foundation ("MachineDefault").

The `AllowCertificatePrompt` property in `EasyUAClient.SharedParameters.EngineParameters` determines whether the component can interact with the user when checking or creating an application instance certificate. If set to false (the default), no dialogs will be displayed.

Some finer details regarding the server instance checking process, such as the certificate's subject name, can be controlled by a configuration file, if `EasyUAClient.SharedParameters.EngineParameters.ConfigurationSources` is correspondingly configured. For more information, see "Application Configuration" chapter under "Advanced Topics".

Regarding the certificate generation, below we have also listed some pieces of advises that we have offered on our Online Forums:

OPC-UA requires that both parties (client and server) mutually identify themselves using application certificates. The certificates are supposed to be unique for each application (instance), and therefore cannot be a constant part of the "toolkit" (such as QuickOPC), and need to be generated. In order to make this process invisible (in common cases) to the developer, QuickOPC

- a) determines the parameters of the certificate automatically, using values such as the EXE name or the calling assembly name,
- b) attempts to look up the certificate in the certificate store,
- c) if not found, it attempts to create it, and save it into the certificate store.

The algorithm described above does not, however fit well with hosted environments such as IIS, for two reasons:

- It is difficult to automatically determine reasonable and unique parameters for the certificate, because the hosting process is the IIS service, not "your" own application EXE.
- The page processing code in IIS typically runs with low privileges that do not allow it to call the necessary CertificateGenerator utility, and even less to save the new certificate to the store.

For usage in such environments, the recommendation is:

- Pre-generate the application certificate manually, and save it to the certificate store. This will remove the need for the application to create and save the certificate.
- In your application, set your own parameters of the certificate. The application will then use these parameters to look up the certificate.

Application certificate can be generated using Opc.Ua.CertificateGenerator.exe utility, typically located (after QuickOPC installation) at C:\Program Files (x86)\Common Files\OPC Foundation\UA\v1.0\Bin. Run it with "?" to obtain usage instructions.

Setting the certificate parameters is done by modifying properties in EasyUAClient.SharedParameters.EngineParameters (EasyUAClient.SharedParameters is a static property, and you need to set the values before creating the first instance EasyUAClient, in order for it to have the desired effect). The properties of interest are:

- ApplicationCertificateSubject
- ApplicationName
- ApplicationUriString
- ProductUriString

The values should match those used when the certificate was generated, because they (or at least some of them) are used to look up the certificate in the store.

1. [certificate generation works on one computer, does not other] For example, when you first run your application your development computer, you may use the development Web server built-in in Visual Studio. This one uses different accounts and permissions from IIS, and the certificate creates just fine.
2. [certificate generation fails under IIS] Most likely this is due to permissions. IIS settings are VERY restrictive (for good reason), and the defaults won't allow the certification generation or storing it into the store.
3. 5. The default certificate store for OPC-UA applications is not one of the Windows stores, but instead, a file-system (directory) based store. It defaults to "%CommonApplicationData%\OPC Foundation\CertificateStores\UA Applications", where %CommonApplicationData% is e.g. "C:\ProgramData". The best way to inspect the UA certificates is to run Start -> All Programs -> OPC Foundation -> UA SDK 1.01 -> UA Configuration Tool. Then, select the "Manage Certificates" tab, press the drop-down next to "Store Path", select the store I listed above, and press the "View Certificates" button.
3. You should not use the same certificate for multiple applications. And, you should not use the same certificate even for multiple instances of the same application (such as when you deploy the application on multiple computers). Each instance of the application (typically, this means each application/computer combination, but can be even more) should have its own, unique certificate. This is how UA security is meant to work, because there needs to be a way for the server to authenticate each client connection separately and possibly deny the connection to unsecure or compromised clients. Technically, if you violate this and use the same certificate, things will probably work OK on the surface, but it is against the OPC-UA security specs. This is the reason why the ceritificate is generated anew on the computer - and the cause of some troubles.

Actually, while using the CertificateGenerator manually is possible, it would be probably too complicated to try to assemble the right set of parameters so that the generated certificate is then properly found by the client application. Instead, I suggest that you create a small Windows based (e.g. Windows Forms app, or a console app). In it, you set the 4 parameters in EasyUAClient.SharedParameters to meaningful values, and then call a

static method EasyUAClient.Install(). You then run this application with administrator/elevated privileges, and it will create the certificate (verify it with the UA Configuration Tool described above). In your actual client app, set the parameters in precisely the same way before instantiating the EasyUAClient object, and doing so should assure that it will find and use the certificate (still, the Web app needs to have read access to the directory of the cert store - you may have to modify the permissions). The additional app you created may become part of your installation procedure.

## 6 Procedural Coding Model

### 6.1 Procedural Coding Model for OPC Data (Classic and UA)

Data Access deals with the representation and use of automation data in OPC Servers. 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](#) or the [EasyUAClient](#) object.

#### 6.1.1 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 Classic items, or value attributes of OPC UA nodes), or from the OPC server itself (getting OPC Classic 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.

##### 6.1.1.1 Reading from OPC Classic 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).

###### A single item

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.

###### C#

```
// This example shows how to read a single item, and display its value, timestamp  
and quality.  
  
using System;  
using OpcLabs.EasyOpc.DataAccess;  
  
namespace DocExamples  
{  
    namespace _EasyDAClient  
    {  
        partial class ReadItem  
        {  
            public static void Main()  
            {  
                var easyDAClient = new EasyDAClient();  
  
                DAVtq vtq = easyDAClient.ReadItem("", "OPCLabs.KitServer.2",  
"Simulation.Random");
```

```
        Console.WriteLine("Vtq: {0}", vtq);
    }
}
}
```

## VB.NET

```
' This example shows how to read a single item, and display its value, timestamp and quality.
```

```
Imports OpcLabs.EasyOpc.DataAccess

Namespace EasyDAClient
    Partial Friend Class ReadItem
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            Dim vtq As DAVtq = easyDAClient.ReadItem("", "OPCLabs.KitServer.2",
"Simulation.Random")

            Console.WriteLine("Vtq: {0}", vtq)
        End Sub
    End Class
End Namespace
```

## JScript

```
// This example shows how to read a single item, and display its value, timestamp and quality.
```

```
var Client = new ActiveXObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient");
var VTQ = Client.ReadItem("", "OPCLabs.KitServer.2", "Simulation.Random");
WScript.Echo("VTQ.ToString(): " + VTQ.ToString());
```

## Object Pascal

```
// This example shows how to read a single item, and display its value, timestamp and quality.

class procedure ReadItem.Main;
var
  Client: TEasyDAClient;
  Vtq: _DAVtq;
begin
  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);

  Vtq := Client.ReadItem('', 'OPCLabs.KitServer.2', 'Simulation.Random');

  // Display results
  WriteLn('Vtq: ', Vtq.ToString());
end;
```

## VBScript

```
Rem This example shows how to read a single item, and display its value, timestamp and quality.
```

```
Option Explicit
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim Vtq: Set Vtq = Client.ReadItem("", "OPCLabs.KitServer.2", "Simulation.Random")
WScript.Echo "Vtq: " & Vtq
```



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

## C#

```
// This example shows how to read a single item using a browse path, and display its value, timestamp and quality.
```

```
using System;
using OpcLabs.EasyOpc;
using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        partial class ReadItem
        {
            public static void BrowsePath()
            {
                var easyDAClient = new EasyDAClient();

                DAVtq vtq = easyDAClient.ReadItem(
                    new ServerDescriptor("", "OPCLabs.KitServer.2"),
                    new DAItemDescriptor(null, "/Simulation/Random"));

                Console.WriteLine("Vtq: {0}", vtq);
            }
        }
    }
}
```

## Multiple items

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.

## C#

```
// This example shows how to read 4 items at once, and display their values, timestamps and qualities.
using System;
using System.Diagnostics;
using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        partial class ReadMultipleItems
```

```

{
    public static void Main()
    {
        var easyDAClient = new EasyDAClient();

        DAVtqResult[] vtqResults =
easyDAClient.ReadMultipleItems("OPCLabs.KitServer.2",
            new DAItemDescriptor[] { "Simulation.Random", "Trends.Ramp (1
min)", "Trends.Sine (1 min)", "Simulation.Register_I4" });

        for (int i = 0; i < vtqResults.Length; i++)
        {
            Debug.Assert(vtqResults[i] != null);
            Console.WriteLine("vtqResult[{0}].Vtq: {1}", i,
vtqResults[i].Vtq);
        }
    }
}
}

```

## VB.NET

---

' This example shows how to read 4 items at once, and display their values, timestamps and qualities.

```

Imports OpcLabs.EasyOpc.DataAccess

Namespace _EasyDAClient
    Partial Friend Class ReadMultipleItems
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            Dim vtqResults() As DAVtqResult =
easyDAClient.ReadMultipleItems("OPCLabs.KitServer.2", New DAItemDescriptor()
{"Simulation.Random", "Trends.Ramp (1 min)", "Trends.Sine (1 min)",
"Simulation.Register_I4"})

            For i As Integer = 0 To vtqResults.Length - 1
                Debug.Assert(vtqResults(i) IsNot Nothing)
                Console.WriteLine("vtqResult[{0}].Vtq: {1}", i, vtqResults(i).Vtq)
            Next i
        End Sub
    End Class
End Namespace

```

## Object Pascal

---

// This example shows how to read 4 items at once, and display their values, timestamps and qualities.

```

class procedure ReadMultipleItems.Main;
var
    Arguments: OleVariant;
    Client: TEasyDAClient;
    I: Cardinal;
    ReadItemArguments1: _DAReadItemArguments;
    ReadItemArguments2: _DAReadItemArguments;
    ReadItemArguments3: _DAReadItemArguments;
    ReadItemArguments4: _DAReadItemArguments;
    Result: _DAVtqResult;

```

```
Results: OleVariant;
begin
  ReadItemArguments1 := CoDAReadItemArguments.Create;
  ReadItemArguments1.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments1.ItemDescriptor.ItemID := 'Simulation.Random';

  ReadItemArguments2 := CoDAReadItemArguments.Create;
  ReadItemArguments2.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments2.ItemDescriptor.ItemID := 'Trends.Ramp (1 min)';

  ReadItemArguments3 := CoDAReadItemArguments.Create;
  ReadItemArguments3.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments3.ItemDescriptor.ItemID := 'Trends.Sine (1 min)';

  ReadItemArguments4 := CoDAReadItemArguments.Create;
  ReadItemArguments4.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments4.ItemDescriptor.ItemID := 'Simulation.Register_I4';

  Arguments := VarArrayCreate([0, 3], varVariant);
  Arguments[0] := ReadItemArguments1;
  Arguments[1] := ReadItemArguments2;
  Arguments[2] := ReadItemArguments3;
  Arguments[3] := ReadItemArguments4;

  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);

  TVarData(Results).VType := varArray or varVariant;
  TVarData(Results).VArray := PVarArray(
    Client.ReadMultipleItems(PSafeArray(TVarData(Arguments).VArray)));

  // Display results
  for I := VarArrayLowBound(Results, 1) to VarArrayHighBound(Results, 1) do
  begin
    Result := IInterface(Results[I]) as _DAVtqResult;
    WriteLn('results(', i, ') .Vtq.ToString(): ', Result.Vtq.ToString());
  end;
end;
```

## VBScript

---

Rem This example shows how to read 4 items at once, and display their values, timestamps and qualities.

### Option Explicit

```
Dim ReadItemArguments1: Set ReadItemArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAReadItemArguments")
ReadItemArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments1.ItemDescriptor.ItemID = "Simulation.Random"

Dim ReadItemArguments2: Set ReadItemArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAReadItemArguments")
ReadItemArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments2.ItemDescriptor.ItemID = "Trends.Ramp (1 min)"

Dim ReadItemArguments3: Set ReadItemArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAReadItemArguments")
ReadItemArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments3.ItemDescriptor.ItemID = "Trends.Sine (1 min)"
```

```
Dim ReadItemArguments4: Set ReadItemArguments4 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAReadItemArguments")
ReadItemArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments4.ItemDescriptor.ItemID = "Simulation.Register_I4"

Dim arguments(3)
Set arguments(0) = ReadItemArguments1
Set arguments(1) = ReadItemArguments2
Set arguments(2) = ReadItemArguments3
Set arguments(3) = ReadItemArguments4

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim results: results = Client.ReadMultipleItems(arguments)

Dim i: For i = LBound(results) To UBound(results)
    WScript.Echo "results(" & i & ").Vtq.ToString(): " & results(i).Vtq.ToString()
Next
```

## Read parameters

Note: In QuickOPC.NET, you can use the optional argument of type `DAReadParameters`, and then set the `ValueAge` property in it to control how "old" may be the values you receive by reading from OPC items. With the `DataSource` property, you can also specify that you want values from the device, or from cache. You can also use the pre-defined `DAReadParameters.CacheSource` and `DAReadParameters.DeviceSource` constants to specify the read parameters. There are also overloads of `ReadXXXX` methods that accept an integer `valueAge` argument.

Be aware that it is physically impossible for any system to always obtain fully up-to-date values.

### 6.1.1.1.1 Reading just the value

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**.

#### A single item

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.

Note: In OPC XML-DA, the `ReadItemValue` method does not wait for the **Good** quality. It makes a single read, and if the value obtained does not have a **Good** quality, the method returns an error.

#### JScript

```
// This example shows how to read values of a single item, and display it.

var Client = new ActiveXObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient");
var value = Client.ReadItemValue("", "OPCLabs.KitServer.2", "Simulation.Random");
WScript.Echo("value: " + value);
```

#### Object Pascal

```
// This example shows how to read value of a single item, and display it.

class procedure ReadItemValue.Main;
var
```

```
Client: TEasyDAClient;
Value: OleVariant;
begin
  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);

  Value := Client.ReadItemValue('', 'OPCLabs.KitServer.2', 'Simulation.Random');

  // Display results
  WriteLn('Value: ', Value);
end;
```

## Python

```
# This example shows how to read value of a single item, and display it.

import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.DataAccess.EasyDAClient')

# Perform the operation
value = client.ReadItemValue('', 'OPCLabs.KitServer.2', 'Demo.Single')

# Display results
print('value: ', value)
```

## VBScript

```
Rem This example shows how to read value of a single item, and display it.

Option Explicit

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim value: value = Client.ReadItemValue("", "OPCLabs.KitServer.2",
"Simulation.Random")
WScript.Echo "value: " & value
```

## Multiple items

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.

Note: In OPC XML-DA, the [ReadMultipleItemValues](#) method does not wait for the **Good** quality. It makes a single read, and if the value obtained does not have a **Good** quality, the method returns an error for such an item.

## Object Pascal

```
// This example shows how to read values of 4 items at once, and display them.

class procedure ReadMultipleItemValues.Main;
var
  Arguments: OleVariant;
  Client: TEasyDAClient;
  I: Cardinal;
  ReadItemArguments1: _DAReadItemArguments;
  ReadItemArguments2: _DAReadItemArguments;
  ReadItemArguments3: _DAReadItemArguments;
```

```

ReadItemArguments4: _DAReadItemArguments;
Result: _ValueResult;
Results: OleVariant;
begin
  ReadItemArguments1 := CoDAReadItemArguments.Create;
  ReadItemArguments1.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments1.ItemDescriptor.ItemID := 'Simulation.Random';

  ReadItemArguments2 := CoDAReadItemArguments.Create;
  ReadItemArguments2.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments2.ItemDescriptor.ItemID := 'Trends.Ramp (1 min)';

  ReadItemArguments3 := CoDAReadItemArguments.Create;
  ReadItemArguments3.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments3.ItemDescriptor.ItemID := 'Trends.Sine (1 min)';

  ReadItemArguments4 := CoDAReadItemArguments.Create;
  ReadItemArguments4.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ReadItemArguments4.ItemDescriptor.ItemID := 'Simulation.Register_I4';

  Arguments := VarArrayCreate([0, 3], varVariant);
  Arguments[0] := ReadItemArguments1;
  Arguments[1] := ReadItemArguments2;
  Arguments[2] := ReadItemArguments3;
  Arguments[3] := ReadItemArguments4;

  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);

  TVarData(Results).VType := varArray or varVariant;
  TVarData(Results).VArray := PVarArray(
    Client.ReadMultipleItemValues(PSafeArray(TVarData(Arguments).VArray)));

  // Display results
  for I := VarArrayLowBound(Results, 1) to VarArrayHighBound(Results, 1) do
  begin
    Result := IInterface(Results[I]) as _ValueResult;
    WriteLn('results(', i, ').Value: ', Result.Value);
  end;
end;

```

## VBScript

---

Rem This example shows how to read values of 4 items at once, and display them.

Option Explicit

```

Dim ReadItemArguments1: Set ReadItemArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAReadItemArguments")
ReadItemArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments1.ItemDescriptor.ItemID = "Simulation.Random"

Dim ReadItemArguments2: Set ReadItemArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAReadItemArguments")
ReadItemArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments2.ItemDescriptor.ItemID = "Trends.Ramp (1 min)"

Dim ReadItemArguments3: Set ReadItemArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAReadItemArguments")
ReadItemArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments3.ItemDescriptor.ItemID = "Trends.Sine (1 min)"

```

```
Dim ReadItemArguments4: Set ReadItemArguments4 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAReadItemArguments")
ReadItemArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ReadItemArguments4.ItemDescriptor.ItemID = "Simulation.Register_I4"

Dim arguments(3)
Set arguments(0) = ReadItemArguments1
Set arguments(1) = ReadItemArguments2
Set arguments(2) = ReadItemArguments3
Set arguments(3) = ReadItemArguments4

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim results: results = Client.ReadMultipleItemValues(arguments)

Dim i: For i = LBound(results) To UBound(results)
    WScript.Echo "results(" & i & ").Value: " & results(i).Value
Next
```



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.

## 6.1.1.1.2 Reading in OPC XML-DA

In OPC XML-DA, reading or writing is in principle no different from COM-based OPC. You just need to specify the OPC server using its URL instead of the ProgID or CLSID, and obey the rules for identification of the items (which are server-dependent). Sometimes, you may need an additional piece of information to identify nodes in OPC XML-DA address space; for more information see **Identifying information in OPC XML (Section 5.12)**.

### C#

```
// This example shows how to read 4 items from an OPC XML-DA server at once, and
// display their values, timestamps
// and qualities.
using System;
using System.Diagnostics;
using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        partial class ReadMultipleItems
        {
            public static void Xml()
            {
                var easyDAClient = new EasyDAClient();

                DAVtqResult[] vtqResults = easyDAClient.ReadMultipleItems(
                    new ServerDescriptor { UrlString = "http://opcxml.demo-
this.com/XmlDaSampleServer/Service.asmx" },
                    new DAItemDescriptor[]
                    {
                        "Dynamic/Analog Types/Double",
                        "Dynamic/Analog Types/Double[]",
                        "Dynamic/Analog Types/Int",
                        "SomeUnknownItem"
                    });
            }
        }
}
```

```
        for (int i = 0; i < vtqResults.Length; i++)
    {
        Debug.Assert(vtqResults[i] != null);

        if (vtqResults[i].Exception != null)
        {
            Console.WriteLine("vtqResults[{0}].Exception: {1}", i,
vtqResults[i].Exception);
            continue;
        }
        Console.WriteLine("vtqResults[{0}].Vtq: {1}", i,
vtqResults[i].Vtq);
    }
}
}
```

## 6.1.1.2 Getting OPC Classic Property Values

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

### A single property

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.

#### C#

```
// This example shows how to get a value of a single OPC property.
//
// Note that some properties may not have a useful value initially (e.g. until the
// item is activated in a group), which also the
// case with Timestamp property as implemented by the demo server. This behavior is
// server-dependent, and normal. You can run
// IEasyDAClient.ReadItemValue.Main.vbs shortly before this example, in order to
// obtain better property values. Your code may
// also subscribe to the item in order to assure that it remains active.

using System;
using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        partial class GetPropertyValue
        {
            public static void Main()
            {
                var easyDAClient = new EasyDAClient();
```

```
        object value = easyDAClient.GetPropertyValues("",  
"OPCLabs.KitServer.2", "Simulation.Random",  
DAPropertyIds.Timestamp);  
  
        Console.WriteLine(value);  
    }  
}  
}  
}
```

## VB.NET

```
' This example shows how to get a value of a single OPC property.  
'  
' Note that some properties may not have a useful value initially (e.g. until the  
item is activated in a group), which also the  
' case with Timestamp property as implemented by the demo server. This behavior is  
server-dependent, and normal. You can run  
' IEasyDAClient.ReadItemValue.Main.vbs shortly before this example, in order to  
obtain better property values. Your code may  
' also subscribe to the item in order to assure that it remains active.  
  
Imports OpcLabs.EasyOpc.DataAccess  
  
Namespace _EasyDAClient  
    Partial Friend Class GetPropertyValue  
        Public Shared Sub Main()  
            Dim easyDAClient = New EasyDAClient()  
  
            Dim value As Object = easyDAClient.GetPropertyValues("",  
"OPCLabs.KitServer.2", "Simulation.Random", DAPropertyIds.Timestamp)  
  
            Console.WriteLine(value)  
        End Sub  
    End Class  
End Namespace
```

## VBScript

```
Rem This example shows how to get a value of a single OPC property.  
Rem  
Rem Note that some properties may not have a useful value initially (e.g. until the  
item is activated in a group), which also the  
Rem case with Timestamp property as implemented by the demo server. This behavior is  
server-dependent, and normal. You can run  
Rem IEeasyDAClient.ReadItemValue.Main.vbs shortly before this example, in order to  
obtain better property values. Your code may  
Rem also subscribe to the item in order to assure that it remains active.
```

```
Option Explicit  
  
Const Timestamp = 4  
  
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")  
  
Dim value: value = Client.GetPropertyValues("", "OPCLabs.KitServer.2",  
"Simulation.Random", Timestamp)  
  
WScript.Echo value
```



In QuickOPC.NET, you can also pass in the [ServerDescriptor](#) in place of the machine name and server class strings.



In addition to basic methods, there are also many property-related methods. Please refer to a chapter in this document that describes EasyOPC.NET Extensions.

## Multiple properties

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).

### C#

```
// This example shows how to obtain a data type of all OPC items under a branch.
using System;
using System.Linq;
using OpcLabs.BaseLib.ComInterop;
using OpcLabs.BaseLib.OperationModel;
using OpcLabs.EasyOpc;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.AddressSpace;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        class GetMultiplePropertyValues
        {
            public static void DataType()
            {
                var easyDAClient = new EasyDAClient();
                ServerDescriptor serverDescriptor = "OPCLabs.KitServer.2";

                // Browse for all leaves under the "Simulation" branch
                DANodeElementCollection nodeElementCollection =
                    easyDAClient.BrowseLeaves(serverDescriptor, "Simulation");

                // Create list of node descriptors, one for each leaf obtained
                DANodeDescriptor[] nodeDescriptorArray = nodeElementCollection
                    .Where(element => !element.IsHint) // filter out hint leafs
that do not represent real OPC items (rare)
                    .Select(element => new DANodeDescriptor(element))
                    .ToArray();

                // Get the value of DataType property; it is a 16-bit signed integer
                ValueResult[] valueResultArray =
                    easyDAClient.GetMultiplePropertyValues(serverDescriptor,
                        nodeDescriptorArray, DAPropertyIds.DataType);

                for (int i = 0; i < valueResultArray.Length; i++)
                {
                    DANodeDescriptor nodeDescriptor = nodeDescriptorArray[i];

                    // Check if there has been an error getting the property value
                    ValueResult valueResult = valueResultArray[i];
                    if (valueResult.Exception != null)
                    {

```

```
        Console.WriteLine("{0}: *** {1}", nodeDescriptor.NodeId,
valueResult.Exception.Message);
        continue;
    }

    // Convert the data type to VarType
    var varType = (VarType)(short)valueResult.Value;

    // Display the obtained data type
    Console.WriteLine("{0}: {1}", nodeDescriptor.ItemId, varType);
}
}
}
}
}
```

## Object Pascal

---

```
// This example shows how to get value of multiple OPC properties.
//
// Note that some properties may not have a useful value initially (e.g. until the
// item is activated in a group), which also the
// case with Timestamp property as implemented by the demo server. This behavior is
// server-dependent, and normal. You can run
// IEasyDAClient.ReadMultipleItemValues.Main.vbs shortly before this example, in
// order to obtain better property values. Your
// code may also subscribe to the items in order to assure that they remain active.

class procedure GetMultiplePropertyValues.Main;
var
  Arguments: OleVariant;
  Client: TEeasyDAClient;
  I: Cardinal;
  PropertyArguments1: _DAPropertyArguments;
  PropertyArguments2: _DAPropertyArguments;
  PropertyArguments3: _DAPropertyArguments;
  PropertyArguments4: _DAPropertyArguments;
  Result: _ValueResult;
  Results: OleVariant;
begin
  PropertyArguments1 := CoDAPropertyArguments.Create;
  PropertyArguments1.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  PropertyArguments1.NodeDescriptor.ItemID := 'Simulation.Random';
  PropertyArguments1.PropertyDescriptor.PropertyId.NumericalValue :=
  DAPropertyIds_Timestamp;

  PropertyArguments2 := CoDAPropertyArguments.Create;
  PropertyArguments2.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  PropertyArguments2.NodeDescriptor.ItemID := 'Simulation.Random';
  PropertyArguments2.PropertyDescriptor.PropertyId.NumericalValue :=
  DAPropertyIds_AccessRights;

  PropertyArguments3 := CoDAPropertyArguments.Create;
  PropertyArguments3.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  PropertyArguments3.NodeDescriptor.ItemID := 'Trends.Ramp (1 min)';
  PropertyArguments3.PropertyDescriptor.PropertyId.NumericalValue :=
  DAPropertyIds_Timestamp;

  PropertyArguments4 := CoDAPropertyArguments.Create;
  PropertyArguments4.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  PropertyArguments4.NodeDescriptor.ItemID := 'Trends.Ramp (1 min');
```

```
PropertyArguments4.PropertyDescriptor.PropertyId.NumericalValue :=  
DAPropertyIds_AccessRights;  
  
Arguments := VarArrayCreate([0, 3], varVariant);  
Arguments[0] := PropertyArguments1;  
Arguments[1] := PropertyArguments2;  
Arguments[2] := PropertyArguments3;  
Arguments[3] := PropertyArguments4;  
  
// Instantiate the client object  
Client := TEasyDAClient.Create(nil);  
  
TVarData(Results).VType := varArray or varVariant;  
TVarData(Results).VArray := PVarArray(  
    Client.GetMultiplePropertyValues(PSafeArray(TVarData(Arguments).VArray)));  
  
// Display results  
for I := VarArrayLowBound(Results, 1) to VarArrayHighBound(Results, 1) do  
begin  
    Result := IInterface(Results[I]) as _ValueResult;  
    WriteLn('results(', i, ').Value: ', Result.Value);  
end;  
end;
```

## VBScript

---

```
Rem This example shows how to get value of multiple OPC properties.  
Rem  
Rem Note that some properties may not have a useful value initially (e.g. until the  
item is activated in a group), which also the  
Rem case with Timestamp property as implemented by the demo server. This behavior is  
server-dependent, and normal. You can run  
Rem IEasyDAClient.ReadMultipleItemValues.Main.vbs shortly before this example, in  
order to obtain better property values. Your  
Rem code may also subscribe to the items in order to assure that they remain active.
```

```
Option Explicit
```

```
Const Timestamp = 4  
Const AccessRights = 5  
  
Dim PropertyArguments1: Set PropertyArguments1 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")  
PropertyArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
PropertyArguments1.NodeDescriptor.ItemID = "Simulation.Random"  
PropertyArguments1.PropertyDescriptor.PropertyID.NumericalValue = Timestamp  
  
Dim PropertyArguments2: Set PropertyArguments2 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")  
PropertyArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
PropertyArguments2.NodeDescriptor.ItemID = "Simulation.Random"  
PropertyArguments2.PropertyDescriptor.PropertyID.NumericalValue = AccessRights  
  
Dim PropertyArguments3: Set PropertyArguments3 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")  
PropertyArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
PropertyArguments3.NodeDescriptor.ItemID = "Trends.Ramp (1 min)"  
PropertyArguments3.PropertyDescriptor.PropertyID.NumericalValue = Timestamp  
  
Dim PropertyArguments4: Set PropertyArguments4 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")
```

```
PropertyArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
PropertyArguments4.NodeDescriptor.ItemID = "Trends.Ramp (1 min)"
PropertyArguments4.PropertyDescriptor.PropertyID.NumericalValue = AccessRights

Dim arguments(3)
Set arguments(0) = PropertyArguments1
Set arguments(1) = PropertyArguments2
Set arguments(2) = PropertyArguments3
Set arguments(3) = PropertyArguments4

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim results: results = Client.GetMultiplePropertyValues(arguments)

Dim i: For i = LBound(results) To UBound(results)
    WScript.Echo "results(" & i & ").Value: " & results(i).Value
Next
```

And, a similar code with added error handling:

## VBScript

```
Rem This example shows how to get value of multiple OPC properties, and handle
errors.
```

```
Option Explicit

Const Timestamp = 4
Const AccessRights = 5
Const SomeInvalidPropertyId = 999111

Dim PropertyArguments1: Set PropertyArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")
PropertyArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
PropertyArguments1.NodeDescriptor.ItemID = "Simulation.Random"
PropertyArguments1.PropertyDescriptor.PropertyID.NumericalValue = Timestamp

Dim PropertyArguments2: Set PropertyArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")
PropertyArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
PropertyArguments2.NodeDescriptor.ItemID = "SomeInvalidItem"
PropertyArguments2.PropertyDescriptor.PropertyID.NumericalValue = AccessRights

Dim PropertyArguments3: Set PropertyArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")
PropertyArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
PropertyArguments3.NodeDescriptor.ItemID = "Trends.Ramp (1 min)"
PropertyArguments3.PropertyDescriptor.PropertyID.NumericalValue =
SomeInvalidPropertyId

Dim PropertyArguments4: Set PropertyArguments4 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAPropertyArguments")
PropertyArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
PropertyArguments4.NodeDescriptor.ItemID = "Trends.Ramp (1 min)"
PropertyArguments4.PropertyDescriptor.PropertyID.NumericalValue = AccessRights

Dim arguments(3)
Set arguments(0) = PropertyArguments1
Set arguments(1) = PropertyArguments2
Set arguments(2) = PropertyArguments3
Set arguments(3) = PropertyArguments4
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim results: results = Client.GetMultiplePropertyValues(arguments)

Dim i: For i = LBound(results) To UBound(results)
    If results(i).Exception Is Nothing Then
        WScript.Echo "results(" & i & ").Value: " & results(i).Value
    Else
        WScript.Echo "results(" & i & ").Exception.Message: " &
        results(i).Exception.Message
    End If
Next
```

## 6.1.1.3 Reading Attributes of OPC UA Nodes

In OPC UA Data Access, reading data from attributes of OPC nodes is one of the most common tasks. The OPC server generally provides data for any OPC attribute in form of a Value, Timestamps and Status Code combination (see chapter "OPC Attribute Data").

### A single node and attribute

If you want to read the current attribute data from an attribute of an OPC node, call the `Read` method. You pass in individual arguments for the endpoint descriptor and the node ID. You will receive back a `UAAttributeData` object holding the current value, timestamps, and status code for the specified OPC node and attribute. The `Read` method returns the current attribute data, regardless of the actual status code. You may receive an Uncertain or even Bad status (and no usable data value), and your code needs to deal with such situations accordingly.

#### C#

```
// This example shows how to read and display data of an attribute (value,
timestamps, and status code).
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class Read
        {
            public static void Main()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Obtain attribute data. By default, the Value attribute of a node
                // will be read.
                UAAttributeData attributeData = easyUAClient.Read(
                    "http://opcua.demo-this.com:51211/UA/SampleServer", // or
                    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                    "nsu=http://test.org/UA/Data/;i=10853");

                // Display results
                Console.WriteLine("Value: {0}", attributeData.Value);
                Console.WriteLine("ServerTimestamp: {0}",
                    attributeData.ServerTimestamp);
                Console.WriteLine("SourceTimestamp: {0}",
                    attributeData.SourceTimestamp);
            }
        }
    }
}
```

```
        Console.WriteLine("StatusCode: {0}", attributeData.StatusCode);

        // Example output:
        //
        //Value: -2.230064E-31
        //ServerTimestamp: 11/6/2011 1:34:30 PM
        //SourceTimestamp: 11/6/2011 1:34:30 PM
        //StatusCode: Good
    }
}
}
```

## VB.NET

```
' This example shows how to read and display data of an attribute (value,
'timestamps, and status code).
Imports OpcLabs.EasyOpc.UA

Namespace _EasyUAClient
    Friend Class Read
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Obtain attribute data. By default, the Value attribute of a node will
            be read.
            Dim attributeData As UAAttributeData =
easyUAClient.Read("http://opcua.demo-this.com:51211/UA/SampleServer", _

"nsu=http://test.org/UA/Data/;i=10853") ' or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"

            ' Display results
            Console.WriteLine("Value: {0}", attributeData.Value)
            Console.WriteLine("ServerTimestamp: {0}", attributeData.ServerTimestamp)
            Console.WriteLine("SourceTimestamp: {0}", attributeData.SourceTimestamp)
            Console.WriteLine("StatusCode: {0}", attributeData.StatusCode)

            ' Example output:
            '
            'Value: -2.230064E-31
            'ServerTimestamp: 11/6/2011 1:34:30 PM
            'SourceTimestamp: 11/6/2011 1:34:30 PM
            'StatusCode: Good
        End Sub
    End Class
End Namespace
```

## C++

```
// This example shows how to read and display data of an attribute (value,
'timestamps, and status code).

#include "stdafx.h"
#include "_EasyUAClient.Read.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;
```

```

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void Read::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

            // Obtain attribute data. By default, the Value attribute of a node will
            be read.
            _UAAttributeDataPtr AttributeDataPtr = ClientPtr->Read(
                L"http://opcua.demo-this.com:51211/UA/SampleServer",
                L"nsu=http://test.org/UA/Data/;i=10853"); // or
            "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"

            // Display results
            _variant_t vString;
            vString.ChangeType(VT_BSTR, &AttributeDataPtr->Value);
            _tprintf(_T("Value: %s\n"), CW2T((bstr_t)vString));
            vString.ChangeType(VT_BSTR, &_variant_t(AttributeNamePtr-
>ServerTimestamp, VT_DATE));
            _tprintf(_T("ServerTimestamp: %s\n"), CW2T((bstr_t)vString));
            vString.ChangeType(VT_BSTR, &_variant_t(AttributeNamePtr-
>SourceTimestamp, VT_DATE));
            _tprintf(_T("SourceTimestamp: %s\n"), CW2T((bstr_t)vString));
            _tprintf(_T("StatusCode: %s\n"), CW2T(AttributeDataPtr->StatusCode-
>ToString));

            // Example output:
            //
            //Value: -2.230064E-31
            //ServerTimestamp: 11/6/2011 1:34:30 PM
            //SourceTimestamp: 11/6/2011 1:34:30 PM
            //StatusCode: Good
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}

```

}

## PHP

```
// This example shows how to read and display data of an attribute (value,
// timestamps, and status code).

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Obtain attribute data. By default, the Value attribute of a node will be read.
$AttributeData = $Client->Read(
    "http://opcua.demo-this.com:51211/UA/SampleServer",
    "nsu=http://test.org/UA/Data/;i=10853"); // or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"

// Display results
printf("Value: %s\n", $AttributeData->Value);
printf("ServerTimestamp: %s\n", $AttributeData->ServerTimestamp);
printf("SourceTimestamp: %s\n", $AttributeData->SourceTimestamp);
printf("StatusCode: %s\n", $AttributeData->StatusCode);

// Example output:
//
//Value: -2.230064E-31
//ServerTimestamp: 11/6/2011 1:34:30 PM
//SourceTimestamp: 11/6/2011 1:34:30 PM
//StatusCode: Good
```

## Python

```
# This example shows how to read and display data of an attribute (value,
// timestamps, and status code).

import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.UA.EasyUAClient')

# Obtain attribute data. By default, the Value attribute of a node will be read.
attributeData = client.Read('http://opcua.demo-this.com:51211/UA/SampleServer', # or
    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
    'nsu=http://test.org/UA/Data/;i=10853')

# Display results
print('Value: ', attributeData.Value)
print('ServerTimestamp: ', attributeData.ServerTimestamp)
print('SourceTimestamp: ', attributeData.SourceTimestamp)
print('StatusCode: ', attributeData.StatusCode)

# Example output:
#
#Value: -2.230064E-31
#ServerTimestamp: 11/6/2011 1:34:30 PM
#SourceTimestamp: 11/6/2011 1:34:30 PM
#StatusCode: Good
```

## Visual Basic (VB 6.)

Rem This example shows how to read and display data of an attribute (value, timestamps, and status code).

```
Private Sub Read_Main_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    ' Obtain attribute data. By default, the Value attribute of a node will be read.
    Dim AttributeData As UAAttributeData
    Set AttributeData = Client.Read("http://opcua.demo-
this.com:51211/UA/SampleServer", _
                                    "nsu=http://test.org/UA/Data/;i=10853") ' or
    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"

    ' Display results
    OutputText = OutputText & "Value: " & AttributeData.value & vbCrLf
    OutputText = OutputText & "ServerTimestamp: " & AttributeData.ServerTimestamp &
    vbCrLf
    OutputText = OutputText & "SourceTimestamp: " & AttributeData.SourceTimestamp &
    vbCrLf
    OutputText = OutputText & "StatusCode: " & AttributeData.StatusCode & vbCrLf

    ' Example output:
    '
    'Value: -2.230064E-31
    'ServerTimestamp: 11/6/2011 1:34:30 PM
    'SourceTimestamp: 11/6/2011 1:34:30 PM
    'StatusCode: Good
End Sub
```

## VBScript

---

Rem This example shows how to read and display data of an attribute (value, timestamps, and status code).

```
Option Explicit

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Obtain attribute data. By default, the Value attribute of a node will be read.
Dim AttributeData: Set AttributeData = Client.Read("http://opcua.demo-
this.com:51211/UA/SampleServer", _

"nsu=http://test.org/UA/Data/;i=10853") ' or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"

' Display results
WScript.Echo "Value: " & AttributeData.Value
WScript.Echo "ServerTimestamp: " & AttributeData.ServerTimestamp
WScript.Echo "SourceTimestamp: " & AttributeData.SourceTimestamp
WScript.Echo "StatusCode: " & AttributeData.StatusCode

' Example output:
'
'Value: -2.230064E-31
'ServerTimestamp: 11/6/2011 1:34:30 PM
'SourceTimestamp: 11/6/2011 1:34:30 PM
'StateCode: Good
```

If the attribute ID is not specified in the method call, the method will read from the **Value** attribute. There are also other overloads of the [Read](#) method that allow you to pass in the arguments in a different way, and with more information in them. For example, you can pass in a [UANodeArguments](#) object, and an attribute ID.

## Multiple nodes or attributes

For reading attribute data of multiple attributes (of the same node or of different nodes) simultaneously in an efficient manner, call the [ReadMultiple](#) method (instead of multiple [Read](#) calls in a loop). You pass in an array of [UAReadArguments](#) objects, and you will receive back an array of [UAAttributeDataResult](#) objects.

### C#

---

```
// This example shows how to read data (value, timestamps, and status code) of 3
// attributes at once. In this example,
// we are reading a Value attribute of 3 different nodes, but the method can also be
// used to read multiple attributes
// of the same node.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class ReadMultiple
        {
            public static void Main()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Obtain attribute data. By default, the Value attributes of the
                // nodes will be read.
                UAAttributeDataResult[] attributedataArray =
                easyUAClient.ReadMultiple(new[]
                {
                    new UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845"),
                    new UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853"),
                    new UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855")
                });

                // Display results
                foreach (UAAttributeDataResult attributeDataResult in
attributedataArray)
                    Console.WriteLine("AttributeData: {0}",
attributeDataResult.AttributeData);

                // Example output:
                //
                //AttributeData: 51 {System.Int16} @11/6/2011 1:49:19 PM @11/6/2011
                1:49:19 PM; Good
                //AttributeData: -1993984 {System.Single} @11/6/2011 1:49:19 PM
                @11/6/2011 1:49:19 PM; Good
                //AttributeData: Yellow% Dragon Cat) White Blue Dog# Green Banana-
```

```
{System.String} @11/6/2011 1:49:19 PM @11/6/2011 1:49:19 PM; Good
    }
}
}
```

## VB.NET

```
' This example shows how to read data (value, timestamps, and status code) of 3
' attributes at once. In this example,
' we are reading a Value attribute of 3 different nodes, but the method can also be
' used to read multiple attributes
' of the same node.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class ReadMultiple
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Obtain attribute data. By default, the Value attributes of the nodes
            ' will be read.
            Dim attributeDataResultArray() As UAAttributeDataResult =
                easyUAClient.ReadMultiple(New UAReadArguments() _
                {
                    _
                    New UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845"),
                    _
                    New UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853"),
                    _
                    New UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855")
                })

            ' Display results
            For Each attributeDataResult As UAAttributeDataResult In
                attributeDataResultArray
                    Console.WriteLine("AttributeData: {0}",
                        attributeDataResult.AttributeData)
                    Next attributeDataResult

            ' Example output:
            '
            'AttributeData: 51 {System.Int16} @11/6/2011 1:49:19 PM @11/6/2011
            1:49:19 PM; Good
            'AttributeData: -1993984 {System.Single} @11/6/2011 1:49:19 PM
            @11/6/2011 1:49:19 PM; Good
            'AttributeData: Yellow% Dragon Cat) White Blue Dog# Green Banana-
            {System.String} @11/6/2011 1:49:19 PM @11/6/2011 1:49:19 PM; Good
        End Sub
    End Class
End Namespace
```

## C++

```
// This example shows how to read the attributes of 4 OPC-UA nodes at once, and
// display the results.

#include "stdafx.h"
#include <atlsafe.h>
```

```
#include "_EasyUAClient.ReadMultiple.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorelib
using namespace mscorelib;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void ReadMultiple::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            _UAReadArgumentsPtr ReadArguments1Ptr(_uuidof(UAReadArguments));
            ReadArguments1Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments1Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;i=10853";

            _UAReadArgumentsPtr ReadArguments2Ptr(_uuidof(UAReadArguments));
            ReadArguments2Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments2Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;i=10845";

            _UAReadArgumentsPtr ReadArguments3Ptr(_uuidof(UAReadArguments));
            ReadArguments3Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments3Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;i=10304";

            _UAReadArgumentsPtr ReadArguments4Ptr(_uuidof(UAReadArguments));
            ReadArguments4Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments4Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;i=10389";

            CComSafeArray<VARIANT> arguments(4);
            arguments.SetAt(0, _variant_t((IDispatch*)ReadArguments1Ptr));
            arguments.SetAt(1, _variant_t((IDispatch*)ReadArguments2Ptr));
            arguments.SetAt(2, _variant_t((IDispatch*)ReadArguments3Ptr));
            arguments.SetAt(3, _variant_t((IDispatch*)ReadArguments4Ptr));
        }
    }
}
```

```

    arguments.SetAt(3, _variant_t((IDispatch*)ReadArguments4Ptr));

    // Instantiate the client object
    _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

    // Obtain values. By default, the Value attributes of the nodes will be
    // read.
    LPSAFEARRAY pArguments = arguments.Detach();
    CComSafeArray<VARIANT> results;
    results.Attach(ClientPtr->ReadMultiple(&pArguments));
    arguments.Attach(pArguments);

    // Display results
    for (int i = results.GetLowerBound(); i <= results.GetUpperBound(); i++)
    {
        _UAAttributeDataResultPtr ResultPtr = results[i];

        _variant_t vString;
        vString.ChangeType(VT_BSTR, &_variant_t((IDispatch*)ResultPtr-
>AttributeData));
        _tprintf(_T("results(%d).AttributeData: %s\n"), i,
        CW2T((_bstr_t)vString));
    }
    // Release all interface pointers BEFORE calling CoUninitialize()
    CoUninitialize();
}
}

```

## PHP

---

```

// This example shows how to read the attributes of 4 OPC-UA nodes at once, and
// display the results.

$ReadArguments1 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments1->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments1->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853";

$ReadArguments2 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments2->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments2->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845";

$ReadArguments3 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments3->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments3->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10304";

$ReadArguments4 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments4->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments4->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10389";

$arguments[0] = $ReadArguments1;
$arguments[1] = $ReadArguments2;
$arguments[2] = $ReadArguments3;

```

```
$arguments[3] = $ReadArguments4;

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Perform the operation
$results = $Client->ReadMultiple($arguments);

// Display results
for ($i = 0; $i < count($results); $i++)
{
    printf("results[%d].AttributeData: %s\n", $i, $results[$i]->AttributeData);
}
```

## Visual Basic (VB 6.)

Rem This example shows how to read the attributes of 4 OPC-UA nodes at once, and display the results.

```
Private Sub ReadMultiple_Main_Command_Click()
    OutputText = ""

    Dim ReadArguments1 As New UAReadArguments
    ReadArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments1.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10853"

    Dim ReadArguments2 As New UAReadArguments
    ReadArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments2.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10845"

    Dim ReadArguments3 As New UAReadArguments
    ReadArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments3.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10304"

    Dim ReadArguments4 As New UAReadArguments
    ReadArguments4.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments4.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10389"

    Dim arguments(3) As Variant
    Set arguments(0) = ReadArguments1
    Set arguments(1) = ReadArguments2
    Set arguments(2) = ReadArguments3
    Set arguments(3) = ReadArguments4

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    ' Obtain values. By default, the Value attributes of the nodes will be read.
    Dim results() As Variant
    results = Client.ReadMultiple(arguments)

    ' Display results
    Dim i: For i = LBound(results) To UBound(results)
```

```
    Dim Result As UAAttributeDataResult: Set Result = results(i)
    OutputText = OutputText & "results(" & i & ").AttributeData: " &
Result.AttributeData & vbCrLf
    Next
End Sub
```

## VBScript

Rem This example shows how to read the attributes of 4 OPC-UA nodes at once, and display the results.

```
Option Explicit

Dim ReadArguments1: Set ReadArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"

Dim ReadArguments2: Set ReadArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"

Dim ReadArguments3: Set ReadArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10304"

Dim ReadArguments4: Set ReadArguments4 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments4.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments4.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10389"

Dim arguments(3)
Set arguments(0) = ReadArguments1
Set arguments(1) = ReadArguments2
Set arguments(2) = ReadArguments3
Set arguments(3) = ReadArguments4

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Perform the operation
Dim results: results = Client.ReadMultiple(arguments)

' Display results
Dim i: For i = LBound(results) To UBound(results)
    WScript.Echo "results(" & i & ").AttributeData: " & results(i).AttributeData
Next
```

## Read parameters

It is possible to specify the maximum age of the value to be read. In order to do that, set the `MaximumAge` property of `UAReadArguments.ReadParameters` object passed to the read method, to the appropriate number of milliseconds. By default, this property is set to `UAReadParameters.FromCache`, which denotes reading from the cache.

Note: OPC "Classic" has separate functions for reading so-called OPC properties. OPC properties contain additional information related to a node. In OPC Unified Architecture, properties are accessed in the same way as other information. That is, if you have a node ID (obtain e.g. by browsing) of a property, you can use one of the `ReadXXXX` methods described in this chapter to get a value of that property.

## 6.1.1.3.1 Reading just the value

Some applications need the actual data value for further processing (e.g. for computations that need be performed on the values), i.e. the status code must be Good and a valid value must be provided by the server, otherwise it is considered an error.

### A single node and attribute

For such usage, call the `ReadValue` method, passing it the same arguments as to the Read method. The method will read the attribute data, check if the status is Good, and you will receive back an Object holding the actual data value. If the status code is not Good, the method will throw a `UAStatusCodeException`.

#### C#

```
// This example shows how to read value of a single node, and display it.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class ReadValue
        {
            public static void Overload1()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Obtain value of a node
                object value = easyUAClient.ReadValue(
                    "http://opcua.demo-this.com:51211/UA/SampleServer",    // or
                    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                    "nsu=http://test.org/UA/Data/;i=10853");

                // Display results
                Console.WriteLine("value: {0}", value);
            }
        }
    }
}
```

#### VB.NET

```
' This example shows how to read value of a single node, and display it.
Imports OpcLabs.EasyOpc.UA

Namespace _EasyUAClient
```

```
Friend Class ReadValue
    Public Shared Sub Overload1()
        ' Instantiate the client object
        Dim easyUAClient = New EasyUAClient()

        ' Obtain value of a node
        Dim value As Object = easyUAClient.ReadValue("http://opcua.demo-
this.com:51211/UA/SampleServer", _
"nsu=http://test.org/UA/Data/;i=10853") ' or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"

        ' Display results
        Console.WriteLine("value: {0}", value)
    End Sub
End Class
End Namespace
```

## C++

---

```
// This example shows how to read value of a single node, and display it.
```

```
#include "stdafx.h"
#include "_EasyUAClient.ReadValue.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void ReadValue::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

            // Perform the operation
```

```
_variant_t value = ClientPtr->ReadValue(
    L"http://opcua.demo-this.com:51211/UA/SampleServer",
    L"nsu=http://test.org/UA/Data/;i=10853");

    // Display results
    _variant_t vString;
    vString.ChangeType(VT_BSTR, &value);
    _tprintf(_T("value: %s\n"), CW2T((_bstr_t)vString));
}

// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}
```

## JScript

```
// This example shows how to read value of a single node, and display it.
var Client = new ActiveXObject("OpcLabs.EasyOpc.UA.EasyUAClient");
var value = Client.ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853");
WScript.Echo("value: " + value);
```

## Free Pascal

```
// This example shows how to read value of a single node, and display it.

class procedure ReadValue.Main;
var
  Client: EasyUAClient;
  Value: OleVariant;
begin
  // Instantiate the client object
  Client := CoEasyUAClient.Create;

  Value := Client.ReadValue(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10853');
  WriteLn('value: ', Value);
end;
```

## Object Pascal

```
// This example shows how to read value of a single node, and display it.

class procedure ReadValue.Main;
var
  Client: TEasyUAClient;
  Value: OleVariant;
begin
  // Instantiate the client object
  Client := TEasyUAClient.Create(nil);

  Value := Client.ReadValue(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10853');
  WriteLn('value: ', Value);
end;
```

## PHP

```
// This example shows how to read value of a single node, and display it.
```

```
// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Perform the operation
$value = $Client->ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853");

// Display results
printf("Value: %s\n", $value);
```

## Python

```
# This example shows how to read value of a single node, and display it.

import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.UA.EasyUAClient')

# Perform the operation
value = client.ReadValue('http://opcua.demo-this.com:51211/UA/SampleServer',
'nsu=http://test.org/UA/Data/;i=10853')

# Display results
print('value: ', value)
```

## Visual Basic (VB 6.)

```
Rem This example shows how to read value of a single node, and display it.

Private Sub ReadValue_Main_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    Dim value As Variant
    value = Client.ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853")
    OutputText = OutputText & "value: " & value & vbCrLf
End Sub
```

## VBScript

```
Rem This example shows how to read value of a single node, and display it.

Option Explicit

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Perform the operation
Dim value: value = Client.ReadValue("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853")

' Display results
WScript.Echo "value: " & value
```

## Multiple nodes or attributes

For reading just the data values of multiple data values (with Good status) in an efficient manner, call the [ReadMultipleValues](#) method (instead of multiple [ReadValue](#) calls in a loop). You pass in an array of [UAReadArguments](#) objects, and you will receive back an array of [ValueResult](#) objects.

## C#

```
// This example shows how to read the Value attributes of 3 different nodes at once.  
Using the same method, it is also possible  
// to read multiple attributes of the same node.  
using OpcLabs.EasyOpc.UA;  
using System;  
  
namespace UADocExamples  
{  
    namespace _EasyUAClient  
    {  
        class ReadMultipleValues  
        {  
            public static void Main()  
            {  
                // Instantiate the client object  
                var easyUAClient = new EasyUAClient();  
  
                // Obtain values. By default, the Value attributes of the nodes will  
                // be read.  
                ValueResult[] valueResultArray =  
                easyUAClient.ReadMultipleValues(new[]  
                {  
                    new UAReadArguments("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845"),  
                    new UAReadArguments("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853"),  
                    new UAReadArguments("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855")  
                });  
  
                // Display results  
                foreach (ValueResult valueResult in valueResultArray)  
                    Console.WriteLine("Value: {0}", valueResult.Value);  
  
                // Example output:  
                //  
                //Value: 8  
                //Value: -8.06803E+21  
                //Value: Strawberry Pig Banana Snake Mango Purple Grape Monkey  
Purple? Blueberry Lemon^  
            }  
        }  
    }  
}
```

## VB.NET

```
' This example shows how to read the Value attributes of 3 different nodes at once.  
Using the same method, it is also possible  
' to read multiple attributes of the same node.  
Imports OpcLabs.BaseLib.OperationModel  
Imports OpcLabs.EasyOpc.UA  
Imports OpcLabs.EasyOpc.UA.OperationModel
```

```

Imports System

Namespace _EasyUAClient
    Friend Class ReadMultipleValues
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Obtain values. By default, the Value attributes of the nodes will be
            ' read.
            Dim valueResultArray() As ValueResult =
                easyUAClient.ReadMultipleValues(New UAReadArguments() _
                { _
                    New UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845"), _
                    New UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853"), _
                    New UAReadArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855") _
                } _
            )

            ' Display results
            For Each valueResult As ValueResult In valueResultArray
                Console.WriteLine("Value: {0}", valueResult.Value)
            Next valueResult

            ' Example output:
            '
            'Value: 8
            'Value: -8.06803E+21
            'Value: Strawberry Pig Banana Snake Mango Purple Grape Monkey Purple?
            Blueberry Lemon^
        End Sub
    End Class
End Namespace

```

## C++

---

```

// This example shows how to read the Value attributes of 3 different nodes at once.
// Using the same method, it is also possible
// to read multiple attributes of the same node.

#include "stdafx.h"
#include <atlsafe.h>
#include "_EasyUAClient.ReadMultipleValues.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

#import "System.Drawing.tlb"
using namespace System_Drawing;

#import "System.Windows.Forms.tlb"
using namespace System_Windows_Forms;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")

```

```

using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void ReadMultipleValues::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(_uuidof(EasyUAClient));

            _UAReadArgumentsPtr ReadArguments1Ptr(_uuidof(UAReadArguments));
            ReadArguments1Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments1Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10845";

            _UAReadArgumentsPtr ReadArguments2Ptr(_uuidof(UAReadArguments));
            ReadArguments2Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments2Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10853";

            _UAReadArgumentsPtr ReadArguments3Ptr(_uuidof(UAReadArguments));
            ReadArguments3Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            ReadArguments3Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10855";

            CComSafeArray<VARIANT> arguments(3);
            arguments.SetAt(0, _variant_t((IDispatch*)ReadArguments1Ptr));
            arguments.SetAt(1, _variant_t((IDispatch*)ReadArguments2Ptr));
            arguments.SetAt(2, _variant_t((IDispatch*)ReadArguments3Ptr));

            // Obtain values. By default, the Value attributes of the nodes will be
            read.
            LPSAFEARRAY pArguments = arguments.Detach();
            CComSafeArray<VARIANT> results;
            results.Attach(ClientPtr->ReadMultipleValues(&pArguments));
            arguments.Attach(pArguments);

            // Display results
            for (int i = results.GetLowerBound(); i <= results.GetUpperBound(); i++)
            {
                _ValueResultPtr ValueResultPtr = results[i];

```

```

        _variant_t vString;
        vString.ChangeType(VT_BSTR, &ValueResultPtr->Value);
        _tprintf(_T("Value: %s\n"), CW2T((_bstr_t)vString));
    }

    // Example output:
    //
    //Value: 8
    //Value: -8.06803E+21
    //Value: Strawberry Pig Banana Snake Mango Purple Grape Monkey Purple?
Blueberry Lemon^
}
// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}
}

```

## PHP

---

```

// This example shows how to read the Value attributes of 3 different nodes at once.
// Using the same method, it is also possible
// to read multiple attributes of the same node.

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

$ReadArguments1 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments1->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments1->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845";

$ReadArguments2 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments2->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments2->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853";

$ReadArguments3 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments");
$ReadArguments3->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$ReadArguments3->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855";

$arguments[0] = $ReadArguments1;
$arguments[1] = $ReadArguments2;
$arguments[2] = $ReadArguments3;

// Obtain values. By default, the Value attributes of the nodes will be read.
$results = $Client->ReadMultipleValues($arguments);

// Display results
for ($i = 0; $i < count($results); $i++)
{
    $ValueResult = $results[$i];
    printf("Value: %s\n", $ValueResult->Value);
}

// Example output:
//
//Value: 8

```

```
//Value: -8.06803E+21
//Value: Strawberry Pig Banana Snake Mango Purple Grape Monkey Purple? Blueberry
Lemon^
```

## Visual Basic (VB 6.)

Rem This example shows how to read the Value attributes of 3 different nodes at once. Using the same method, it is also possible  
Rem to read multiple attributes of the same node.

```
Private Sub ReadMultipleValues_Main_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    Dim ReadArguments1 As New UAReadArguments
    ReadArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments1.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10845"

    Dim ReadArguments2 As New UAReadArguments
    ReadArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments2.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10853"

    Dim ReadArguments3 As New UAReadArguments
    ReadArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    ReadArguments3.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10855"

    Dim arguments(2) As Variant
    Set arguments(0) = ReadArguments1
    Set arguments(1) = ReadArguments2
    Set arguments(2) = ReadArguments3

    ' Obtain values. By default, the Value attributes of the nodes will be read.
    Dim results() As Variant
    results = Client.ReadMultipleValues(arguments)

    ' Display results
    Dim i: For i = LBound(results) To UBound(results)
        Dim Result As ValueResult: Set Result = results(i)
        OutputText = OutputText & "Value: " & Result.value & vbCrLf
    Next

    ' Example output:
    '
    'Value: 8
    'Value: -8.06803E+21
    'Value: Strawberry Pig Banana Snake Mango Purple Grape Monkey Purple? Blueberry
Lemon^
End Sub
```

## VBScript

Rem This example shows how to read the Value attributes of 3 different nodes at

once. Using the same method, it is also possible Rem to read multiple attributes of the same node.

## Option Explicit

```
' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

Dim ReadArguments1: Set ReadArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"

Dim ReadArguments2: Set ReadArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"

Dim ReadArguments3: Set ReadArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAReadArguments")
ReadArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
ReadArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"

Dim arguments(2)
Set arguments(0) = ReadArguments1
Set arguments(1) = ReadArguments2
Set arguments(2) = ReadArguments3

' Obtain values. By default, the Value attributes of the nodes will be read.
Dim results: results = Client.ReadMultipleValues(arguments)

' Display results
Dim i: For i = LBound(results) To UBound(results)
    Dim ValueResult: Set ValueResult = results(i)
    WScript.Echo "Value: " & ValueResult.Value
Next

' Example output:
'
'Value: 8
'Value: -8.06803E+21
'Value: Strawberry Pig Banana Snake Mango Purple Grape Monkey Purple? Blueberry
Lemon^
```

## 6.1.2 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 Classic items, or attributes of OPC UA nodes). 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 the Browsing for Information chapter first.

## 6.1.2.1 Writing to OPC Classic Items

### A single item

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.

#### C#

```
// This example shows how to write a value into a single item.

using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        class WriteItemValue
        {
            public static void Main()
            {
                var easyDAClient = new EasyDAClient();

                easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 12345);
            }
        }
    }
}
```

#### VB.NET

```
' This example shows how to write a value into a single item.

Imports OpcLabs.EasyOpc.DataAccess

Namespace _EasyDAClient
    Friend Class WriteItemValue
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 12345)
        End Sub
    End Class
End Namespace
```

#### Object Pascal

```
// This example shows how to write a value into a single item.

class procedure WriteItemValue.Main;
var
    Client: TEasyDAClient;
begin
    // Instantiate the client object
    Client := TEasyDAClient.Create(nil);
```

```
Client.WriteItemValue("", "OPCLabs.KitServer.2", "Simulation.Register_I4", 12345);
end;
```

## VBScript

Rem This example shows how to write a value into a single item.

```
Option Explicit
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Client.WriteItemValue "", "OPCLabs.KitServer.2", "Simulation.Register_I4", 12345
```



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

## Multiple items

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 `DAItemValueArguments` objects, each specifying the location of OPC item, and the value to be written.

For an efficient writing into several items, use the `WriteMultipleItemValues` method.

## VB.NET

```
' This example shows how to write values into multiple items.

Imports OpcLabs.BaseLib.ObjectModel
Imports OpcLabs.EasyOpc.DataAccess
Imports OpcLabs.EasyOpc.DataAccess.ObjectModel

Namespace _EasyDAClient
    Friend Class WriteMultipleItemValues
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            Dim argumentsArray = New DAItemValueArguments() {
                New DAItemValueArguments("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 12345),
                New DAItemValueArguments("", "OPCLabs.KitServer.2",
"Simulation.Register_BOOL", True),
                New DAItemValueArguments("", "OPCLabs.KitServer.2",
"Simulation.Register_R4", 123.45) _
            }

            Dim resultsArray As OperationResult() =
easyDAClient.WriteMultipleItemValues(argumentsArray)

            For i = 0 To resultsArray.Length - 1
                Debug.Assert(resultsArray(i) IsNot Nothing)
                Console.WriteLine("resultsArray[{0}].Succeeded: {1}",
resultsArray(i).Succeeded)
            Next i
        End Sub
    End Class
End Namespace
```

## Object Pascal

```
// This example shows how to write values into 3 items at once.

class procedure WriteMultipleItemValues.Main;
var
  Arguments: OleVariant;
  Client: TEasyDAClient;
  ItemValueArguments1: _DAItemValueArguments;
  ItemValueArguments2: _DAItemValueArguments;
  ItemValueArguments3: _DAItemValueArguments;
  Results: OleVariant;
begin
  ItemValueArguments1 := CoDAItemValueArguments.Create;
  ItemValueArguments1.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ItemValueArguments1.ItemDescriptor.ItemID := 'Simulation.Register_I4';
  ItemValueArguments1.Value := 23456;

  ItemValueArguments2 := CoDAItemValueArguments.Create;
  ItemValueArguments2.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ItemValueArguments2.ItemDescriptor.ItemID := 'Simulation.Register_R8';
  ItemValueArguments2.Value := 2.34567890;

  ItemValueArguments3 := CoDAItemValueArguments.Create;
  ItemValueArguments3.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ItemValueArguments3.ItemDescriptor.ItemID := 'Simulation.Register_BSTR';
  ItemValueArguments3.Value := 'ABC';

  Arguments := VarArrayCreate([0, 2], varVariant);
  Arguments[0] := ItemValueArguments1;
  Arguments[1] := ItemValueArguments2;
  Arguments[2] := ItemValueArguments3;

  // Instantiate the client object
  Client := TEeasyDAClient.Create(nil);

  TVarData(Results).VType := varArray or varVariant;
  TVarData(Results).VArray := PVarArray(
    Client.WriteMultipleItemValues(PSafeArray(TVarData(Arguments).VArray)));
end;
```

## VBScript

Rem This example shows how to write values into 3 items at once.

Option Explicit

```
Dim ItemValueArguments1: Set ItemValueArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAItemValueArguments")
ItemValueArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemValueArguments1.ItemDescriptor.ItemID = "Simulation.Register_I4"
ItemValueArguments1.Value = 23456

Dim ItemValueArguments2: Set ItemValueArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAItemValueArguments")
ItemValueArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemValueArguments2.ItemDescriptor.ItemID = "Simulation.Register_R8"
ItemValueArguments2.Value = 2.34567890

Dim ItemValueArguments3: Set ItemValueArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAItemValueArguments")
ItemValueArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemValueArguments3.ItemDescriptor.ItemID = "Simulation.Register_BSTR"
```

```
ItemValueArguments3.Value = "ABC"

Dim arguments(2)
Set arguments(0) = ItemValueArguments1
Set arguments(1) = ItemValueArguments2
Set arguments(2) = ItemValueArguments3

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim results: results = Client.WriteMultipleItemValues(arguments)
```

## 6.1.2.1.1 Writing value, timestamp and quality

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.

### A single item

#### Object Pascal

```
// This example shows how to write a value, timestamp and quality into a single item.

class procedure WriteItem.Main;
var
  Client: TEasyDAClient;
begin
  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);

  Client.WriteItem('', 'OPCLabs.KitServer.2', 'Simulation.Register_I4', 12345,
EncodeDate(1980, 1, 1), DAQualities_GoodNonSpecific);
end;
```

#### VBScript

```
Rem This example shows how to write a value, timestamp and quality into a single item.

Option Explicit

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Client.WriteItem "", "OPCLabs.KitServer.2", "Simulation.Register_I4", 12345,
DateSerial(1980, 1, 1), 192
```

### Multiple items

#### Object Pascal

```
// This example shows how to write values, timestamps and qualities into 3 items at once.

class procedure WriteMultipleItems.Main;
var
  Arguments: OleVariant;
```

```

Client: TEasyDAClient;
ItemVtqArguments1: _DAItemVtqArguments;
ItemVtqArguments2: _DAItemVtqArguments;
ItemVtqArguments3: _DAItemVtqArguments;
Results: OleVariant;
begin
  ItemVtqArguments1 := CoDAItemVtqArguments.Create;
  ItemVtqArguments1.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ItemVtqArguments1.ItemDescriptor.ItemID := 'Simulation.Register_I4';
  ItemVtqArguments1.Vtq.Value := 23456;
  ItemVtqArguments1.Vtq.TimestampLocal := Now();
  ItemVtqArguments1.Vtq.Quality.NumericalValue := DAQualities_GoodNonSpecific;

  ItemVtqArguments2 := CoDAItemVtqArguments.Create;
  ItemVtqArguments2.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ItemVtqArguments2.ItemDescriptor.ItemID := 'Simulation.Register_R8';
  ItemVtqArguments2.Vtq.Value := 2.34567890;
  ItemVtqArguments2.Vtq.TimestampLocal := Now();
  ItemVtqArguments2.Vtq.Quality.NumericalValue := DAQualities_GoodNonSpecific;

  ItemVtqArguments3 := CoDAItemVtqArguments.Create;
  ItemVtqArguments3.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
  ItemVtqArguments3.ItemDescriptor.ItemID := 'Simulation.Register_BSTR';
  ItemVtqArguments3.Vtq.Value := 'ABC';
  ItemVtqArguments3.Vtq.TimestampLocal := Now();
  ItemVtqArguments3.Vtq.Quality.NumericalValue := DAQualities_GoodNonSpecific;

  Arguments := VarArrayCreate([0, 2], varVariant);
  Arguments[0] := ItemVtqArguments1;
  Arguments[1] := ItemVtqArguments2;
  Arguments[2] := ItemVtqArguments3;

  // Instantiate the client object
  Client := TEasyDAClient.Create(nil);

  TVarData(Results).VType := varArray or varVariant;
  TVarData(Results).VArray := PVarArray(
    Client.WriteMultipleItems(PSafeArray(TVarData(Arguments).VArray)));
end;

```

## VBScript

---

Rem This example shows how to write values, timestamps and qualities into 3 items at once.

### Option Explicit

```

Dim ItemVtqArguments1: Set ItemVtqArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAItemVtqArguments")
ItemVtqArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemVtqArguments1.ItemDescriptor.ItemID = "Simulation.Register_I4"
ItemVtqArguments1.Vtq.Value = 23456
ItemVtqArguments1.Vtq.TimestampLocal = Now()
ItemVtqArguments1.Vtq.Quality.NumericalValue = 192

Dim ItemVtqArguments2: Set ItemVtqArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.ObjectModel.DAItemVtqArguments")
ItemVtqArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemVtqArguments2.ItemDescriptor.ItemID = "Simulation.Register_R8"
ItemVtqArguments2.Vtq.Value = 2.34567890
ItemVtqArguments2.Vtq.TimestampLocal = Now()

```

```
ItemVtqArguments2.Vtq.Quality.NumericalValue = 192

Dim ItemVtqArguments3: Set ItemVtqArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAItemVtqArguments")
ItemVtqArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemVtqArguments3.ItemDescriptor.ItemID = "Simulation.Register_BSTR"
ItemVtqArguments3.Vtq.Value = "ABC"
ItemVtqArguments3.Vtq.TimestampLocal = Now()
ItemVtqArguments3.Vtq.Quality.NumericalValue = 192

Dim arguments(2)
Set arguments(0) = ItemVtqArguments1
Set arguments(1) = ItemVtqArguments2
Set arguments(2) = ItemVtqArguments3

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim results: results = Client.WriteMultipleItems(arguments)
```

## 6.1.2.2 Writing Attributes of OPC UA Nodes

### A single node and attribute

If you want to write a data value into an attribute of a specific node in OPC UA, call the `WriteValue` method, passing it the data value you want to write, arguments for endpoint descriptor, node ID, and an optional data type.

#### C#

```
// This example shows how to write a value into a single node.
using OpcLabs.EasyOpc.UA;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class WriteValue
        {
            public static void Main()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Modify value of a node
                easyUAClient.writeValue(
                    "http://opcua.demo-this.com:51211/UA/SampleServer",      // or
                    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                        "nsu=http://test.org/UA/Data/;i=10221",
                    12345);
            }
        }
    }
}
```

#### VB.NET

```
' This example shows how to write a value into a single node.
Imports OpcLabs.EasyOpc.UA
```

```
Namespace _EasyUAClient
    Friend Class WriteValue
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Modify value of a node
            easyUAClient.writeValue("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10221", 12345) ' or
            "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
        End Sub
    End Class
End Namespace
```

## C++

---

```
// This example shows how to write a value into a single node.

#include "stdafx.h"
#include "_EasyUAClient.WriteValue.h"

// OpcLabs.BaseLib
#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
namespace _EasyUAClient
{
    void WriteValue::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

            // Perform the operation
            ClientPtr->WriteValue(
                L"http://opcua.demo-this.com:51211/UA/SampleServer", // or
                "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                L"nsu=http://test.org/UA/Data/;i=10221",

```

```
    12345);
}
// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}
}
```

## PHP

```
// This example shows how to write a value into a single node.

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Perform the operation
$Client->WriteValue(
    "http://opcua.demo-this.com:51211/UA/SampleServer", // or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"
    "nsu=http://test.org/UA/Data/;i=10221",
    12345);
```

## Python

```
# This example shows how to write a value into a single node.

import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.UA.EasyUAClient')

# Perform the operation
client.WriteValue(
    'http://opcua.demo-this.com:51211/UA/SampleServer', # or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"
    'nsu=http://test.org/UA/Data/;i=10221',
    12345)
```

## Visual Basic (VB 6.)

```
Rem This example shows how to write a value into a single node.
```

```
Private Sub WriteValue_Main_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    ' Perform the operation
    Call Client.writeValue("http://opcua.demo-this.com:51211/UA/SampleServer",
    "nsu=http://test.org/UA/Data/;i=10221", 12345)
End Sub
```

## VBScript

```
Rem This example shows how to write a value into a single node.
```

```
Option Explicit

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
```

```
' Perform the operation
Client.writeValue "http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10221", 12345 ' or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"
```

When you specify individual arguments to the simplest overload of the [WriteValue](#) method, the method will write to the **Value** attribute of the node, and if the value type is an array, it will modify the whole contents of that array. If you want to write into a different attribute, or write only to a subset of an array, use one of the more complex overloads of the [WriteValue](#) method, such as with an argument of [UAWriteValueArguments](#) type, and fill this argument with all necessary information.

## Multiple nodes or attributes

For writing data values into multiple OPC attributes (in the same or different nodes) in an efficient manner, call the [WriteMultipleValues](#) method. You pass in an array of [UAWriteValueArgument](#) objects, each specifying the location of OPC node, attribute, and the value to be written.

### C#

```
// This example shows how to write values into 3 nodes at once.
using OpcLabs.EasyOpc.UA;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class WriteMultipleValues
        {
            public static void Main()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Modify value of a node
                easyUAClient.WriteMultipleValues(new []
                {
                    new UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10221", 23456),
                    new UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10226", 2.34567890),
                    new UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10227", "ABC")
                });
            }
        }
    }
}
```

### VB.NET

```
' This example shows how to write values into 3 nodes at once.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel
```

```

Namespace _EasyUAClient
    Partial Friend Class WriteMultipleValues
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Modify value of a node
            easyUAClient.WriteMultipleValues(New UAWriteValueArguments() _
                {
                    New UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10221", 23456), _
                    New UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10226", 2.3456789),
                    New UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10227", "ABC")
                }
            )
        End Sub
    End Class
End Namespace

```

The example above can be extended by properly testing for success of each operation, like this:

## C#

```

// This example shows how to write values into 3 nodes at once, test for success of
// each write and display the exception
// message in case of failure.
using System;
using OpcLabs.EasyOpc.UA;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class WriteMultipleValues
        {
            public static void TestSuccess()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Modify value of a node
                OperationResult[] operationResultArray =
easyUAClient.WriteMultipleValues(new[]
                {
                    new UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10221", 23456),
                    new UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10226", "This string
cannot be converted to Double"),
                    new UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;s=UnknownNode", "ABC")
                });

                for (int i = 0; i < operationResultArray.Length; i++)
                    if (operationResultArray[i].Succeeded)

```

```
        Console.WriteLine("Result {0}: success", i);
    else
        Console.WriteLine("Result {0}: {1}", i,
operationResultArray[i].Exception.GetBaseException().Message);
    }
}
}
```

## VB.NET

```
' This example shows how to write values into 3 nodes at once, test for success of
each write and display the exception
' message in case of failure.
Imports OpcLabs.BaseLib.OperationModel
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel
Imports System

Namespace _EasyUAClient
    Partial Friend Class WriteMultipleValues
        Public Shared Sub TestSuccess()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Modify value of a node
            Dim operationResultArray() As OperationResult =
easyUAClient.WriteMultipleValues(New UAWriteValueArguments() _
{ _
    New UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10221", 23456), _
    New UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10226", "This
string cannot be converted to Double"), _
    New UAWriteValueArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;s=UnknownNode",
"ABC") _
} _
)

        For i As Integer = 0 To operationResultArray.Length - 1
            If operationResultArray(i).Succeeded Then
                Console.WriteLine("Result {0}: success", i)
            Else
                Console.WriteLine("Result {0}: {1}", i,
operationResultArray(i).Exception.GetBaseException().Message)
            End If
        Next i
    End Sub
End Class
End Namespace
```

## C++

```
// This example shows how to write values into 3 nodes at once, test for success of
each write and display the exception
// message in case of failure.

#include "stdafx.h"
#include <atlsafe.h>
#include "_EasyUAClient.WriteMultipleValues.h"
```

```

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void WriteMultipleValues::TestSuccess()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(_uuidof(EasyUAClient));

            _UAWriteValueArgumentsPtr
            WriteValueArguments1Ptr(_uuidof(UAWriteValueArguments));
            WriteValueArguments1Ptr->EndpointDescriptor->UrlString =
L"http://opcua.demo-this.com:51211/UA/SampleServer";
            WriteValueArguments1Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;i=10221";
            WriteValueArguments1Ptr->Value = 23456;

            _UAWriteValueArgumentsPtr
            WriteValueArguments2Ptr(_uuidof(UAWriteValueArguments));
            WriteValueArguments2Ptr->EndpointDescriptor->UrlString =
L"http://opcua.demo-this.com:51211/UA/SampleServer";
            WriteValueArguments2Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;i=10226";
            WriteValueArguments2Ptr->Value = L"This string cannot be converted to
Double";

            _UAWriteValueArgumentsPtr
            WriteValueArguments3Ptr(_uuidof(UAWriteValueArguments));
            WriteValueArguments3Ptr->EndpointDescriptor->UrlString =
L"http://opcua.demo-this.com:51211/UA/SampleServer";
            WriteValueArguments3Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data/;s=UnknownNode";
            WriteValueArguments3Ptr->Value = L"ABC";

            CComSafeArray<VARIANT> arguments(3);
    }
}

```

```

        arguments.SetAt(0, _variant_t((IDispatch*)WriteValueArguments1Ptr));
        arguments.SetAt(1, _variant_t((IDispatch*)WriteValueArguments2Ptr));
        arguments.SetAt(2, _variant_t((IDispatch*)WriteValueArguments3Ptr));

        // Obtain values. By default, the Value attributes of the nodes will be
        Write.

        LPSAFEARRAY pArguments = arguments.Detach();
        CComSafeArray<VARIANT> results;
        results.Attach(ClientPtr->WriteMultipleValues(&pArguments));
        arguments.Attach(pArguments);

        // Display results
        for (int i = results.GetLowerBound(); i <= results.GetUpperBound(); i++)
        {
            _UAWriteResultPtr ResultPtr = results[i];

            if (ResultPtr->Succeeded)
                _tprintf(_T("Result %d success\n"), i);
            else
                _tprintf(_T("Result %d: %s\n"), i, CW2T(ResultPtr->Exception-
>GetBaseException()->Message));
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}

```

## PHP

---

```

// This example shows how to write values into 3 nodes at once, test for success of
each write and display the exception
// message in case of failure.

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

$WriteValueArguments1 = new
COM("OpcLabs.EasyOpc.UA.OperationModel.UAWriteValueArguments");
$WriteValueArguments1->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$WriteValueArguments1->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10221";
$WriteValueArguments1->Value = 23456;

$WriteValueArguments2 = new
COM("OpcLabs.EasyOpc.UA.OperationModel.UAWriteValueArguments");
$WriteValueArguments2->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$WriteValueArguments2->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10226";
$WriteValueArguments2->Value = "This string cannot be converted to Double";

$WriteValueArguments3 = new
COM("OpcLabs.EasyOpc.UA.OperationModel.UAWriteValueArguments");
$WriteValueArguments3->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$WriteValueArguments3->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;s=UnknownNode";
$WriteValueArguments3->Value = "ABC";

```

```
$arguments[0] = $WriteValueArguments1;
$arguments[1] = $WriteValueArguments2;
$arguments[2] = $WriteValueArguments3;

// Modify values of nodes
$results = $Client->WriteMultipleValues($arguments);

// Display results
for ($i = 0; $i < count($results); $i++)
{
    $WriteResult = $results[$i];
    // The UA Test Server does not support this, and therefore a failure will occur.
    if ($WriteResult->Succeeded)
        printf("Result %d success\n", $i);
    else
        printf("Result %d: %s \n", $i, $WriteResult->Exception->GetBaseException()->Message);
}
```

## Visual Basic (VB 6.)

---

```
Rem This example shows how to write values into 3 nodes at once, test for success of
each write and display the exception
Rem message in case of failure.
```

```
Private Sub WriteMultipleValues_TestSuccess_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    Dim WriteValueArguments1 As New UAWriteValueArguments
    WriteValueArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    WriteValueArguments1.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10221"
    WriteValueArguments1.SetValue 23456

    Dim WriteValueArguments2 As New UAWriteValueArguments
    WriteValueArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    WriteValueArguments2.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10226"
    WriteValueArguments2.SetValue "This string cannot be converted to Double"

    Dim WriteValueArguments3 As New UAWriteValueArguments
    WriteValueArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    WriteValueArguments3.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;s=UnknownNode"
    WriteValueArguments3.SetValue "ABC"

    Dim arguments(2) As Variant
    Set arguments(0) = WriteValueArguments1
    Set arguments(1) = WriteValueArguments2
    Set arguments(2) = WriteValueArguments3

    ' Modify values of nodes
    Dim results As Variant
    results = Client.WriteMultipleValues(arguments)
```

```
' Display results
Dim i: For i = LBound(results) To UBound(results)
    Dim Result As UAWriteResult: Set Result = results(i)
    If Result.Succeeded Then
        OutputText = OutputText & "Result " & i & " success" & vbCrLf
    Else
        OutputText = OutputText & "Result " & i & ": " &
Result.Exception.GetBaseException().Message & vbCrLf
    End If
Next
End Sub
```

## VBScript

```
Rem This example shows how to write values into 3 nodes at once, test for success of
each write and display the exception
Rem message in case of failure.
```

```
Option Explicit
```

```
' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

Dim WriteValueArguments1: Set WriteValueArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAWriteValueArguments")
WriteValueArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
WriteValueArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10221"
WriteValueArguments1.Value = 23456

Dim WriteValueArguments2: Set WriteValueArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAWriteValueArguments")
WriteValueArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
WriteValueArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10226"
WriteValueArguments2.Value = "This string cannot be converted to Double"

Dim WriteValueArguments3: Set WriteValueArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAWriteValueArguments")
WriteValueArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
WriteValueArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;s=UnknownNode"
WriteValueArguments3.Value = "ABC"

Dim arguments(2)
Set arguments(0) = WriteValueArguments1
Set arguments(1) = WriteValueArguments2
Set arguments(2) = WriteValueArguments3

' Modify values of nodes
Dim results: results = Client.WriteMultipleValues(arguments)

' Display results
Dim i: For i = LBound(results) To UBound(results)
    Dim WriteResult: Set WriteResult = results(i)
    If WriteResult.Succeeded Then
        WScript.Echo "Result " & i & " success"
    Else
```

```
WScript.Echo "Result " & i & ":" &
WriteResult.Exception.GetBaseException().Message
End If
Next
```

## Write results

The [WriteMultiple](#) and [WriteMultipleValues](#) methods return an array of [UAWriteResult](#) objects. Besides the inherited members of [OperationResult](#), such as the [Exception](#), the [UAWriteResult](#) contains additional information that describes the outcome of the Write operation in case of success:

- The [Clamped](#) flag: Determines whether the value written was accepted but was clamped.
- The [CompletesAsynchronously](#) flag: Determines whether the processing will complete asynchronously.

### 6.1.2.2.1 Data Type in OPC UA Write

The value written needs to be of correct type, and the data type needs to be passed to the OPC-UA server together with the value.

QuickOPC-UA processes the value and data type in following steps:

1. Determine the (CLS-compliant) .NET data type. If you are writing to an attribute other than the **Value** attribute, the component uses the data type of that attribute as given by the OPC-UA specification.

Wherever the **Value** attribute is being written (which is the most common case), and your code has not provided a specific value type, the data type is resolved from the OPC-UA server by reading the **DataType** and **ValueRank** attributes of the same node. This is because the type of **Value** attribute can differ with each node.

When the data type of the node is not among the standard data types defined in OPC UA, `ProductName%%` will send the value and corresponding type ID to the OPC-UA server as a signed integer. If the server expects a value that must be an unsigned integer, even though the value is otherwise correct (and also non-negative), an error can occur.

How do you specify the value type in your code? This depends on the type of method you are calling. Some methods accept individual arguments, and they have overloads with either [valueType](#) or [valueTypeCode](#) argument. Other methods accept [UAWriteArguments](#) or [UAWriteValueArguments](#) object; in this case, you specify the value type by properly setting the [ValueType](#) or [ValueTypeCode](#) property in this object.

The [TypeCode](#) can be used for basic, simple types. For more complex types, such as arrays, you need to specify the .NET [Type](#).

When the value type is a null reference, or the value type code is [TypeCode.Empty](#), it means that the value type is not specified, and the component will determine it by interrogating the OPC-UA server, as described above.

### 6.1.2.2.2 Writing value, timestamps and status code

Some OPC servers allow a combination of value, timestamps, and status code be written into their nodes (or to some nodes only). If you need to do this, call [Write](#) or [WriteMultiple](#) method.

#### A single node and attribute

##### C#

```
// This example shows how to write data (a value, timestamps and status code) into a
// single attribute of a node.
using System;
```

```

using OpcLabs.EasyOpc.UA;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class Write
        {
            public static void Main()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                try
                {
                    // Modify data of a node's attribute
                    easyUAClient.Write(
                        "http://opcua.demo-this.com:51211/UA/SampleServer",      // or
                        "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                        "nsu=http://test.org/UA/Data/;i=10221",
                        new UAAttributeData(12345, UASeverity.GoodOrSuccess,
                        DateTime.UtcNow));
                    // Writing server timestamp is not supported by the sample server.

                    // The UA Test Server does not support this, and therefore a failure
                    will occur.
                }
                catch (UAException uaException)
                {
                    Console.WriteLine("Failure: {0}",
                        uaException.GetBaseException().Message);
                }
            }
        }
    }
}

```

## VB.NET

---

```

' This example shows how to write data (a value, timestamps and status code) into a
single attribute of a node.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class Write
        Public Shared Sub Main()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            Try
                ' Modify data of a node's attribute
                easyUAClient.Write("http://opcua.demo-this.com:51211/UA/SampleServer",
                    "nsu=http://test.org/UA/Data/;i=10221",
                    New UAAttributeData(12345, UASeverity.GoodOrSuccess,
                    Date.UtcNow)) ' or "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                ' Writing server timestamp is not supported by the sample server.

                ' The UA Test Server does not support this, and therefore a failure will
                occur.
            Catch uaException As UAException
                Console.WriteLine("Failure: {0}", uaException.GetBaseException().Message)
            End Try
        End Sub
    End Class

```

End Namespace

## PHP

```
// This example shows how to write data (a value, timestamps and status code) into a
single attribute of a node.

$GoodOrSuccess = 0;

// Instantiate the client object
$client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Modify data of a node's attribute
$statusCode = new COM("OpcLabs.EasyOpc.UA.UAStatusCode");
$statusCode->Severity = $GoodOrSuccess;
$attributeData = new COM("OpcLabs.EasyOpc.UA.UAAttributeData");
$attributeData->Value = 12345;
$attributeData->StatusCode = $statusCode;
$attributeData->SourceTimestamp = (time() - 25569)/86400.0;
// Writing server timestamp is not supported by the sample server.

// Perform the operation
try
{
    $client->Write(
        "http://opcua.demo-this.com:51211/UA/SampleServer", // or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"
        "nsu=http://test.org/UA/Data/;i=10221",
        $attributeData);
    // The UA Test Server does not support this, and therefore a failure will occur.
}
catch (com_exception $e)
{
    printf("Failure: %s\n", $e->getMessage());
}
```

## VBScript

Rem This example shows how to write data (a value, timestamps and status code) into a single attribute of a node.

```
Option Explicit

Const GoodOrSuccess = 0

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Modify data of a node's attribute
Dim StatusCode: Set StatusCode = CreateObject("OpcLabs.EasyOpc.UA.UAStatusCode")
StatusCode.Severity = GoodOrSuccess
Dim AttributeData: Set AttributeData = CreateObject("OpcLabs.EasyOpc.UA.UAAttributeData")
AttributeData.Value = 12345
AttributeData.StatusCode = StatusCode
AttributeData.SourceTimestamp = Now
' Writing server timestamp is not supported by the sample server.

' Perform the operation
On Error Resume Next
Client.Write "http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10221",
    AttributeData ' or "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
' The UA Test Server does not support this, and therefore a failure will occur.
If Err <> 0 Then
    WScript.Echo "Failure: " & Err.Description
```

```
End If  
On Error Goto 0
```

## Multiple nodes or attributes

It is more efficient to use the [WriteMultiple](#) method, instead of several calls to the [Write](#) method.

### C#

```
// This example shows how to write data (a value, timestamps and status code) into 3  
nodes at once, test for success of each  
// write and display the exception message in case of failure.  
using System;  
using OpcLabs.EasyOpc.UA;  
  
namespace UADocExamples  
{  
    namespace _EasyUAClient  
    {  
        class WriteMultiple  
        {  
            public static void TestSuccess()  
            {  
                // Instantiate the client object  
                var easyUAClient = new EasyUAClient();  
  
                // Modify data of nodes' attributes  
                OperationResult[] operationResultArray = easyUAClient.WriteMultiple(new[]  
                {  
                    new UAWriteArguments("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10221",  
                        new UAAttributeData(23456, UASeverity.GoodOrSuccess,  
DateTime.UtcNow),  
                        new UAWriteArguments("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10226",  
                            new UAAttributeData(2.34567890, UASeverity.GoodOrSuccess,  
DateTime.UtcNow),  
                            new UAWriteArguments("http://opcua.demo-  
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10227",  
                                new UAAttributeData("ABC", UASeverity.GoodOrSuccess,  
DateTime.UtcNow))  
                    // Writing server timestamp is not supported by the sample  
server.  
                });  
  
                // The UA Test Server does not support this, and therefore failures will  
occur.  
  
                for (int i = 0; i < operationResultArray.Length; i++)  
                    if (operationResultArray[i].Succeeded)  
                        Console.WriteLine("Result {0}: success", i);  
                    else  
                        Console.WriteLine("Result {0}: {1}", i,  
operationResultArray[i].Exception.GetBaseException().Message);  
            }  
        }  
    }  
}
```

### VB.NET

```
' This example shows how to write data (a value, timestamps and status code) into 3 nodes  
at once, test for success of each
```

```
' write and display the exception message in case of failure.
Imports System
Imports OpcLabs.BaseLib.OperationModel
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class WriteMultiple
        Public Shared Sub TestSuccess()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            ' Modify data of nodes' attributes
            Dim operationResultArray() As OperationResult =
easyUAClient.WriteMultiple(New UAWriteArguments() _
{
    New UAWriteArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10221",
                           New UAAttributeData(23456,
UASeverity.GoodOrSuccess, Date.UtcNow)),
    New UAWriteArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10226",
                           New UAAttributeData(2.3456789,
UASeverity.GoodOrSuccess, Date.UtcNow)),
    New UAWriteArguments("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10227",
                           New UAAttributeData("ABC",
UASeverity.GoodOrSuccess, Date.UtcNow))
}
)
        ' Writing server timestamp is not supported by the sample server.

        ' The UA Test Server does not support this, and therefore failures will
occur.

        For i As Integer = 0 To operationResultArray.Length - 1
            If operationResultArray(i).Succeeded Then
                Console.WriteLine("Result {0}: success", i)
            Else
                Console.WriteLine("Result {0}: {1}", i,
operationResultArray(i).Exception.GetBaseException().Message)
            End If
        Next i
    End Sub
End Class
End Namespace
```

## PHP

---

```
// This example shows how to write data (a value, timestamps and status code) into 3
nodes at once, test for success of each
// write and display the exception message in case of failure.

$GoodOrSuccess = 0;

// Instantiate the client object
$client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

$statusCode = new COM("OpcLabs.EasyOpc.UA.UAStatusCode");
$statusCode->Severity = $GoodOrSuccess;

$writeArguments1 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAWriteArguments");
$writeArguments1->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
```

```
$WriteArguments1->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10221";
$AttributeData1 = new COM("OpcLabs.EasyOpc.UA.UAAttributeData");
$AttributeData1->Value = 23456;
$AttributeData1->StatusCode = $StatusCode;
$AttributeData1->SourceTimestamp = (time() - 25569)/86400.0;
// Writing server timestamp is not supported by the sample server.
$WriteArguments1->AttributeData = $AttributeData1;

$WriteArguments2 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAWriteArguments");
$WriteArguments2->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$WriteArguments2->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10226";
$AttributeData2 = new COM("OpcLabs.EasyOpc.UA.UAAttributeData");
$AttributeData2->Value = 2.3456789;
$AttributeData2->StatusCode = $StatusCode;
$AttributeData2->SourceTimestamp = (time() - 25569)/86400.0;
// Writing server timestamp is not supported by the sample server.
$WriteArguments2->AttributeData = $AttributeData2;

$WriteArguments3 = new COM("OpcLabs.EasyOpc.UA.OperationModel.UAWriteArguments");
$WriteArguments3->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$WriteArguments3->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10227";
$AttributeData3 = new COM("OpcLabs.EasyOpc.UA.UAAttributeData");
$AttributeData3->Value = "ABC";
$AttributeData3->StatusCode = $StatusCode;
$AttributeData3->SourceTimestamp = (time() - 25569)/86400.0;
// Writing server timestamp is not supported by the sample server.
$WriteArguments3->AttributeData = $AttributeData3;

$arguments[0] = $WriteArguments1;
$arguments[1] = $WriteArguments2;
$arguments[2] = $WriteArguments3;

// Modify data of nodes' attributes
$results = $Client->WriteMultiple($arguments);

// Display results
for ($i = 0; $i < count($results); $i++)
{
    $WriteResult = $results[$i];
    // The UA Test Server does not support this, and therefore a failure will occur.
    if ($WriteResult->Succeeded)
        printf("Result %d success\n", $i);
    else
        printf("Result %d: %s \n", $i, $WriteResult->Exception->GetBaseException()-
>Message);
}
```

## VBScript

---

Rem This example shows how to write data (a value, timestamps and status code) into 3 nodes at once, test for success of each  
 Rem write and display the exception message in case of failure.

```
Option Explicit

Const GoodOrSuccess = 0

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
```

```
Dim StatusCode: Set StatusCode = CreateObject("OpcLabs.EasyOpc.UA.UAStatusCode")
StatusCode.Severity = GoodOrSuccess

Dim WriteArguments1: Set WriteArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAWriteArguments")
WriteArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
WriteArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10221"
Dim AttributeData1: Set AttributeData1 =
CreateObject("OpcLabs.EasyOpc.UA.UAAttributeData")
AttributeData1.Value = 23456
AttributeData1.StatusCode = StatusCode
AttributeData1.SourceTimestamp = Now
' Writing server timestamp is not supported by the sample server.
WriteArguments1.AttributeData = AttributeData1

Dim WriteArguments2: Set WriteArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAWriteArguments")
WriteArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
WriteArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10226"
Dim AttributeData2: Set AttributeData2 =
CreateObject("OpcLabs.EasyOpc.UA.UAAttributeData")
AttributeData2.Value = 2.3456789
AttributeData2.StatusCode = StatusCode
AttributeData2.SourceTimestamp = Now
' Writing server timestamp is not supported by the sample server.
WriteArguments2.AttributeData = AttributeData2

Dim WriteArguments3: Set WriteArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UAWriteArguments")
WriteArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
WriteArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10227"
Dim AttributeData3: Set AttributeData3 =
CreateObject("OpcLabs.EasyOpc.UA.UAAttributeData")
AttributeData3.Value = "ABC"
AttributeData3.StatusCode = StatusCode
AttributeData3.SourceTimestamp = Now
' Writing server timestamp is not supported by the sample server.
WriteArguments3.AttributeData = AttributeData3

Dim arguments(2)
Set arguments(0) = WriteArguments1
Set arguments(1) = WriteArguments2
Set arguments(2) = WriteArguments3

' Modify data of nodes' attributes
Dim results: results = Client.WriteMultiple(arguments)

' Display results
Dim i: For i = LBound(results) To UBound(results)
    Dim WriteResult: Set WriteResult = results(i)
    ' The UA Test Server does not support this, and therefore a failure will occur.
    If WriteResult.Succeeded Then
        WScript.Echo "Result " & i & " success"
    Else
        WScript.Echo "Result " & i & ": " &
        WriteResult.Exception.GetBaseException().Message
    End If
Next
```

## 6.1.3 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 text).

The methods we are describing in this chapter are for programmatic browsing, with no user interface (or when you provide the user interface by your own code).

### 6.1.3.1 Browsing for OPC Classic 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 a [String](#) or [ServerDescriptor](#), and pass the resulting string as a serverClass or a serverUrl argument either directly to the method call, or to a constructor of [ServerDescriptor](#) object.



In QuickOPC-COM, if you want to connect to some 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. For an OPC XML server, it contains its URL.

#### JScript

```
// This example shows how to obtain all ProgIDs of all OPC Data Access servers on  
// the local machine.  
  
var Client = new ActiveXObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient");  
var ServerElements = Client.BrowseServers("")  
  
for (var objEnum = new Enumerator(ServerElements) ; !objEnum.atEnd() ;  
objEnum.moveNext()) {  
    var ServerElement = objEnum.item();  
    WScript.Echo("ServerElements(\"" + ServerElement.UrlString + "\").ProgId: " +  
ServerElement.ProgId);  
}
```

#### VBScript

```
Rem This example shows how to obtain all ProgIDs of all OPC Data Access servers on  
Rem the local machine.  
  
Option Explicit  
  
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")  
Dim ServerElements: Set ServerElements = Client.BrowseServers("")  
  
Dim ServerElement: For Each ServerElement In ServerElements
```

```
WScript.Echo "ServerElements("") & ServerElement.ClsidString & "") .ProgId: " &
ServerElement.ProgId
Next
```

## 6.1.3.2 Browsing for OPC Classic 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. You will receive back a [DANodeElementCollection](#) object. Each [DANodeElement](#) contains information gathered about one sub-branch node, such as its name, or indication whether it has children.

### VBScript

```
Rem This example shows how to obtain all branches at the root of the address space.
For each branch, it displays whether
Rem it may have child nodes.
```

```
Option Explicit
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim BranchElements: Set BranchElements = Client.BrowseBranches("", 
"OPCLabs.KitServer.2", "")

Dim BranchElement: For Each BranchElement In BranchElements
    WScript.Echo "BranchElements("") & BranchElement.Name & "") .HasChildren: " &
    BranchElement.HasChildren
Next
```

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, 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](#).

### VBScript

```
Rem This example shows how to obtain all leaves under the "Simulation" branch of the
address space. For each leaf, it displays
Rem the ItemID of the node.
```

```
Option Explicit
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim LeafElements: Set LeafElements = Client.BrowseLeaves("", "OPCLabs.KitServer.2",
"Simulation")
```

```
Dim LeafElement: For Each LeafElement In LeafElements
    WScript.Echo "LeafElements(\"\" & LeafElement.Name & \"\") .ItemId: " &
LeafElement.ItemId
Next
```

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 [DABrowseParameters](#), or individual arguments for data type filter and access rights filter.

## C#

---

```
// This example shows how to recursively browse the nodes in the OPC address space.
using System;
using System.Diagnostics;
using OpcLabs.EasyOpc;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.AddressSpace;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        partial class BrowseNodes
        {
            public static void Recursive()
            {
                var stopwatch = new Stopwatch();
                stopwatch.Start();

                var easyDAClient = new EasyDAClient();
                _branchCount = 0;
                _leafCount = 0;
                BrowseFromNode(eeasyDAClient, "OPCLabs.KitServer.2", "");

                stopwatch.Stop();
                Console.WriteLine("Browsing has taken (milliseconds): {0}",
stopwatch.ElapsedMilliseconds);
                Console.WriteLine("Branch count: {0}", _branchCount);
                Console.WriteLine("Leaf count: {0}", _leafCount);
            }

            private static void BrowseFromNode(
                EasyDAClient client,
                ServerDescriptor serverDescriptor,
                DANodeDescriptor parentNodeDescriptor)
            {
                Debug.Assert(client != null);
                Debug.Assert(serverDescriptor != null);
                Debug.Assert(parentNodeDescriptor != null);

                // Obtain all node elements under parentNodeDescriptor
                var browseParameters = new DABrowseParameters();      // no filtering
whatsoever
                DANodeElementCollection nodeElementCollection =
                    client.BrowseNodes(serverDescriptor, parentNodeDescriptor,
browseParameters);
                // Remark: that BrowseNodes(...) may also throw OpcException; a
production code should contain handling for
                // it, here omitted for brevity.
            }
        }
    }
}
```

```
foreach (DANodeElement nodeElement in nodeElementCollection)
{
    Debug.Assert(nodeElement != null);

    Console.WriteLine(nodeElement);

    // If the node is a branch, browse recursively into it.
    if (nodeElement.IsBranch)
    {
        _branchCount++;
        BrowseFromNode(client, serverDescriptor, nodeElement);
    }
    else
    {
        _leafCount++;
    }
}

private static int _branchCount;
private static int _leafCount;
}
```

## Object Pascal

```
// This example shows how to obtain all nodes under the "Simulation" branch of the
// address space. For each node, it displays
// whether the node is a branch or a leaf.

class procedure BrowseNodes.Main;
var
  BrowseParameters: _DABrowseParameters;
  Client: TEasyDAClient;
  Count: Cardinal;
  Element: OleVariant;
  ServerDescriptor: _ServerDescriptor;
  NodeDescriptor: _DANodeDescriptor;
  NodeElement: _DANodeElement;
  NodeElementEnumerator: IEnumVariant;
  NodeElements: _DANodeElementCollection;
begin
  ServerDescriptor := CoServerDescriptor.Create;
  ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';

  NodeDescriptor := CoDANodeDescriptor.Create;
  NodeDescriptor.ItemId := 'Simulation';

  BrowseParameters := CoDABrowseParameters.Create;

  // Instantiate the client object
  Client := TEeasyDAClient.Create(nil);

  NodeElements := Client.BrowseNodes(
    ServerDescriptor,
    NodeDescriptor,
    BrowseParameters);

  NodeElementEnumerator := NodeElements.GetEnumerator;
```

```
while (NodeElementEnumerator.Next(1, Element, Count) = S_OK) do
begin
    NodeElement := IUnknown(Element) as _DANodeElement;
    WriteLn(NodeElement.Name, ': ', NodeElement.ItemId);
end;
end;
```

## VBScript

```
Rem This example shows how to obtain all nodes under the "Simulation" branch of the
address space. For each node, it displays
Rem whether the node is a branch or a leaf.
```

```
Option Explicit
```

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"

Dim NodeDescriptor: Set NodeDescriptor =
CreateObject("OpcLabs.EasyOpc.DataAccess.DANodeDescriptor")
NodeDescriptor.ItemID = "Simulation"

Dim BrowseParameters: Set BrowseParameters =
CreateObject("OpcLabs.EasyOpc.DataAccess.DABrowseParameters")

Dim Client: Set Client= CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim NodeElements: Set NodeElements = Client.BrowseNodes(ServerDescriptor,
NodeDescriptor, BrowseParameters)

Dim NodeElement: For Each NodeElement In NodeElements
    WScript.Echo "NodeElements("")" & NodeElement.Name & ""):""
    With NodeElement
        WScript.Echo Space(4) & ".IsBranch: " & .IsBranch
        WScript.Echo Space(4) & ".IsLeaf: " & .IsLeaf
    End With
Next
```

### 6.1.3.3 Browsing for OPC Classic Access Paths

 Access paths are somewhat obsolete feature of OPC Data Access specification, and few OPC servers 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 [DALItemDescriptor](#) object and later use that descriptor with various methods.

## VBScript

```
Rem This example shows how to obtain all access paths available for an item.
```

```
Option Explicit
```

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"

Dim NodeDescriptor: Set NodeDescriptor =
CreateObject("OpcLabs.EasyOpc.DataAccess.DANodeDescriptor")
NodeDescriptor.ItemID = "Simulation.Random"

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim accessPaths: accessPaths = Client.BrowseAccessPaths(ServerDescriptor,
NodeDescriptor)

Dim i: For i = LBound(accessPaths) To UBound(accessPaths)
    WScript.Echo "accessPaths(" & i & "): " & accessPaths(i)
Next
```

## 6.1.3.4 Browsing for OPC Classic 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. You will receive back a [DAPropertyElementCollection](#) object. 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](#).

You can use [DAPropertyId.GetName](#) and [GetPropertyType](#) methods to obtain the string identifier of the property, or its type. With OPC XML, properties are identified by a XML qualified name, and you will find it in the [DAPropertyElement.QualifiedName](#) property.

Constants for specific (well-known) OPC properties are contained in the [DAPropertyIds](#) enumeration.

### Object Pascal

```
// This example shows how to enumerate all properties of an OPC item. For each
// property, it displays its Id and description.

class procedure BrowseProperties.Main;
var
  Client: TEasyDAClient;
  Count: Cardinal;
  Element: OleVariant;
  ServerDescriptor: _ServerDescriptor;
  NodeDescriptor: _DANodeDescriptor;
  PropertyElement: _DAPropertyElement;
  PropertyElementEnumerator: IEnumVariant;
  PropertyElements: _DAPropertyElementCollection;
begin
  ServerDescriptor := CoServerDescriptor.Create;
  ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';

  NodeDescriptor := CoDANodeDescriptor.Create;
  NodeDescriptor.ItemId := 'Simulation.Random';

  // Instantiate the client object
  Client := TEeasyDAClient.Create(nil);
```

```
PropertyElements := Client.BrowseProperties(
    ServerDescriptor,
    NodeDescriptor);

PropertyElementEnumerator := PropertyElements.GetEnumerator;
while (PropertyElementEnumerator.Next(1, Element, Count) = S_OK) do
begin
    PropertyElement := IUnknown(Element) as _DAPropertyElement;
    WriteLn('PropertyElements("", PropertyElement.PropertyId.NumericalValue,
') .Description: ', PropertyElement.Description);
end;
end;
```

## VBScript

Rem This example shows how to enumerate all properties of an OPC item. For each property, it displays its Id and description.

Option Explicit

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"

Dim NodeDescriptor: Set NodeDescriptor =
CreateObject("OpcLabs.EasyOpc.DataAccess.DANodeDescriptor")
NodeDescriptor.ItemID = "Simulation.Random"

Dim EasyDAClient: Set EasyDAClient =
CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim PropertyElements: Set PropertyElements =
EasyDAClient.BrowseProperties(ServerDescriptor, NodeDescriptor)

Dim PropertyElement: For Each PropertyElement In PropertyElements
    WScript.Echo "PropertyElements("") & PropertyElement.PropertyID.NumericalValue &
") .Description: " & PropertyElement.Description
Next
```

## 6.1.3.5 Discovering OPC UA Servers

OPC UA Discovery allows QuickOPC to find available OPC UA servers, and determine how to make connections to them. There are several approaches to discovery, and they can be combined together to achieve the desired functionality. Clients and servers can be on the same host, on different hosts in the same subnet, or even on completely different locations in an administrative domain.

QuickOPC provides following kinds of discovery:

- **OPC UA Local Discovery (Section 6.1.3.5.1).** Obtains information about the servers on a specified host using OPC UA Local Discovery Server (LDS) running on that host.
- **OPC UA Global Discovery (Section 6.1.3.5.2).** Obtains information about applications available in an administrative domain by interrogating an OPC UA Global Discovery Server (GDS).

It is also possible to access the discovery functionality in a common way, using **Generalized OPC UA Discovery (Section 6.1.3.5.3)**.

### 6.1.3.5.1 OPC UA Local Discovery

QuickOPC-UA allows you to work with so called OPC-UA Local Discovery Server(s). OPC-UA Local Discovery Server is a special OPC-UA service that provides information about other OPC-UA servers available.

**⚠** The term "local discovery" might be misleading sometimes. OPC UA Local Discovery is not just for discovering OPC UA servers that reside on the same computer as where your OPC UA client is. OPC UA Local Discover allows you to discover servers on a remote computer as well, provided that you know the host name of the computer, and the OPC UA Local Discovery Server is running on that remote computer.

If you want to retrieve a list of OPC Unified Architecture servers registered on a local or remote computer, call the [DiscoverServers](#) method, passing it the name or address of the remote machine.

You will receive back a [UAApplicationElementCollection](#) object, which is collection of [UAApplicationElement](#)-s keyed by their so-called discovery URLs. The discovery URL is the main piece of information that you can further use if you want to connect to that OPC-UA server. Each [UAApplicationElement](#) contains information gathered about one OPC server found by the discovery process, including things like the application name, application type, its Product URI etc.

## C#

```
// This example shows how to obtain application URLs of all OPC Unified Architecture
// servers on a given machine.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class DiscoverServers
        {
            public static void Overload1()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Obtain collection of server elements
                UAApplicationElementCollection applicationElementCollection =
                    easyUAClient.DiscoverServers("opcua.demo-this.com");

                // Display results
                foreach (UAApplicationElement applicationElement in
                    applicationElementCollection)
                    Console.WriteLine("applicationElementCollection[\""
                        + {0} + "\"].ApplicationUriString: {1}",
                        applicationElement.DiscoveryUriString,
                        applicationElement.ApplicationUriString);

                // Example output:
                // applicationElementCollection["opc.tcp://opcua.demo-
                this.com:51210/UA/SampleServer"].ApplicationUriString: urn:Test-PC:UA Sample Server
                // applicationElementCollection["http://opcua.demo-
                this.com:51211/UA/SampleServer"].ApplicationUriString: urn:Test-PC:UA Sample Server
            }
        }
    }
}
```

## VB.NET

```
' This example shows how to obtain application URLs of all OPC Unified Architecture
```

```
servers on a given machine.  
Imports OpcLabs.EasyOpc.UA  
Imports OpcLabs.EasyOpc.UA.Discovery  
  
Namespace _EasyUAClient  
    Friend Class DiscoverServers  
        Public Shared Sub Overload1()  
            ' Instantiate the client object  
            Dim easyUAClient = New EasyUAClient()  
  
            ' Obtain collection of server elements  
            Dim applicationElementCollection As UAApplicationElementCollection =  
                easyUAClient.DiscoverServers("opcua.demo-this.com")  
  
            ' Display results  
            For Each applicationElement As UAApplicationElement In  
                applicationElementCollection  
                Console.WriteLine("applicationElementCollection[""  
{0}"""].ApplicationUriString: {1}", _  
                    applicationElement.DiscoveryUriString,  
                    applicationElement.ApplicationUriString)  
                Next applicationElement  
  
            ' Example output:  
            ' applicationElementCollection["opc.tcp://opcua.demo-  
this.com:51210/UA/SampleServer"].ApplicationUriString: urn:Test-PC:UA Sample Server  
            ' applicationElementCollection["http://opcua.demo-  
this.com:51211/UA/SampleServer"].ApplicationUriString: urn:Test-PC:UA Sample Server  
        End Sub  
    End Class  
End Namespace
```

## C++

---

```
// This example shows how to obtain application URLs of all OPC Unified Architecture  
servers on the specified host.  
  
#include "stdafx.h"  
#include "_EasyUAClient.DiscoverServers.h"  
  
  
#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorelible  
using namespace mscorelible;  
  
// OpcLabs.BaseLib  
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \  
    rename("value", "Value")  
using namespace OpcLabs_BaseLib;  
  
// OpcLabs.EasyOpcUA  
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \  
    rename("attributeData", "AttributeData") \  
    rename("browsePath", "BrowsePath") \  
    rename("endpointDescriptor", "EndpointDescriptor") \  
    rename("expandedText", "ExpandedText") \  
    rename("inputArguments", "InputArguments") \  
    rename("inputTypeCodes", "InputTypeCodes") \  
    rename("nodeDescriptor", "NodeDescriptor") \  
    rename("nodeId", "NodeId") \  
    rename("value", "Value")
```

```

using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void DiscoverServers::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(_uuidof(EasyUAClient));

            // Obtain collection of server elements
            _UAApplicationElementCollectionPtr ApplicationElementsPtr = ClientPtr-
>DiscoverServers(L"opcua.demo-this.com");

            // Display results
            IEnumVARIANTPtr EnumApplicationElementPtr = ApplicationElementsPtr-
>GetEnumerator();
            _variant_t vApplicationElement;
            while (EnumApplicationElementPtr->Next(1, &vApplicationElement, NULL) ==
S_OK)
            {
                _UAApplicationElementPtr ApplicationElementPtr(vApplicationElement);
                _tprintf(_T("ApplicationElementCollection[%s]: "),
CW2T(ApplicationElementPtr->DiscoveryUriString));
                _tprintf(_T("%s\n"), CW2T(ApplicationElementPtr-
>ApplicationUriString));
                vApplicationElement.Clear();
            }
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}

```

## Free Pascal

---

```

// This example shows how to obtain application URLs of all OPC Unified Architecture
servers on the specified host.

class procedure DiscoverServers.Main;
var
    Client: EasyUAClient;
    Count: Cardinal;
    Element: OleVariant;
    ApplicationElement: _UAApplicationElement;
    ApplicationElementEnumerator: IEnumVariant;
    ApplicationElements: _UAApplicationElementCollection;
begin
    // Instantiate the client object
    Client := CoEasyUAClient.Create;

    // Obtain collection of server elements
    ApplicationElements := Client.DiscoverServers('opcua.demo-this.com');

    // Display results
    ApplicationElementEnumerator := ApplicationElements.GetEnumerator;
    while (ApplicationElementEnumerator.Next(1, Element, Count) = S_OK) do

```

```
begin
    ApplicationElement := IUnknown(Element) as _UAApplicationElement;
    WriteLn(
        'ApplicationElements["',
        ApplicationElement.DiscoveryUriString,
        '".ApplicationUriString: ',
        ApplicationElement.ApplicationUriString);
end;
end;
```

## Object Pascal

```
// This example shows how to obtain application URLs of all OPC Unified Architecture
// servers on the specified host.

class procedure DiscoverServers.Main;
var
    Client: TEasyUAClient;
    Count: Cardinal;
    Element: OleVariant;
    ApplicationElement: _UAApplicationElement;
    ApplicationElementEnumerator: IEnumVariant;
    ApplicationElements: _UAApplicationElementCollection;
begin
    // Instantiate the client object
    Client := TEasyUAClient.Create(nil);

    // Obtain collection of server elements
    ApplicationElements := Client.DiscoverServers('opcua.demo-this.com');

    // Display results
    ApplicationElementEnumerator := ApplicationElements.GetEnumerator;
    while (ApplicationElementEnumerator.Next(1, Element, Count) = S_OK) do
    begin
        ApplicationElement := IUnknown(Element) as _UAApplicationElement;
        WriteLn(
            'ApplicationElements["',
            ApplicationElement.DiscoveryUriString,
            '".ApplicationUriString: ',
            ApplicationElement.ApplicationUriString);
    end;
end;
```

## PHP

```
// This example shows how to obtain application URLs of all OPC Unified Architecture
// servers on the specified host.

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Obtain collection of server elements
$ApplicationElementCollection = $Client->DiscoverServers("opcua.demo-this.com");

// Display results
foreach ($ApplicationElementCollection as $ApplicationElement)
{
    printf("ApplicationElementCollection[%s].ApplicationUriString: %s\n",
        $ApplicationElement->DiscoveryUriString, $ApplicationElement-
    >ApplicationUriString);
}
```

## Python

```
# This example shows how to obtain application URLs of all OPC Unified Architecture
# servers on the specified host.

import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.UA.EasyUAClient')

# Obtain collection of server elements
applicationElementCollection = client.DiscoverServers('opcua.demo-this.com')

# Display results
for applicationElement in applicationElementCollection:
    print('ApplicationElementCollection["', applicationElement.DiscoveryUriString,
'"].ApplicationUriString: ',
        applicationElement.ApplicationUriString)
```

## Visual Basic (VB 6.)

```
Rem This example shows how to obtain application URLs of all OPC Unified
Architecture servers on the specified host.

Private Sub DiscoverServers_Main_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    ' Obtain collection of server elements
    Dim ApplicationElementCollection As UAApplicationElementCollection
    Set ApplicationElementCollection = Client.DiscoverServers("opcua.demo-this.com")

    ' Display results
    Dim ApplicationElement As UAApplicationElement: For Each ApplicationElement In
ApplicationElementCollection
        OutputText = OutputText & "ApplicationElementCollection["""" &
ApplicationElement.DiscoveryUriString & """].ApplicationUriString: " & _
            ApplicationElement.ApplicationUriString & vbCrLf
    Next
End Sub
```

## VBScript

```
Rem This example shows how to obtain application URLs of all OPC Unified
Architecture servers on the specified host.

Option Explicit

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Obtain collection of server elements
Dim ApplicationElementCollection: Set ApplicationElementCollection =
Client.DiscoverServers("opcua.demo-this.com")

' Display results
Dim ApplicationElement: For Each ApplicationElement In ApplicationElementCollection
    WScript.Echo "ApplicationElementCollection["""" &
```

```
ApplicationElement.DiscoveryUriString & "")].ApplicationUriString: " & _  
    ApplicationElement.ApplicationUriString
```

[Next](#)

In .NET, if you want to connect to the discovered OPC server later in your code by calling other methods, use the built-in conversion of [UAApplicationElement](#) to [UAEEndpointDescriptor](#), and pass the resulting object as an endpointDescriptor argument either directly to some other method call.

When performing a local discovery of OPC-UA servers, the component attempts to connect to multiple possible discovery endpoints in parallel, speeding up the discovery. This behavior is controlled by the [ParallelDiscovery](#) property in the [UADiscoveryParameters](#) object. The endpoints that the components attempts to use for discovery on a given host (either sequentially, or in parallel), are given by the [DiscoveryUriTemplateStrings](#) property of the [UAHostParameters](#).

You can also bypass the use of [UAHostParameters](#), and call the [DiscoverServers](#) overload that takes an input array or URL strings to attempt discovery on.

There are also [DiscoverApplications](#) methods, which allow you to specify the application types you are interested in, using the combination of flags from the [UAApplicationTypes](#) enumeration. This can be useful e.g. if you want to do hierarchical discovery and return also the discovery servers (which are not returned by default).

The [FindApplications](#) method can be used, if you want to interrogate specific discovery endpoints.

## VBScript

```
Rem This example shows how to obtain application URLs of all OPC Unified  
Architecture servers, using specified discovery URI strings.
```

```
Option Explicit

Const UAApplicationTypes_Server = 1

Dim discoveryUriStrings(2)
discoveryUriStrings(0) = "opc.tcp://opcua.demo-this.com:4840/UADiscovery"
discoveryUriStrings(1) = "http://opcua.demo-this.com/UADiscovery/Default.svc"
discoveryUriStrings(2) = "http://opcua.demo-this.com:52601/UADiscovery"

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Obtain collection of application elements
Dim ApplicationElementCollection: Set ApplicationElementCollection =
Client.FindApplications(discoveryUriStrings, UAApplicationTypes_Server)

' Display results
Dim ApplicationElement: For Each ApplicationElement In ApplicationElementCollection
    WScript.Echo "ApplicationElementCollection["""] &
ApplicationElement.DiscoveryUriString & """].ApplicationUriString: " & _
    ApplicationElement.ApplicationUriString

```

[Next](#)

### 6.1.3.5.2 OPC UA Global Discovery

QuickOPC-UA allows you to query information contained in so called OPC-UA Global Discovery Server(s). A Global Discovery Server (GDS) maintains discovery information for OPC UA applications available in an administrative domain.

If you want to retrieve a list of OPC Unified Architecture servers in an administrative domain, call the [QueryServers](#) method, passing it the endpoint of the GDS.

In addition, you can call an overload of the [QueryServers](#) method that also takes a filter parameter of type [UAQueryServerFilter](#). Using this filter, you can limit the search only to servers that meet the specified criteria. The criteria include:

- Application name.
- Application URI string.
- Product URI string.
- Server capabilities.

You can also call the [EasyUAClient.QueryServers](#) method that takes no arguments, in which case the endpoint descriptor of the GDS will be taken from the [GdsEndpointDescriptor](#) property of the [InstanceParameters](#).

In all cases described above, you will receive back a [UAApplicationElementCollection](#) object, which is collection of [UAApplicationElement](#)-s keyed by their so-called discovery URLs. The discovery URL is the main piece of information that you can further use if you want to connect to that OPC-UA server. Each [UAApplicationElement](#) contains information gathered about one OPC server found by the discovery process, including things like the application name, server capabilities, etc.

In .NET, if you want to connect to the discovered OPC server later in your code by calling other methods, use the built-in conversion of [UAApplicationElement](#) to [UAEndpointDescriptor](#), and pass the resulting object as an `endpointDescriptor` argument either directly to some other method call.

### 6.1.3.5.3 Generalized OPC UA Discovery

The various kinds of OPC UA discovery can be generalized into just one method, [DiscoverApplications](#). This method is given a discovery query argument (of [UADiscoveryQuery](#) type) that tells it the requirements for the discovery. The specific kinds of OPC UA discovery described earlier are, in fact, implemented by constructing the discovery query appropriately, and then calling the [DiscoverApplications](#) method.

The [UADiscoveryQuery](#) type is an abstract class which is at the root of a structured class hierarchy, with several concrete subclasses:

- [UAEndpointDiscoveryQuery](#) is used when you know the LDS endpoints, and want to perform discovery on these endpoints (usually in parallel). This is a specialized kind of local discovery.
- [UAGlobalDiscoveryQuery](#) is for searching the applications globally in the administrative domain. With this query, you need to specify the GDS endpoint, and optionally a filter ([UAQueryServerFilter](#)), used to find servers that meet the specified criteria.
- [UALocalDiscoveryQuery](#) is for finding applications local to a specified host.

This generalized discovery mechanism allows for flexibility both on the component and developer sides, and possible future expansion with other kinds of discovery, without a need to introduce new methods into the [IEasyUAClient](#) interface.

### 6.1.3.6 Browsing for OPC UA Nodes

In OPC Unified Architecture, the address space is composed of nodes. The nodes are interconnected by means of references. The references do not necessarily have to be hierarchical; this means that the address space does not form a tree, but it is generally an interconnected “mesh” of nodes.

Nodes are of different node classes, and each node class has a fixed set of attributes, defined in the OPC specification. References can also be of various types, and the set of reference types is extensible.

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 criteria such as the node classes of interest, or reference types to follow.

The address space may contain all kinds of information and various node classes and types of references. In this

chapter, we will focus on browsing for node classes and reference types that are relevant for data access tasks.

There are following browse methods specialized for data access:

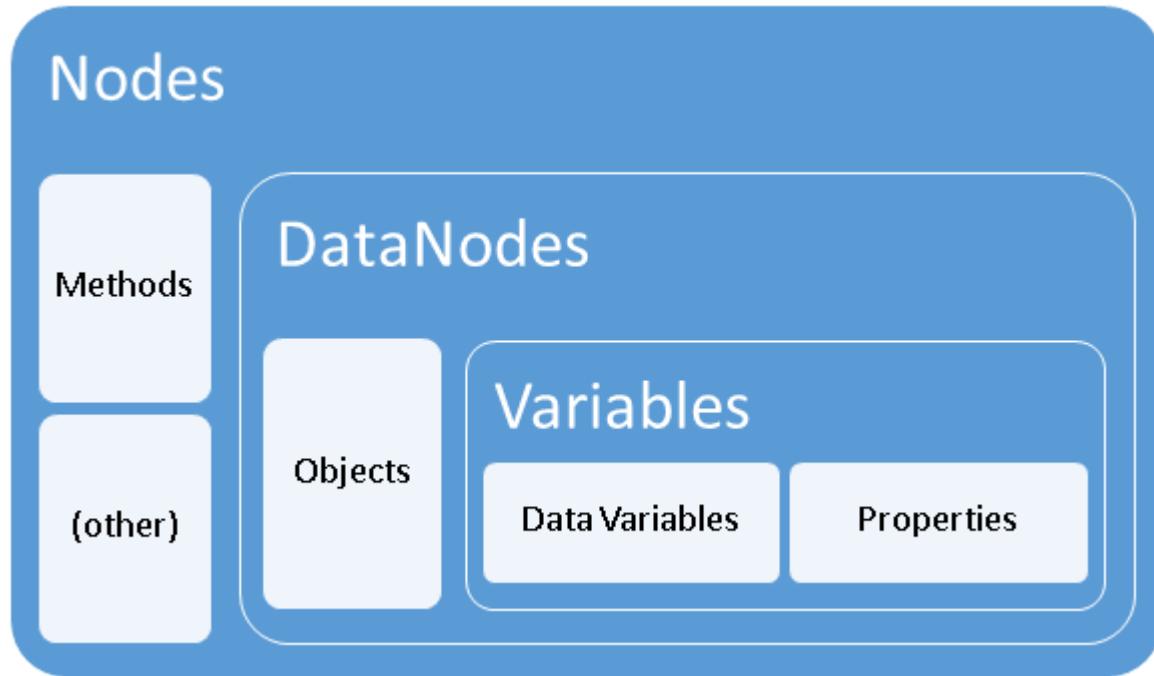
- [BrowseObjects](#)
- [BrowseDataVariables](#)
- [BrowseProperties](#)
- [BrowseVariables](#)
- [BrowseDataNodes](#)

Other browsing methods are:

- [BrowseMethods](#)
- [BrowseNodes](#)

Objects (returned by the [BrowseObjects](#) method) are equivalent of folders in a file system – they provide means of organizing other nodes logically. Data variables (returned by the [BrowseDataVariables](#) method) represent a value, and clients can read the value, write the value, or subscribe to changes of the value. Properties (returned by the [BrowseProperties](#) method) represent additional characteristics of a node which is not described by its attributes. Variables (either data variables, or properties), are returned by the [BrowseVariables](#) method. The [BrowseDataNodes](#) method returns all kinds of data nodes – and we use the term data nodes for any of objects, variables, or properties.

The following picture shows the relationship between various browsing options.



If you want to retrieve a list of all nodes that can be followed from a given node (or from an Objects folder that is a default starting point for data access browsing) of the OPC server, call one of the methods listed above. You will receive back a [UANodeElementCollection](#), which is a collection of [UANodeElement](#) objects. Each [UANodeElement](#) contains information gathered about one node that can be followed from a given node, such as its browse name, its node ID, its display name, or a type definition. The [UANodeElement](#) is convertible to [UANodeDescriptor](#), and you can therefore pass it to methods for further browsing from that node, or to methods like [Read](#), [ReadValue](#), [SubscribeMonitoredItem](#), or [SubscribeDataChange](#).

## C#

```
// This example shows how to obtain "data nodes" (objects, variables and properties)
// under the "Objects" node in the address
// space.
```

```
using System;
using OpcLabs.EasyOpc.UA;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class BrowseDataNodes
        {
            public static void Overload1()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                // Obtain data nodes under "Objects" node
                UANodeElementCollection nodeElementCollection =
easyUAClient.BrowseDataNodes(
                    "http://opcua.demo-this.com:51211/UA/SampleServer"); // or
"opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"

                // Display results
                foreach (UANodeElement nodeElement in nodeElementCollection)
                {
                    Console.WriteLine();
                    Console.WriteLine("nodeElement.NodeId: {0}",
nodeElement.NodeId);
                    Console.WriteLine("nodeElement.DisplayName: {0}",
nodeElement.DisplayName);
                }

                // Example output:
                //
                //nodeElement.NodeId: nsu=http://opcfoundation.org/UA/;i=2253
                //nodeElement.DisplayName: Server
                //
                //nodeElement.NodeId: nsu=http://test.org/UA/Data/;i=10157
                //nodeElement.DisplayName: Data
                //
                //nodeElement.NodeId: nsu=http://opcfoundation.org/UA/Boiler/;i=1240
                //nodeElement.DisplayName: Boilers
                //
                //nodeElement.NodeId: nsu=http://samples.org/UA/memorybuffer;i=1025
                //nodeElement.DisplayName: MemoryBuffers
            }
        }
    }
}
```

## VB.NET

```
' This example shows how to obtain "data nodes" (objects, variables and properties)
under the "Objects" node in the address
' space.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.AddressSpace

Namespace _EasyUAClient
    Friend Class BrowseDataNodes
        Public Shared Sub Overload1()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()
```

```
' Obtain data nodes under "Objects" node
Dim nodeElementCollection As UANodeElementCollection =
easyUAClient.BrowseDataNodes(
    "http://opcua.demo-this.com:51211/UA/SampleServer") ' or
"opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"

    ' Display results
    For Each nodeElement As UANodeElement In nodeElementCollection
        Console.WriteLine()
        Console.WriteLine("nodeElement.NodeId: {0}", nodeElement.NodeId)
        Console.WriteLine("nodeElement.DisplayName: {0}",
nodeElement.DisplayName)
    Next nodeElement

    ' Example output:
    '
    'nodeElement.NodeId: nsu=http://opcfoundation.org/UA/;i=2253
    'nodeElement.DisplayName: Server
    '
    'nodeElement.NodeId: nsu=http://test.org/UA/Data/;i=10157
    'nodeElement.DisplayName: Data
    '
    'nodeElement.NodeId: nsu=http://opcfoundation.org/UA/Boiler/;i=1240
    'nodeElement.DisplayName: Boilers
    '
    'nodeElement.NodeId: nsu=http://samples.org/UA/memorybuffer;i=1025
    'nodeElement.DisplayName: MemoryBuffers
End Sub
End Class
End Namespace
```

## C++

---

```
// This example shows how to obtain all data nodes (objects and variables) under a
// given node of the OPC-UA address space.
// For each node, it displays its browse name and node ID.

#include "stdafx.h"
#include "_EasyUAClient.BrowseDataNodes.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorelible
using namespace mscorelible;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
```

```

rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void BrowseDataNodes::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // Instantiate the client object
            _EasyUAClientPtr ClientPtr(_uuidof(EasyUAClient));

            // Perform the operation
            UANodeElementCollectionPtr NodeElementsPtr = ClientPtr-
>BrowseDataNodes(
                L"http://opcua.demo-this.com:51211/UA/SampleServer",
                L"nsu=http://test.org/UA/Data/;i=10791");

            // Display results
            IEnumVARIANTPtr EnumNodeElementPtr = NodeElementsPtr->GetEnumerator();
            _variant_t vNodeElement;
            while (EnumNodeElementPtr->Next(1, &vNodeElement, NULL) == S_OK)
            {
                _UANodeElementPtr NodeElementPtr(vNodeElement);
                _tprintf(_T("%s: "), CW2T(NodeElementPtr->BrowseName->ToString));
                _tprintf(_T("%s\n"), CW2T(NodeElementPtr->NodeId->ToString));
                vNodeElement.Clear();
            }
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}

```

## Free Pascal

---

```

// This example shows how to obtain all data nodes (objects and variables)
// under a given node of the OPC-UA address space. For each node, it displays
// its browse name and node ID.

class procedure BrowseDataNodes.Main;
var
    Client: EasyUAClient;
    Count: Cardinal;
    Element: OleVariant;
    NodeElement: _UANodeElement;
    NodeElementEnumerator: IEnumVariant;
    NodeElements: _UANodeElementCollection;
begin
    // Instantiate the client object
    Client := CoEasyUAClient.Create;

    NodeElements := Client.BrowseDataNodes(
        'http://opcua.demo-this.com:51211/UA/SampleServer',
        'nsu=http://test.org/UA/Data/;i=10791');

    NodeElementEnumerator := NodeElements.GetEnumerator;
    while (NodeElementEnumerator.Next(1, Element, Count) = S_OK) do

```

```
begin
  NodeElement := IUnknown(Element) as _UANodeElement;
  WriteLn(NodeElement.BrowseName.ToString, ': ', NodeElement.NodeId.ToString);
end;
end;
```

## Object Pascal

```
// This example shows how to obtain all data nodes (objects and variables)
// under a given node of the OPC-UA address space. For each node, it displays
// its browse name and node ID.

class procedure BrowseDataNodes.Main;
var
  Client: TEasyUAClient;
  Count: Cardinal;
  Element: OleVariant;
  NodeElement: _UANodeElement;
  NodeElementEnumerator: IEnumVariant;
  NodeElements: _UANodeElementCollection;
begin
  // Instantiate the client object
  Client := TEasyUAClient.Create(nil);

  NodeElements := Client.BrowseDataNodes(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10791');

  NodeElementEnumerator := NodeElements.GetEnumerator;
  while (NodeElementEnumerator.Next(1, Element, Count) = S_OK) do
  begin
    NodeElement := IUnknown(Element) as _UANodeElement;
    WriteLn(NodeElement.BrowseName.ToString, ': ', NodeElement.NodeId.ToString);
  end;
end;
```

## PHP

```
// This example shows how to obtain all data nodes (objects and variables) under a
// given node of the OPC-UA address space.
// For each node, it displays its browse name and node ID.

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Perform the operation
$NodeElements = $Client->BrowseDataNodes("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10791");

// Display results
foreach ($NodeElements as $NodeElement)
{
  printf("%s: %s\n", $NodeElement->BrowseName, $NodeElement->NodeId);
}
```

## Python

```
# This example shows how to obtain all data nodes (objects and variables) under a
# given node of the OPC-UA address space.
# For each node, it displays its browse name and node ID.
```

```
import win32com.client

# Instantiate the client object
client = win32com.client.Dispatch('OpcLabs.EasyOpc.UA.EasyUAClient')

# Perform the operation
nodeElements = client.BrowseDataNodes('http://opcua.demo-
this.com:51211/UA/SampleServer', 'nsu=http://test.org/UA/Data/;i=10791')

# Display results
for nodeElement in nodeElements:
    print(nodeElement.BrowseName, ': ', nodeElement.NodeId)
```

## Visual Basic (VB 6.)

```
Rem This example shows how to obtain all data nodes (objects and variables) under a
given node of the OPC-UA address space.
Rem For each node, it displays its browse name and node ID.
```

```
Private Sub BrowseDataNodes_Main_Command_Click()
    OutputText = ""

    ' Instantiate the client object
    Dim Client As New EasyUAClient

    ' Perform the operation
    Dim NodeElements As UANodeElementCollection
    Set NodeElements = Client.BrowseDataNodes("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10791")

    ' Display results
    Dim NodeElement: For Each NodeElement In NodeElements
        OutputText = OutputText & NodeElement.BrowseName & ":" & NodeElement.NodeId
    & vbCrLf
    Next
End Sub
```

## VBScript

```
Rem This example shows how to obtain all data nodes (objects and variables) under a
given node of the OPC-UA address space.
Rem For each node, it displays its browse name and node ID.
```

```
Option Explicit

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Perform the operation
Dim NodeElements: Set NodeElements = Client.BrowseDataNodes("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10791")

' Display results
Dim NodeElement: For Each NodeElement In NodeElements
    WScript.Echo NodeElement.BrowseName & ":" & NodeElement.NodeId
Next
```

Note: In some cases, defined by the OPC UA specification, the elements returned in the [UANodeElementCollection](#) will have unique browse names. In general, however, OPC UA allows multiple returned elements share the same browse name.

The OPC UA address space may also contain Methods. They can be used to call (invoke) a code in the OPC UA server, passing input and output arguments to/from the method. Methods available on a specified object are returned by the [BrowseMethods](#) call.

The most generic address space browsing method is [BrowseNodes](#). It allows the widest range of filtering options by passing in an argument of type [UABrowseParameters](#). You can specify any set of node classes, and reference type IDs, with this object.

Using the [ReferenceTypelds](#) property in the [UABrowseParameters](#), you can specify which references in the address space will be followed in browsing. The [IncludeSubtypes](#) flag determines whether subtypes of the specified reference type should be returned by the browsing.

Using the [BrowseDirections](#) property in the [UABrowseParameters](#), you can specify which directions of the references the [BrowseNodes](#) method should return. The available choices are given by the [UABrowseDirections](#) enumeration, and are [Forward](#), [Inverse](#), and [Both](#).

## 6.1.4 Subscribing to Information

If your application needs to monitor changes of certain process value (OPC Classic item, OPC UA monitored 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 Classic items or OPC UA monitored items, change the subscription parameters, and unsubscribe.

### 6.1.4.1 Subscribing to OPC Classic 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.

#### A single item

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 [EasyDALItemChangedEventArgs.Arguments.State](#) property. The [SubscribeItem](#) method returns a subscription handle that you can later use to change the subscription parameters, or unsubscribe.

#### Object Pascal

```
// This example shows how to subscribe to changes of a single item and display the value of the item with each change.
```

```
type
```

```
TSubscribeItem_ClientEventHandlers = class
  // Item changed event handler
  procedure OnItemChanged(
    ASender: TObject;
    sender: OleVariant;
    const eventArgs: _EasyDAItemChangedEventArgs);
  end;

procedure TSubscribeItem_ClientEventHandlers.OnItemChanged(
  ASender: TObject;
  sender: OleVariant;
  const eventArgs: _EasyDAItemChangedEventArgs);
begin
  WriteLn(eventArgs.Vtq.ToString());
end;

class procedure SubscribeItem.Main;
var
  Client: TEasyDAClient;
  ClientEventHandlers: TSubscribeItem_ClientEventHandlers;
begin
  // Instantiate the client object and hook events
  Client := TEeasyDAClient.Create(nil);
  ClientEventHandlers := TSubscribeItem_ClientEventHandlers.Create;
  Client.OnItemChanged := ClientEventHandlers.OnItemChanged;

  Client.SubscribeItem('', 'OPCLabs.KitServer.2', 'Simulation.Random', 1000);

  WriteLn('Processing item changed events for 1 minute...');

  PumpSleep(60*1000);

  WriteLn('Unsubscribing...');
  Client.UnsubscribeAllItems;
end;
```

## PHP

```
// This example shows how to subscribe to changes of a single item and display the value
// of the item with each change.
//
// Some related documentation: http://php.net/manual/en/function.com-event-sink.php .
Pay attention to the comment that says
// "Be careful how you use this feature; if you are doing something similar to the
example below, then it doesn't really make
// sense to run it in a web server context.". What they are trying to say is that
processing a web request should be
// a short-lived code, which does not fit well with the idea of being subscribed to
events and received them over longer time.
// It is possible to write such code, but it is only useful when processing the request
is allowed to take relatively long. Or,
// when you are using PHP from command-line, or otherwise - not to serve a web page
directly.
//
// Subscribing to QuickOPC-COM events in the context of PHP Web application, while not
imposing the limitations to the request
// processing time, has to be "worked around", e.g. using the "event pull" mechanism.

class DEasyDAClientEvents {
  function ItemChanged($varSender, $varE)
  {
    print $varE->Vtq->ToString();
    print "\n";
  }
}
```

```
$Client = new COM("OpcLabs.EasyOpc.DataAccess.EasyDAClient");
$Events = new DEasyDAClientEvents();
com_event_sink($Client, $Events, "DEasyDAClientEvents");

$client->SubscribeItem("", "OPCLabs.KitServer.2", "Simulation.Random", 1000);

print "Processing item changed events for 1 minute...\n";
for ($time = 0; $time < 60; $time++) com_message_pump(1000);
```

## VBScript

Rem This example shows how to subscribe to changes of a single item and display the value of the item with each change.

Option Explicit

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

Client.SubscribeItem "", "OPCLabs.KitServer.2", "Simulation.Random", 1000

WScript.Echo "Processing item changed events for 1 minute..."
WScript.Sleep 60*1000
```

```
Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Vtq.ToString
End Sub
```



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 [EasyDALItemChangedEventArgs](#), without having to look it up by ItemId or so.

## Multiple items

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 integers, which are the subscription handles.

You pass in an array of [DAItemGroupArguments](#) objects (each containing information for a single subscription to be made), to the [SubscribeMultipleItems](#) method.

## C#

```
// This example shows how subscribe to changes of multiple items and display the value
of the item with each change.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.DataAccess;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyDAClient
    {
```

```

partial class SubscribeMultipleItems
{
    public static void Main()
    {
        using (var easyDAClient = new EasyDAClient())
        {
            easyDAClient.ItemChanged += easyDAClient_ItemChanged;

            easyDAClient.SubscribeMultipleItems(
                new[] {
                    new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Simulation.Random", 1000, null),
                    new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Trends.Ramp (1 min)", 1000, null),
                    new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Trends.Sine (1 min)", 1000, null),
                    new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 1000, null)
                });
        }

        Console.WriteLine("Processing item changed events for 1 minute...");
        Thread.Sleep(60 * 1000);
    }
}

// Item changed event handler
static void easyDAClient_ItemChanged([NotNull] object sender, [NotNull]
EasyDAItemChangedEventArgs e)
{
    Console.WriteLine("{0}: {1}", e.Arguments.ItemDescriptor.ItemId, e.Vtq);
}
}
}
}

```

## VB.NET

```

' This example shows how subscribe to changes of multiple items and display the value of
the item with each change.

Imports JetBrains.Annotations
Imports OpcLabs.EasyOpc.DataAccess
Imports System.Threading

Namespace _EasyDAClient
    Partial Friend Class SubscribeMultipleItems
        Public Shared Sub Main()
            Using easyDAClient = New EasyDAClient()
                AddHandler easyDAClient.ItemChanged, AddressOf easyDAClient_ItemChanged

                easyDAClient.SubscribeMultipleItems(New DAIItemGroupArguments() { _
                    New DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Simulation.Random", 1000, Nothing), _
                    New DAIItemGroupArguments("", "OPCLabs.KitServer.2", "Trends.Ramp (1
min)", 1000, Nothing), _
                    New DAIItemGroupArguments("", "OPCLabs.KitServer.2", "Trends.Sine (1
min)", 1000, Nothing), _
                    New DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 1000, Nothing) _
                })

                Console.WriteLine("Processing item changed events for 1 minute...")
                Thread.Sleep(60 * 1000)
            End Using
        End Sub
    End Class
End Namespace

```

```

    End Sub

    ' Item changed event handler
    Private Shared Sub easyDAClient_ItemChanged(<NotNull()> ByVal sender As Object,
<NotNull()> ByVal e As EasyDAItemChangedEventArgs)
        ' Display the data
        ' Remark: Production code would check e.Exception before accessing e.Vtq.
        Console.WriteLine("{0}: {1}", e.Arguments.ItemDescriptor.ItemId, e.Vtq)
    End Sub
End Class
End Namespace

```

## Object Pascal

---

```

// This example shows how to subscribe to changes of multiple items and display the
value of the item with each change.

type
    TSubscribeMultipleItems_ClientEventHandlers = class
        // Item changed event handler
        procedure OnItemChanged(
            ASender: TObject;
            sender: OleVariant;
            const eventArgs: _EasyDAItemChangedEventArgs);
    end;

procedure TSubscribeMultipleItems_ClientEventHandlers.OnItemChanged(
    ASender: TObject;
    sender: OleVariant;
    const eventArgs: _EasyDAItemChangedEventArgs);
begin
    WriteLn(eventArgs.Arguments.ItemDescriptor.ItemId, ': ', eventArgs.Vtq.ToString());
end;

class procedure SubscribeMultipleItems.Main;
var
    Arguments: OleVariant;
    Client: TEasyDAClient;
    ClientEventHandlers: TSubscribeMultipleItems_ClientEventHandlers;
    HandleArray: OleVariant;
    ItemSubscriptionArguments1: _EasyDAItemSubscriptionArguments;
    ItemSubscriptionArguments2: _EasyDAItemSubscriptionArguments;
    ItemSubscriptionArguments3: _EasyDAItemSubscriptionArguments;
    ItemSubscriptionArguments4: _EasyDAItemSubscriptionArguments;
begin
    ItemSubscriptionArguments1 := CoEasyDAItemSubscriptionArguments.Create;
    ItemSubscriptionArguments1.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
    ItemSubscriptionArguments1.ItemDescriptor.ItemID := 'Simulation.Random';
    ItemSubscriptionArguments1.GroupParameters.RequestedUpdateRate := 1000;

    ItemSubscriptionArguments2 := CoEasyDAItemSubscriptionArguments.Create;
    ItemSubscriptionArguments2.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
    ItemSubscriptionArguments2.ItemDescriptor.ItemID := 'Trends.Ramp (1 min)';
    ItemSubscriptionArguments2.GroupParameters.RequestedUpdateRate := 1000;

    ItemSubscriptionArguments3 := CoEasyDAItemSubscriptionArguments.Create;
    ItemSubscriptionArguments3.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
    ItemSubscriptionArguments3.ItemDescriptor.ItemID := 'Trends.Sine (1 min)';
    ItemSubscriptionArguments3.GroupParameters.RequestedUpdateRate := 1000;

    ItemSubscriptionArguments4 := CoEasyDAItemSubscriptionArguments.Create;
    ItemSubscriptionArguments4.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';
    ItemSubscriptionArguments4.ItemDescriptor.ItemID := 'Simulation.Register_I4';
    ItemSubscriptionArguments4.GroupParameters.RequestedUpdateRate := 1000;

```

```
Arguments := VarArrayCreate([0, 3], varVariant);
Arguments[0] := ItemSubscriptionArguments1;
Arguments[1] := ItemSubscriptionArguments2;
Arguments[2] := ItemSubscriptionArguments3;
Arguments[3] := ItemSubscriptionArguments4;

// Instantiate the client object and hook events
Client := TEasyDAClient.Create(nil);
ClientEventHandlers := TSubscribeMultipleItems_ClientEventHandlers.Create;
Client.OnItemChanged := ClientEventHandlers.OnItemChanged;

TVarData(HandleArray).VType := varArray or varVariant;
TVarData(HandleArray).VArray := PVarArray(
    Client.SubscribeMultipleItems(PSafeArray(TVarData(Arguments).VArray))) ;

WriteLn('Processing item changed events for 1 minute...');
PumpSleep(60*1000);

WriteLn('Unsubscribing...');

Client.UnsubscribeAllItems;
end;
```

## VBScript

Rem This example shows how to subscribe to changes of multiple items and display the value of the item with each change.

```
Option Explicit
```

```
Dim ItemSubscriptionArguments1: Set ItemSubscriptionArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments1.ItemDescriptor.ItemID = "Simulation.Random"
ItemSubscriptionArguments1.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments2: Set ItemSubscriptionArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments2.ItemDescriptor.ItemID = "Trends.Ramp (1 min)"
ItemSubscriptionArguments2.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments3: Set ItemSubscriptionArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments3.ItemDescriptor.ItemID = "Trends.Sine (1 min)"
ItemSubscriptionArguments3.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments4: Set ItemSubscriptionArguments4 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments4.ItemDescriptor.ItemID = "Simulation.Register_I4"
ItemSubscriptionArguments4.GroupParameters.RequestedUpdateRate = 1000

Dim arguments(3)
Set arguments(0) = ItemSubscriptionArguments1
Set arguments(1) = ItemSubscriptionArguments2
Set arguments(2) = ItemSubscriptionArguments3
Set arguments(3) = ItemSubscriptionArguments4
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

Client.SubscribeMultipleItems arguments

WScript.Echo "Processing item changed events for 1 minute..."
WScript.Sleep 60*1000

Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Arguments.ItemDescriptor.ItemId & ":" & e.Vtq
End Sub
```

## Common considerations

Note: It is NOT an error to subscribe to the same item 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 items. Internally, however, the subscription made to the OPC server will be optimized (merged together) if possible.

### 6.1.4.2 Subscribing to OPC UA Monitored Items

Subscription is initiated by calling one of the [SubscribeMonitoredItem](#), [SubscribeDataChange](#), or [SubscribeMultipleMonitoredItems](#) method. For any significant change in the subscribed item, your application will receive the [DataChangeNotification](#) 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 [SubscribeMonitoredItem](#), [SubscribeDataChange](#) or [SubscribeMultipleMonitoredItems](#) call.

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. You may also want to influence the amount of received information, or control how and how often the information is delivered. QuickOPC-UA has two sets of parameters for this: subscription parameters and monitoring parameters.

Subscription parameters (can be fully contained in [UASubscriptionParameters](#) object) influence the information delivery. Main subscription parameters are the relative priority, and publishing interval (in milliseconds). The publishing interval defines the cyclic rate that the subscription is requested to return notifications to the client.

Monitoring parameters (can be fully contained in [UAMonitoringParameters](#) object) influence which information is delivered to the client. Main monitoring parameters are the data change filter (optional), size of monitored item queue, and sampling interval (in milliseconds; it is the fastest rate at which the monitored items should be accessed and evaluated).

The data change filter, if present, is contained in the [UADataChangeFilter](#) object, and defines the conditions under which a data change notification should be reported and, optionally, a deadband for value changes where no notification is generated. The [Trigger](#) property selects whether a notification is generated only when the status code associated with the value changes, or either the status code or the value change, or if any of the status code, value, or a source timestamp change. The [DeadbandType](#) property determines the type of deadband used, and can either select no deadband, absolute deadband, or percent deadband.

There are implicit conversions available from [System.Double](#) and [UADataChangeTrigger](#) to [UADataChangeFilter](#). This allows the developer to simply use the absolute deadband, or the trigger selection, in place of any data change filter.

In order to make the settings of various parameters easier, QuickOPC-UA allows you to omit the specification of publishing interval in subscription parameters by leaving it at zero (the default). In this case, the component will calculate the publishing interval by dividing the sampling interval by the so-called automatic publishing factor ([EasyUAClient.SharedParameters.EngineParameters.AutomaticPublishingFactor](#)) with a default pre-set by the component. The resulting value is limited by the value of [EasyUAClient.SharedParameters.EngineParameters.FastestAutomaticPublishingInterval](#) property. This algorithm should

determine a suitable publishing interval in the most common cases.

## A single node and attribute

If you want to subscribe to a specific value in OPC address space, call the [SubscribeDataChange](#) or [SubscribeMonitoredItem](#) method. These methods have number of overloads, but in its simplest form, only three arguments are necessary: an endpoint descriptor, node ID, and a sampling interval. Other callbacks allow you to specify e.g. the callback method, a user-defined state object, absolute deadband value, or pass in a full set of parameters using [EasyUAMonitoredItemArguments](#) object. The [SubscribeDataChange](#) and [SubscribeMonitoredItem](#) methods return a subscription handle that you can later use to change the subscription parameters, or unsubscribe.

It is common to pass in a [State](#) argument of type [Object](#). When the monitored item's value changes, the [State](#) argument is then passed to the [DataChangeNotification](#) event handler (or your callback method) in the [EasyUADataChangeEventArgs.Arguments](#) 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 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 [EasyUADataChangeEventArgs.Arguments](#), without having to look it up by Node Id or so.

### C#

```
// This example shows how to subscribe to changes of a single monitored item and
// display the value of the item with each change.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class SubscribeDataChange
        {
            public static void Overload1()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
                    easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeDataChange(
                    "http://opc.tcp://opcua.demo-this.com:51211/UA/SampleServer", // or
                    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                    "nsu=http://test.org/UA/Data/;i=10853",
                    1000);

                Console.WriteLine("Processing data change events for 20
seconds...");
                System.Threading.Thread.Sleep(20 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeAllMonitoredItems();

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }
        }
    }
}
```

```
        static void easyUAClient_DataChangeNotification(object sender,
EasyUADataChangeNotificationEventArgs e)
    {
        // Display value
        // Remark: Production code would check e.Exception before accessing
e.AttributeData.
        Console.WriteLine(e.AttributeData.Value);
    }
}
}
```

## VB.NET

```
' This example shows how to subscribe to changes of a single monitored item and
display the value of the item with each change.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.OperationModel

Namespace _EasyUAClient
    Friend Class SubscribeDataChange
        Public Shared Sub Overload1()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            easyUAClient.SubscribeDataChange("http://opcua.demo-
this.com:51211/UA/SampleServer", _
                                         "nsu=http://test.org/UA/Data/;i=10853",
1000) ' or "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"

            Console.WriteLine("Processing monitored item changed events for 10
seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Unsubscribing...")
            easyUAClient.UnsubscribeAllMonitoredItems()

            Console.WriteLine("Waiting for 5 seconds...")
            Threading.Thread.Sleep(5 * 1000)
        End Sub

        Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As
Object, ByVal e As EasyUADataChangeNotificationEventArgs)
            ' Display value
            ' Remark: Production code would check e.Exception before accessing
e.AttributeData.
            Console.WriteLine(e.AttributeData.Value)
        End Sub
    End Class
End Namespace
```

## C++

```
// This example shows how to subscribe to changes of a single monitored item and
display each change.

#include "stdafx.h"
#include <atlcom.h>
```

```
#include "_EasyUAClient.SubscribeDataChange.h"

// OpcLabs.BaseLib
#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

#define DISPID_EASYUAClientEvents_DataChangeNotification 1002

namespace _EasyUAClient
{
    // CEeasyUAClientEvents

    class CEeasyUAClientEvents : public IDispEventImpl<1, CEeasyUAClientEvents>
    {
        public:
            BEGIN_SINK_MAP(CEeasyUAClientEvents)
                // Event handlers must have the __stdcall calling convention
                SINK_ENTRY(1, DISPID_EASYUAClientEvents_DataChangeNotification,
                           &CEeasyUAClientEvents::DataChangeNotification)
            END_SINK_MAP()

            public:
                // The handler for EasyUAClient.DataChangeNotification event
                STDMETHOD(DataChangeNotification)(VARIANT varSender,
                    _EasyUADataChangeEventArgs* pEventArgs)
                {
                    // Display the data
                    // Remark: Production code would check EventArgsPtr->Exception before
                    // accessing EventArgsPtr->AttributeData.
                    _UAAttributeDataPtr AttributeDataPtr(pEventArgs->AttributeData);
                    _tprintf(_T("%s\n"), CW2T(AttributeDataPtr->ToString()));

                    return S_OK;
                }
            };

        void SubscribeDataChange::Main()
        {
            // Initialize the COM library
```

```

CoInitializeEx(NULL, COINIT_MULTITHREADED);
{
    // Instantiate the client object
    _EasyUAClientPtr ClientPtr(_uuidof(EasyUAClient));

    // Hook events
    CEasyUAClientEvents* pClientEvents = new CEasyUAClientEvents();
    AtlGetObjectSourceInterface(ClientPtr, &pClientEvents->m_lbid,
&pClientEvents->m_iid,
        &pClientEvents->m_wMajorVerNum, &pClientEvents->m_wMinorVerNum);
    pClientEvents->m_iid = _uuidof(DEasyUAClientEvents);
    pClientEvents->DispEventAdvise(ClientPtr, &pClientEvents->m_iid);

    //
    _tprintf(_T("Subscribing...\n"));
    ClientPtr->SubscribeDataChange(
        L"http://opcua.demo-this.com:51211/UA/SampleServer",
        L"nsu=http://test.org/UA/Data/;i=10853",
        1000);

    _tprintf(_T("Processing monitored item changed events for 1
minute...\n"));
    Sleep(60*1000);

    // Unhook events
    pClientEvents->DispEventUnadvise(ClientPtr, &pClientEvents->m_iid);
}
// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}
}

```

## PHP

---

```

// This example shows how to subscribe to changes of a single monitored item and
// display each change.

class ClientEvents {
    function DataChangeNotification($Sender, $E)
    {
        // Display the data
        // Remark: Production code would check e.Exception before accessing
e.AttributeData.
        printf("%s\n", $E->AttributeData);
    }
}

// Instantiate the client object and hook events
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");
$ClientEvents = new ClientEvents();
com_event_sink($Client, $ClientEvents, "DEasyUAClientEvents");

printf("Subscribing...\n");
$Client->SubscribeDataChange(
    "http://opcua.demo-this.com:51211/UA/SampleServer",
    "nsu=http://test.org/UA/Data/;i=10853",
    1000);

printf("Processing monitored item changed events for 1 minute... ");

```

```
$startTime = time(); do { com_message_pump(1000); } while (time() < $startTime + 60);
```

## Visual Basic (VB 6.)

Rem This example shows how to subscribe to changes of a single monitored item and display each change.

```
' The client object, with events
'Public WithEvents Client1 As EasyUAClient

Private Sub SubscribeDataChange_Main_Command_Click()
    OutputText = ""

    Set Client1 = New EasyUAClient

    OutputText = OutputText & "Subscribing..." & vbCrLf
    Call Client1.SubscribeDataChange("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000)

    OutputText = OutputText & "Processing data changed notification events for 1
minute..." & vbCrLf
    Pause 60000

    OutputText = OutputText & "Unsubscribing..." & vbCrLf
    Client1.UnsubscribeAllMonitoredItems

    OutputText = OutputText & "Waiting for 5 seconds..." & vbCrLf
    Pause 5000

    Set Client1 = Nothing
End Sub

Private Sub Client1_DataChangeNotification(ByVal sender As Variant, ByVal eventArgs
As EasyUADataChangeNotificationEventArgs)
    ' Display the data
    ' Remark: Production code would check eventArgs.Exception before accessing
    eventArgs.AttributeData.
    OutputText = OutputText & eventArgs.AttributeData & vbCrLf
End Sub
```

## VBScript

Rem This example shows how to subscribe to changes of a single monitored item and display each change.

```
Option Explicit

' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Client.SubscribeDataChange "http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853", 1000

WScript.Echo "Processing monitored item changed events for 1 minute..."
WScript.Sleep 60*1000
```

```
Sub Client_DataChangeNotification(Sender, e)
    ' Display the data
    Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else
    display = e.ErrorMessageBrief
    WScript.Echo display
End Sub
```

## Multiple nodes or attributes

To subscribe to multiple items simultaneously in an efficient manner, call the `SubscribeMultipleMonitoredItems` method (instead of multiple `SubscribeDataChange` or `SubscribeMonitoredItem` calls in a loop). You receive back an array of subscription handles. You pass in an array of `EasyUAMonitoredItemArguments` objects (each containing information for a single subscription to be made), to the `SubscribeMultipleMonitoredItems` method.

### C#

```
// This example shows how to subscribe to changes of multiple monitored items and
// display the value of the monitored item with
// each change.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class SubscribeMultipleMonitoredItems
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
                    easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeMultipleMonitoredItems(new[]
                {
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10845", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10853", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10855", 1000)
                });

                Console.WriteLine("Processing monitored item changed events for 10
seconds...");
                System.Threading.Thread.Sleep(10 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeAllMonitoredItems();

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }
        }
    }
}
```

```
    static void easyUAClient_DataChangeNotification(object sender,
EasyUADataChangeEventArgs e)
    {
        // Display value
        // Remark: Production code would check e.Exception before accessing
e.AttributeData.
        Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,
e.AttributeData.Value);
    }
}
```

## VB.NET

```
' This example shows how to subscribe to changes of multiple monitored items and
display the value of the monitored item with
' each change.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class SubscribeMultipleMonitoredItems
        Public Shared Sub Main()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            easyUAClient.SubscribeMultipleMonitoredItems(New
EasyUAMonitoredItemArguments() _
{
    New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845", 1000),
    New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000),
    New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855", 1000)
})
}

Console.WriteLine("Processing monitored item changed events for 10
seconds...")
Threading.Thread.Sleep(10 * 1000)

Console.WriteLine("Unsubscribing...")
easyUAClient.UnsubscribeAllMonitoredItems()

Console.WriteLine("Waiting for 5 seconds...")
Threading.Thread.Sleep(5 * 1000)
End Sub

Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As
Object, ByVal e As EasyUADataChangeEventArgs)
    ' Display value
    ' Remark: Production code would check e.Exception before accessing
e.AttributeData.
    Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,
e.AttributeData.Value)

```

```
    End Sub
End Class
End Namespace
```

## C++

---

```
// This example shows how to subscribe to changes of multiple monitored items and
// display the value of the monitored item with
// each change.

#include "stdafx.h"
#include <atlcom.h>
#include <atlsafe.h>
#include "_EasyUAClient.SubscribeMultipleMonitoredItems.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"           // mscorelible
using namespace mscorelible;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

#define DISPID_EASYUAClientEvents_DataChangeNotification 1002

namespace _EasyUAClient
{
    // CEeasyUAClientEvents

    class CEeasyUAClientEvents : public IDispEventImpl<1, CEeasyUAClientEvents>
    {
        public:
            BEGIN_SINK_MAP(CEeasyUAClientEvents)
                // Event handlers must have the __stdcall calling convention
                SINK_ENTRY(1, DISPID_EASYUAClientEvents_DataChangeNotification,
                           &CEeasyUAClientEvents::DataChangeNotification)
            END_SINK_MAP()

            public:
                // The handler for EasyUAClient.DataChangeNotification event
                STDMETHOD(DataChangeNotification)(VARIANT varSender,
                _EasyUADataChangeNotificationEventArgs* pEventArgs)
                {
                    // Display the data
```

```

        _tprintf(_T("%s: "), CW2T(pEventArgs->Arguments->NodeDescriptor-
>ToString));
        // Remark: Production code would check EventArgsPtr->Exception before
accessing EventArgsPtr->AttributeData.
        _UAAttributeDataPtr AttributeDataPtr(pEventArgs->AttributeData);
        _tprintf(_T("%s\n"), CW2T(AttributeDataPtr->ToString));

        return S_OK;
    }
};

void SubscribeMultipleMonitoredItems::Main()
{
    // Initialize the COM library
    CoInitializeEx(NULL, COINIT_MULTITHREADED);
    {
        // Instantiate the client object
        _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

        // Hook events
        CEasyUAClientEvents* pClientEvents = new CEasyUAClientEvents();
        AtlGetObjectSourceInterface(ClientPtr, &pClientEvents->m_lbid,
&pClientEvents->m_iid,
            &pClientEvents->m_wMajorVerNum, &pClientEvents->m_wMinorVerNum);
        pClientEvents->m_iid = __uuidof(DEasyUAClientEvents);
        pClientEvents->DispEventAdvise(ClientPtr, &pClientEvents->m_iid);

        //
        _tprintf(_T("Subscribing...\n"));

        _UAMonitoringParametersPtr
MonitoringParametersPtr(__uuidof(UAMonitoringParameters));
        MonitoringParametersPtr->SamplingInterval = 1000;

        _UAMonitoredItemArgumentsPtr
MonitoringArguments1Ptr(__uuidof(EasyUAMonitoredItemArguments));
        MonitoringArguments1Ptr->EndpointDescriptor->UrlString =
L"http://opcua.demo-this.com:51211/UA/SampleServer";
        MonitoringArguments1Ptr->NodeDescriptor->NodeId->ExpandedText =
I"nsu=http://test.org/UA/Data/;i=10845";
        MonitoringArguments1Ptr->MonitoringParameters = MonitoringParametersPtr;

        _UAMonitoredItemArgumentsPtr
MonitoringArguments2Ptr(__uuidof(EasyUAMonitoredItemArguments));
        MonitoringArguments2Ptr->EndpointDescriptor->UrlString =
L"http://opcua.demo-this.com:51211/UA/SampleServer";
        MonitoringArguments2Ptr->NodeDescriptor->NodeId->ExpandedText =
I"nsu=http://test.org/UA/Data/;i=10853";
        MonitoringArguments2Ptr->MonitoringParameters = MonitoringParametersPtr;

        _UAMonitoredItemArgumentsPtr
MonitoringArguments3Ptr(__uuidof(EasyUAMonitoredItemArguments));
        MonitoringArguments3Ptr->EndpointDescriptor->UrlString =
L"http://opcua.demo-this.com:51211/UA/SampleServer";
        MonitoringArguments3Ptr->NodeDescriptor->NodeId->ExpandedText =
I"nsu=http://test.org/UA/Data/;i=10855";
        MonitoringArguments3Ptr->MonitoringParameters = MonitoringParametersPtr;

        CComSafeArray<VARIANT> arguments(3);
        arguments.SetAt(0, _variant_t((IDispatch*)MonitoringArguments1Ptr));

```

```

arguments.SetAt(1, _variant_t((IDispatch*)MonitoringArguments2Ptr));
arguments.SetAt(2, _variant_t((IDispatch*)MonitoringArguments3Ptr));

LPSAFEARRAY pArguments = arguments.Detach();
CComSafeArray<VARIANT> handles;
handles.Attach(ClientPtr->SubscribeMultipleMonitoredItems(&pArguments));
arguments.Attach(pArguments);

for (int i = handles.GetLowerBound(); i <= handles.GetUpperBound(); i++)
{
    _variant_t vString;
    vString.ChangeType(VT_BSTR, &_variant_t(handles[i]));
    _tprintf(_T("handleArray(%d): %s\n"), i, CW2T((_bstr_t)vString));
}

_tprintf(_T("Processing monitored item changed events for 10
seconds...\n"));
Sleep(10*1000);

_tprintf(_T("Unsubscribing...\n"));
ClientPtr->UnsubscribeAllMonitoredItems();

_tprintf(_T("Waiting for 5 seconds...\n"));
Sleep(5*1000);

// Unhook events
pClientEvents->DispEventUnadvise(ClientPtr, &pClientEvents->m_iid);
}
// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}
}
}

```

## PHP

---

```

// This example shows how to subscribe to changes of multiple monitored items and
// display the value of the monitored item with
// each change.

class ClientEvents {
    function DataChangeNotification($Sender, $E)
    {
        // Display the data
        // Remark: Production code would check e.Exception before accessing
        e.AttributeData.
        printf("%s: %s\n", $E->Arguments->NodeDescriptor, $E->AttributeData);
    }
}

// Instantiate the client object and hook events
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");
$ClientEvents = new ClientEvents();
com_event_sink($Client, $ClientEvents, "DEasyUAClientEvents");

printf("Subscribing...\n");
$MonitoringParameters = new COM("OpcLabs.EasyOpc.UA.UAMonitoringParameters");
$MonitoringParameters->SamplingInterval = 1000;
$MonitoredItemArguments1 = new
COM("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments");

```

```
$MonitoredItemArguments1->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$MonitoredItemArguments1->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845";
$MonitoredItemArguments1->MonitoringParameters = $MonitoringParameters;
$MonitoredItemArguments2 = new
COM("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments");
$MonitoredItemArguments2->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$MonitoredItemArguments2->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853";
$MonitoredItemArguments2->MonitoringParameters = $MonitoringParameters;
$MonitoredItemArguments3 = new
COM("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments");
$MonitoredItemArguments3->EndpointDescriptor->UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer";
$MonitoredItemArguments3->NodeDescriptor->NodeId->ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855";
$MonitoredItemArguments3->MonitoringParameters = $MonitoringParameters;
$arguments[0] = $MonitoredItemArguments1;
$arguments[1] = $MonitoredItemArguments2;
$arguments[2] = $MonitoredItemArguments3;
$handleArray = $Client->SubscribeMultipleMonitoredItems($arguments);

for ($i = 0; $i < count($handleArray); $i++)
{
    sprintf("handleArray[%d]: %d", $i, $handleArray[$i]);
}

printf("Processing monitored item changed events for 10 seconds...\n");
$startTime = time(); do { com_message_pump(1000); } while (time() < $startTime +
10);

printf("Unsubscribing...\n");
$Client->UnsubscribeAllMonitoredItems;

printf("Waiting for 5 seconds...\n");
$startTime = time(); do { com_message_pump(1000); } while (time() < $startTime + 5);
```

## Visual Basic (VB 6.)

---

Rem This example shows how to subscribe to changes of multiple monitored items and display the value of the monitored item with  
Rem each change.

```
' The client object, with events
'Public WithEvents Client2 As EasyUAClient

Private Sub SubscribeMultipleMonitoredItems_Main_Command_Click()
    OutputText = ""

    Set Client2 = New EasyUAClient

    OutputText = OutputText & "Subscribing..." & vbCrLf
    Dim MonitoringParameters As New UAMonitoringParameters
    MonitoringParameters.SamplingInterval = 1000

    Dim MonitoredItemArguments1 As New EasyUAMonitoredItemArguments
    MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    MonitoredItemArguments1.NodeDescriptor.NodeId.expandedText =
```

```
"nsu=http://test.org/UA/Data/;i=10845"
    Set MonitoredItemArguments1.MonitoringParameters = MonitoringParameters
    MonitoredItemArguments1.SetState ("Item1")

    Dim MonitoredItemArguments2 As New EasyUAMonitoredItemArguments
    MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    MonitoredItemArguments2.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10853"
    Set MonitoredItemArguments2.MonitoringParameters = MonitoringParameters
    MonitoredItemArguments2.SetState ("Item2")

    Dim MonitoredItemArguments3 As New EasyUAMonitoredItemArguments
    MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    MonitoredItemArguments3.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10855"
    Set MonitoredItemArguments3.MonitoringParameters = MonitoringParameters
    MonitoredItemArguments3.SetState ("Item3")

    Dim arguments(2) As Variant
    Set arguments(0) = MonitoredItemArguments1
    Set arguments(1) = MonitoredItemArguments2
    Set arguments(2) = MonitoredItemArguments3
    Dim handleArray As Variant
    handleArray = Client2.SubscribeMultipleMonitoredItems(arguments)

    Dim i As Long: For i = LBound(handleArray) To UBound(handleArray)
        OutputText = OutputText & "handleArray(" & i & ")：" & handleArray(i) &
vbCrLf
    Next

    OutputText = OutputText & "Processing monitored item changed events for 10
seconds..." & vbCrLf
    Pause 10000

    OutputText = OutputText & "Unsubscribing..." & vbCrLf
    Call Client2.UnsubscribeAllMonitoredItems

    OutputText = OutputText & "Waiting for 5 seconds..." & vbCrLf
    Pause 5000

    Set Client2 = Nothing
    OutputText = OutputText & "Done." & vbCrLf
End Sub

Private Sub Client2_DataChangeNotification(ByVal sender As Variant, ByVal eventArgs
As EasyUADataChangeNotificationEventArgs)
    ' Display the data
    ' Remark: Production code would check eventArgs.Exception before accessing
eventArgs.AttributeData.
    OutputText = OutputText & "[" & eventArgs.arguments.State & "] " &
eventArgs.arguments.NodeDescriptor & ":" & eventArgs.AttributeData & vbCrLf
End Sub
```

## VBScript

Rem This example shows how to subscribe to changes of multiple monitored items and  
display the value of the monitored item with  
Rem each change.

## Option Explicit

```
' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "ClientClient_"

WScript.Echo "Subscribing..."
Dim MonitoringParameters: Set MonitoringParameters =
CreateObject("OpcLabs.EasyOpc.UA.UAMonitoringParameters")
MonitoringParameters.SamplingInterval = 1000
Dim MonitoredItemArguments1: Set MonitoredItemArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"
MonitoredItemArguments1.MonitoringParameters = MonitoringParameters
Dim MonitoredItemArguments2: Set MonitoredItemArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"
MonitoredItemArguments2.MonitoringParameters = MonitoringParameters
Dim MonitoredItemArguments3: Set MonitoredItemArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"
MonitoredItemArguments3.MonitoringParameters = MonitoringParameters
Dim arguments(2)
Set arguments(0) = MonitoredItemArguments1
Set arguments(1) = MonitoredItemArguments2
Set arguments(2) = MonitoredItemArguments3
Dim handleArray: handleArray = Client.SubscribeMultipleMonitoredItems(arguments)

Dim i: For i = LBound(handleArray) To UBound(handleArray)
    WScript.Echo "handleArray(" & i & "): " & handleArray(i)
Next

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllMonitoredItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

Sub Client_DataChangeNotification(Sender, e)
    ' Display the data
    Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else
    display = e.ErrorMessageBrief
    WScript.Echo e.Arguments.NodeDescriptor & ":" & display
End Sub
```

## Common considerations

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 might be optimized (merged together), depending on other parameters and circumstances.

## 6.1.4.3 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 (OPC Classic) or sampling interval (OPC UA).

### A single subscription

Instead of unsubscribing and subscribing again, change the parameters by calling the [ChangeItemSubscription](#) (OPC Classic) or [ChangeDataChangeSubscription](#) or [ChangeMonitoredItemSubscription](#) (OPC UA) method, passing it the subscription handle, and the new parameters in form of [DAGroupParameters](#) object (in QuickOPC.NET), [EasyUASubscriptionChangeArguments](#) (in QuickOPC-UA) or individually a requested update rate or sampling interval, and optionally a deadband value (this is the only way in QuickOPC-COM).

Examples for OPC Classic:

#### C#

```
// This example shows how change the update rate of an existing subscription.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.DataAccess;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyDAClient
    {

        partial class ChangeItemSubscription
        {
            public static void Main()
            {
                using (var easyDAClient = new EasyDAClient())
                {
                    easyDAClient.ItemChanged += easyDAClient_ItemChanged;

                    Console.WriteLine("Subscribing...");
                    int handle = easyDAClient.SubscribeItem("", "OPCLabs.KitServer.2",
"Simulation.Random", 1000);

                    Console.WriteLine("Waiting for 10 seconds...");
                    Thread.Sleep(10 * 1000);

                    Console.WriteLine("Changing subscription...");
                    easyDAClient.ChangeItemSubscription(handle, new
DAGroupParameters(100));

                    Console.WriteLine("Waiting for 10 seconds...");
                    Thread.Sleep(10 * 1000);

                    Console.WriteLine("Unsubscribing...");
                    easyDAClient.UnsubscribeAllItems();

                    Console.WriteLine("Waiting for 10 seconds...");
                }
            }

            private static void easyDAClient_ItemChanged(object sender, ItemChangedEventArgs e)
            {
                Console.WriteLine("Item changed: " + e.ItemName);
            }
        }
    }
}
```

```
        Thread.Sleep(10 * 1000);
    }
}

// Item changed event handler
static void easyDAClient_ItemChanged([NotNull] object sender, [NotNull]
EasyDAIItemChangedEventArgs e)
{
    Console.WriteLine(e.Vtq);
}
}
```

VB.NET

```

' This example shows how change the update rate of an existing subscription.

Imports JetBrains.Annotations
Imports OpcLabs.EasyOpc.DataAccess
Imports System.Threading

Namespace _EasyDAClient

    Partial Friend Class ChangeItemSubscription
        Public Shared Sub Main()
            Using easyDAClient = New EasyDAClient()
                AddHandler easyDAClient.ItemChanged, AddressOf easyDAClient_ItemChanged

                Console.WriteLine("Subscribing...")
                Dim handle As Integer = easyDAClient.SubscribeItem("", "OPCLabs.KitServer.2", "Simulation.Random", 1000)

                Console.WriteLine("Waiting for 10 seconds...")
                Thread.Sleep(10 * 1000)

                Console.WriteLine("Changing subscription...")
                easyDAClient.ChangeItemSubscription(handle, New DAGroupParameters(100))

                Console.WriteLine("Waiting for 10 seconds...")
                Thread.Sleep(10 * 1000)

                Console.WriteLine("Unsubscribing...")
                easyDAClient.UnsubscribeAllItems()

                Console.WriteLine("Waiting for 10 seconds...")
                Thread.Sleep(10 * 1000)
            End Using
        End Sub

        ' Item changed event handler
        Private Shared Sub easyDAClient_ItemChanged(<NotNull()> ByVal sender As Object, <NotNull()> ByVal e As EasyDAItemChangedEventArgs)
            Console.WriteLine(e.Vtq)
        End Sub
    End Class
End Namespace

```

# VBS

Rem This example shows how change the update rate of an existing subscription.

## Option Explicit

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handle: handle = Client.SubscribeItem("", "OPCLabs.KitServer.2",
"Simulation.Random", 1000)

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Changing subscription..."
Client.ChangeItemSubscription handle, 100

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllItems

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.DisconnectObject Client
Set Client = Nothing

Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Vtq
End Sub
```

Examples for OPC UA:

## C#

```
// This example shows how to change the sampling rate of an existing monitored item
subscription.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class ChangeMonitoredItemSubscription
        {
            public static void Overload1()
            {
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                int handle = easyUAClient.SubscribeDataChange(
                    "http://opcua.demo-this.com:51211/UA/SampleServer", // or
"opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                    "nsu=http://test.org/UA/Data/;i=10853",
                    1000);

                Console.WriteLine("Processing monitored item changed events for 10
seconds...");
```

```
                System.Threading.Thread.Sleep(10 * 1000);
```

```
        Console.WriteLine("Changing subscription...");  
        easyUAClient.ChangeMonitoredItemSubscription(handle, 100);  
  
        Console.WriteLine("Processing monitored item changed events for 10  
seconds...");  
        System.Threading.Thread.Sleep(10 * 1000);  
  
        Console.WriteLine("Unsubscribing...");  
        easyUAClient.UnsubscribeAllMonitoredItems();  
  
        Console.WriteLine("Waiting for 5 seconds...");  
        System.Threading.Thread.Sleep(5 * 1000);  
    }  
  
    static void easyUAClient_DataChangeNotification(object sender,  
EasyUADataChangeEventArgs e)  
    {  
        // Remark: Production code would check e.Exception before accessing  
e.AttributeData.  
        Console.WriteLine(e.AttributeData.Value);  
    }  
}  
}  
}
```

## VB.NET

---

```
' This example shows how to change the sampling rate of an existing monitored item  
subscription.  
Imports OpcLabs.EasyOpc.UA  
Imports OpcLabs.EasyOpc.UA.OperationModel  
  
Namespace _EasyUAClient  
    Friend Class ChangeMonitoredItemSubscription  
        Public Shared Sub Overload1()  
            Dim easyUAClient = New EasyUAClient()  
            AddHandler easyUAClient.DataChangeNotification, AddressOf  
easyUAClient_DataChangeNotification  
  
            Console.WriteLine("Subscribing...")  
            Dim handle As Integer = easyUAClient.SubscribeDataChange("http://opcua.demo-  
this.com:51211/UA/SampleServer", _  
"nsu=http://test.org/UA/Data/;i=10853", 1000) ' or "opc.tcp://opcua.demo-  
this.com:51210/UA/SampleServer"  
  
            Console.WriteLine("Processing monitored item changed events for 10  
seconds...")  
            Threading.Thread.Sleep(10 * 1000)  
  
            Console.WriteLine("Changing subscription...")  
            easyUAClient.ChangeMonitoredItemSubscription(handle, 100)  
  
            Console.WriteLine("Processing monitored item changed events for 10  
seconds...")  
            Threading.Thread.Sleep(10 * 1000)  
  
            Console.WriteLine("Unsubscribing...")  
            easyUAClient.UnsubscribeAllMonitoredItems()  
  
            Console.WriteLine("Waiting for 5 seconds...")  
            Threading.Thread.Sleep(5 * 1000)  
        End Sub  
    End Class  
End Namespace
```

```
Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As Object,
 ByVal e As EasyUADataChangeNotificationEventArgs)
    ' Remark: Production code would check e.Exception before accessing
    e.AttributeData.
    Console.WriteLine(e.AttributeData.Value)
End Sub
End Class
End Namespace
```

## VBScript

Rem This example shows how to change the sampling rate of an existing monitored item subscription.

```
Option Explicit
```

```
' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handle: handle = Client.SubscribeDataChange(
    "http://opcua.demo-this.com:51211/UA/SampleServer", _
    "nsu=http://test.org/UA/Data/;i=10853", _
    1000)

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Changing subscription..."
Client.ChangeMonitoredItemSubscription handle, 100

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllMonitoredItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000
```

```
Sub Client_DataChangeNotification(Sender, e)
    Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else display =
    e.ErrorMessageBrief
    WScript.Echo display
End Sub
```

## Multiple subscriptions

For changing parameters of multiple subscriptions in an efficient manner, call the [ChangeMultipleItemSubscriptions](#) (OPC Classic), or [ChangeMultipleDataChangeSubscriptions](#) or [ChangeMultipleMonitoredItemSubscriptions](#) (OPC UA) method.

Examples for OPC Classic:

## VBScript

Rem This example shows how change the update rate of multiple existing subscriptions.

```
Option Explicit
```

```
Dim ItemSubscriptionArguments1: Set ItemSubscriptionArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments1.ItemDescriptor.ItemID = "Simulation.Random"
ItemSubscriptionArguments1.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments2: Set ItemSubscriptionArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments2.ItemDescriptor.ItemID = "Trends.Ramp (1 min)"
ItemSubscriptionArguments2.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments3: Set ItemSubscriptionArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments3.ItemDescriptor.ItemID = "Trends.Sine (1 min)"
ItemSubscriptionArguments3.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments4: Set ItemSubscriptionArguments4 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments4.ItemDescriptor.ItemID = "Simulation.Register_I4"
ItemSubscriptionArguments4.GroupParameters.RequestedUpdateRate = 1000

Dim arguments(3)
Set arguments(0) = ItemSubscriptionArguments1
Set arguments(1) = ItemSubscriptionArguments2
Set arguments(2) = ItemSubscriptionArguments3
Set arguments(3) = ItemSubscriptionArguments4

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handleArray: handleArray = Client.SubscribeMultipleItems(arguments)

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Changing subscriptions..."

Dim HandleGroupArguments1: Set HandleGroupArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAHandleGroupArguments")
HandleGroupArguments1.Handle = handleArray(0)
HandleGroupArguments1.GroupParameters.RequestedUpdateRate = 100

Dim HandleGroupArguments2: Set HandleGroupArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.DAHandleGroupArguments")
HandleGroupArguments2.Handle = handleArray(1)
HandleGroupArguments2.GroupParameters.RequestedUpdateRate = 100

Dim subscriptionChangeArguments(1)
Set subscriptionChangeArguments(0) = HandleGroupArguments1
Set subscriptionChangeArguments(1) = HandleGroupArguments2

Client.ChangeMultipleItemSubscriptions subscriptionChangeArguments

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000
```

```
WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllItems

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.DisconnectObject Client
Set Client = Nothing

Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Vtq
End Sub
```

Examples for OPC UA:

## C#

```
// This example shows how change the sampling rate of multiple existing monitored item
subscriptions.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class ChangeMultipleMonitoredItemSubscriptions
        {
            public static void Overload2()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                int[] handleArray = easyUAClient.SubscribeMultipleMonitoredItems(new []
                {
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10845", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10853", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10855", 1000)
                });
                Console.WriteLine("Processing monitored item changed events for 10
seconds...");
                System.Threading.Thread.Sleep(10 * 1000);

                Console.WriteLine("Changing subscriptions...");
                easyUAClient.ChangeMultipleMonitoredItemSubscriptions(handleArray, 100);

                Console.WriteLine("Processing monitored item changed events for 10
seconds...");
                System.Threading.Thread.Sleep(10 * 1000);

                Console.WriteLine("Unsubscribing...");
```

```
        easyUAClient.UnsubscribeAllMonitoredItems();

        Console.WriteLine("Waiting for 5 seconds...");
        System.Threading.Thread.Sleep(5 * 1000);
    }

    static void easyUAClient_DataChangeNotification(object sender,
EasyUADataChangeNotificationEventArgs e)
{
    // Display value
    // Remark: Production code would check e.Exception before accessing
e.AttributeData.
    Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,
e.AttributeData.Value);
}
}
}
}
```

## VB.NET

```
' This example shows how change the sampling rate of multiple existing monitored item
subscriptions.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class ChangeMultipleMonitoredItemSubscriptions
        Public Shared Sub Overload2()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            Dim handleArray() As Integer =
easyUAClient.SubscribeMultipleMonitoredItems(New EasyUAMonitoredItemArguments() _
{
    New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845", 1000),
    New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000),
    New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855", 1000)
})

            Console.WriteLine("Processing monitored item changed events for 10
seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Changing subscriptions...")
            easyUAClient.ChangeMultipleMonitoredItemSubscriptions(handleArray, 100)

            Console.WriteLine("Processing monitored item changed events for 10
seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Unsubscribing...")
            easyUAClient.UnsubscribeAllMonitoredItems()

            Console.WriteLine("Waiting for 5 seconds...")
            Threading.Thread.Sleep(5 * 1000)
        End Sub
    End Class
End Namespace
```

```
    Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As Object,
 ByVal e As EasyUADataChangeNotificationEventArgs)
        ' Display value
        ' Remark: Production code would check e.Exception before accessing
        e.AttributeData.
        Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,
 e.AttributeData.Value)
    End Sub
End Class
End Namespace
```

## VBScript

Rem This example shows how change the sampling rate of multiple existing monitored item subscriptions.

```
' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim OldMonitoringParameters: Set OldMonitoringParameters =
CreateObject("OpcLabs.EasyOpc.UA.UAMonitoringParameters")
OldMonitoringParameters.SamplingInterval = 1000
Dim MonitoredItemArguments1: Set MonitoredItemArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"
MonitoredItemArguments1.MonitoringParameters = OldMonitoringParameters
Dim MonitoredItemArguments2: Set MonitoredItemArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"
MonitoredItemArguments2.MonitoringParameters = OldMonitoringParameters
Dim MonitoredItemArguments3: Set MonitoredItemArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"
MonitoredItemArguments3.MonitoringParameters = OldMonitoringParameters
Dim arguments(2)
Set arguments(0) = MonitoredItemArguments1
Set arguments(1) = MonitoredItemArguments2
Set arguments(2) = MonitoredItemArguments3
Dim handleArray: handleArray = Client.SubscribeMultipleMonitoredItems(arguments)

Dim i: For i = LBound(handleArray) To UBound(handleArray)
    WScript.Echo "handleArray(" & i & "): " & handleArray(i)
Next

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Changing subscriptions..."
Dim NewMonitoringParameters: Set NewMonitoringParameters =
CreateObject("OpcLabs.EasyOpc.UA.UAMonitoringParameters")
NewMonitoringParameters.SamplingInterval = 100
Dim SubscriptionChangeArguments1: Set SubscriptionChangeArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUASubscriptionChangeArguments")
```

```
SubscriptionChangeArguments1.Handle = handleArray(0)
Set SubscriptionChangeArguments1.MonitoringParameters = NewMonitoringParameters
Dim SubscriptionChangeArguments2: Set SubscriptionChangeArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUASubscriptionChangeArguments")
SubscriptionChangeArguments2.Handle = handleArray(1)
Set SubscriptionChangeArguments2.MonitoringParameters = NewMonitoringParameters
Dim SubscriptionChangeArguments3: Set SubscriptionChangeArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUASubscriptionChangeArguments")
SubscriptionChangeArguments3.Handle = handleArray(2)
Set SubscriptionChangeArguments3.MonitoringParameters = NewMonitoringParameters
Dim subscriptionChangeArguments(2)
Set subscriptionChangeArguments(0) = SubscriptionChangeArguments1
Set subscriptionChangeArguments(1) = SubscriptionChangeArguments2
Set subscriptionChangeArguments(2) = SubscriptionChangeArguments3
Client.ChangeMultipleMonitoredItemSubscriptions subscriptionChangeArguments

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllMonitoredItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

Sub Client_DataChangeNotification(Sender, e)
    ' Display the data
    Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else display =
e.ErrorMessageBrief
    WScript.Echo e.Arguments.NodeDescriptor & ":" & display
End Sub
```

## 6.1.4.4 Unsubscribing

### From a single item

If you no longer want to receive item change notifications, you need to unsubscribe from them. In QuickOPC.NET (OPC Classic), to unsubscribe from a single OPC item, call the [UnsubscribeItem](#) method, passing it the subscription handle. In QuickOPC-UA, to unsubscribe from a single monitored item, call the [UnsubscribeMonitoredItem](#) method, passing it the subscription handle as well.

Examples for OPC Classic:

#### C#

```
// This example shows how subscribe to changes of multiple items, and unsubscribe from
one of them.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.DataAccess;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        class UnsubscribeItem
        {
```

```

public static void Main()
{
    using (var easyDAClient = new EasyDAClient())
    {
        easyDAClient.ItemChanged += easyDAClient_ItemChanged;

        int[] handleArray = easyDAClient.SubscribeMultipleItems(
            new[] {
                new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Simulation.Random", 1000, null),
                new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Trends.Ramp (1 min)", 1000, null),
                new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Trends.Sine (1 min)", 1000, null),
                new DAIItemGroupArguments("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 1000, null)
            });
    }

    Console.WriteLine("Processing item changed events for 30
seconds...");
    Thread.Sleep(30 * 1000);

    Console.WriteLine("Unsubscribing from the first item...");
    easyDAClient.UnsubscribeItem(handleArray[0]);

    Console.WriteLine();

    Console.WriteLine("Processing item changed events for 30
seconds...");
    Thread.Sleep(30 * 1000);
}

// Item changed event handler
static void easyDAClient_ItemChanged([NotNull] object sender, [NotNull]
EasyDAIChangedEventArgs e)
{
    Console.WriteLine("{0}: {1}", e.Arguments.ItemDescriptor.ItemId, e.Vtq);
}
}
}

```

# VBS

Rem This example shows how unsubscribe from changes of a single item.

## Option Explicit

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handle: handle = Client.SubscribeItem("", "OPCLabs.KitServer.2",
"Simulation.Random", 1000)

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeItem handle

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000
```

```
Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Vtq
End Sub
```

Examples for OPC UA:

## C#

```
// This example shows how unsubscribe from changes of a single monitored item.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class UnsubscribeMonitoredItem
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                int handle = easyUAClient.SubscribeDataChange(
                    "http://opcua.demo-this.com:51211/UA/SampleServer",    // or
"opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                    "nsu=http://test.org/UA/Data/;i=10853",
                    1000);

                Console.WriteLine("Processing monitored item changed events for 10
seconds...");
                System.Threading.Thread.Sleep(10 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeMonitoredItem(handle);

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }

            static void easyUAClient_DataChangeNotification(object sender,
EasyUADataChangeNotificationEventArgs e)
            {
                // Display value
                // Remark: Production code would check e.Exception before accessing
e.AttributeData.
                Console.WriteLine(e.AttributeData.Value);
            }
        }
    }
}
```

## VB.NET

```
' This example shows how unsubscribe from changes of a single monitored item.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel
```

```
Namespace _EasyUAClient
    Friend Class UnsubscribeMonitoredItem
        Public Shared Sub Main()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            Dim handle As Integer = easyUAClient.SubscribeDataChange("http://opcua.demo-
this.com:51211/UA/SampleServer", _

"nsu=http://test.org/UA/Data/;i=10853", 1000) ' or "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"

            Console.WriteLine("Processing monitored item changed events for 10
seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Unsubscribing...")
            easyUAClient.UnsubscribeMonitoredItem(handle)

            Console.WriteLine("Waiting for 5 seconds...")
            Threading.Thread.Sleep(5 * 1000)
        End Sub

        Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As Object,
 ByVal e As EasyUADataChangeNotificationEventArgs)
            ' Display value
            ' Remark: Production code would check e.Exception before accessing
e.AttributeData.
            Console.WriteLine(e.AttributeData.Value)
        End Sub
    End Class
End Namespace
```

## VBScript

```
Rem This example shows how unsubscribe from changes of a single monitored item.

Option Explicit

' Instantiate the client object and hook events
Dim Client: Set Client= CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handle: handle = Client.SubscribeDataChange("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000)

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeMonitoredItem handle

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

Sub Client_DataChangeNotification(Sender, e)
    ' Display the data
End Sub
```

```
Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else display =
e.ErrorMessageBrief
WScript.Echo display
End Sub
```

## From multiple items

In QuickOPC.NET (OPC Classic), 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. In QuickOPC-UA, to unsubscribe from multiple monitored items in an efficient manner, call the [UnsubscribeMultipleMonitoredItems](#) method (instead of calling [UnsubscribeMonitoredItem](#) in a loop), passing it an array of subscription handles.

Examples for OPC Classic;

### VBScript

```
Rem This example shows how to unsubscribe from changes of multiple items.

Option Explicit

Dim ItemSubscriptionArguments1: Set ItemSubscriptionArguments1 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments1.ItemDescriptor.ItemID = "Simulation.Random"
ItemSubscriptionArguments1.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments2: Set ItemSubscriptionArguments2 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments2.ItemDescriptor.ItemID = "Trends.Ramp (1 min)"
ItemSubscriptionArguments2.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments3: Set ItemSubscriptionArguments3 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments3.ItemDescriptor.ItemID = "Trends.Sine (1 min)"
ItemSubscriptionArguments3.GroupParameters.RequestedUpdateRate = 1000

Dim ItemSubscriptionArguments4: Set ItemSubscriptionArguments4 =
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")

ItemSubscriptionArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"
ItemSubscriptionArguments4.ItemDescriptor.ItemID = "Simulation.Register_I4"
ItemSubscriptionArguments4.GroupParameters.RequestedUpdateRate = 1000

Dim arguments(3)
Set arguments(0) = ItemSubscriptionArguments1
Set arguments(1) = ItemSubscriptionArguments2
Set arguments(2) = ItemSubscriptionArguments3
Set arguments(3) = ItemSubscriptionArguments4

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handleArray: handleArray = Client.SubscribeMultipleItems(arguments)

Dim i: For i = LBound(handleArray) To UBound(handleArray)
```

```
WScript.Echo "handleArray(" & i & ")：" & handleArray(i)
Next

WScript.Echo "Processing item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing from two items..."
Dim handles1(1)
handles1(0) = handleArray(1)
handles1(1) = handleArray(2)
Client.UnsubscribeMultipleItems handles1

WScript.Echo "Processing item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing from all remaining items..."
Client.UnsubscribeAllItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Arguments.ItemDescriptor.ItemId & ":" & e.Vtq
End Sub
```

Examples for OPC UA:

## C#

```
// This example shows how to unsubscribe from changes of multiple items.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class UnsubscribeMultipleMonitoredItems
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                int[] handleArray = easyUAClient.SubscribeMultipleMonitoredItems(new[]
                {
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10845", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10853", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10855", 1000)
                });
                Console.WriteLine("Processing monitored item changed events for 10
```

```
seconds...");  
        System.Threading.Thread.Sleep(10 * 1000);  
  
        Console.WriteLine("Unsubscribing...");  
        easyUAClient.UnsubscribeMultipleMonitoredItems(handleArray);  
  
        Console.WriteLine("Waiting for 5 seconds...");  
        System.Threading.Thread.Sleep(5 * 1000);  
    }  
  
    static void easyUAClient_DataChangeNotification(object sender,  
EasyUADeclareDataChangeNotificationEventArgs e)  
    {  
        // Display value  
        // Remark: Production code would check e.Exception before accessing  
e.AttributeData.  
        Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,  
e.AttributeData.Value);  
    }  
}  
}  
}
```

# VB.NET

```

' This example shows how to unsubscribe from changes of multiple items.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class UnsubscribeMultipleMonitoredItems
        Public Shared Sub Main()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            Dim handleArray() As Integer =
easyUAClient.SubscribeMultipleMonitoredItems(New EasyUAMonitoredItemArguments() _
{
    {
        New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845", 1000),
        New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000),
        New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855", 1000)
    }
})
            Console.WriteLine("Processing monitored item changed events for 10
seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Unsubscribing...")
            easyUAClient.UnsubscribeMultipleMonitoredItems(handleArray)

            Console.WriteLine("Waiting for 5 seconds...")
            Threading.Thread.Sleep(5 * 1000)
        End Sub

        Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As Object,
ByVal e As EasyUADataChangeNotificationEventArgs)
            ' Display value
        End Sub
    End Class
End Namespace

```

```
' Remark: Production code would check e.Exception before accessing
e.AttributeData.
    Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,
e.AttributeData.Value)
End Sub
End Class
End Namespace
```

## VBScript

---

Rem This example shows how to unsubscribe from changes of multiple items.

```
Option Explicit
```

```
' Instantiate the client object and hook events
Dim Client: Set Client= CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim MonitoringParameters: Set MonitoringParameters =
CreateObject("OpcLabs.EasyOpc.UA.UAMonitoringParameters")
MonitoringParameters.SamplingInterval = 1000
Dim MonitoredItemArguments1: Set MonitoredItemArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"
MonitoredItemArguments1.MonitoringParameters = MonitoringParameters
Dim MonitoredItemArguments2: Set MonitoredItemArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"
MonitoredItemArguments2.MonitoringParameters = MonitoringParameters
Dim MonitoredItemArguments3: Set MonitoredItemArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"
MonitoredItemArguments3.MonitoringParameters = MonitoringParameters
Dim arguments(2)
Set arguments(0) = MonitoredItemArguments1
Set arguments(1) = MonitoredItemArguments2
Set arguments(2) = MonitoredItemArguments3
Dim handleArray: handleArray = Client.SubscribeMultipleMonitoredItems(arguments)

Dim i: For i = LBound(handleArray) To UBound(handleArray)
    WScript.Echo "handleArray(" & i & "): " & handleArray(i)
Next

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing from two monitored items..."
Dim handles1(1)
handles1(0) = handleArray(1)
handles1(1) = handleArray(2)
Client.UnsubscribeMultipleMonitoredItems handles1

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000
```

```
WScript.Echo "Unsubscribing from all remaining items..."  
Client.UnsubscribeAllMonitoredItems  
  
WScript.Echo "Waiting for 5 seconds..."  
WScript.Sleep 5 * 1000  
  
Sub Client_DataChangeNotification(Sender, e)  
    ' Display the data  
    Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else display =  
    e.ErrorMessageBrief  
    WScript.Echo e.Arguments.NodeDescriptor & ":" & display  
End Sub
```

## From all items

In QuickOPC Classic, you can also unsubscribe from all items you have previously subscribed to (on the same instance of [EasyDAClient](#) object) by calling the [UnsubscribeAllItems](#) method. In QuickOPC-UA, you can unsubscribe from all monitored items you have previously subscribed to (on the same instance of [EasyUAClient](#) object) by calling the [UnsubscribeAllMonitoredItems](#) method.

Examples for OPC Classic:

### VBScript

```
Rem This example shows how to unsubscribe from changes of all items.  
  
Option Explicit  
  
Dim ItemSubscriptionArguments1: Set ItemSubscriptionArguments1 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")  
  
ItemSubscriptionArguments1.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
ItemSubscriptionArguments1.ItemDescriptor.ItemID = "Simulation.Random"  
ItemSubscriptionArguments1.GroupParameters.RequestedUpdateRate = 1000  
  
Dim ItemSubscriptionArguments2: Set ItemSubscriptionArguments2 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")  
  
ItemSubscriptionArguments2.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
ItemSubscriptionArguments2.ItemDescriptor.ItemID = "Trends.Ramp (1 min)"  
ItemSubscriptionArguments2.GroupParameters.RequestedUpdateRate = 1000  
  
Dim ItemSubscriptionArguments3: Set ItemSubscriptionArguments3 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")  
  
ItemSubscriptionArguments3.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
ItemSubscriptionArguments3.ItemDescriptor.ItemID = "Trends.Sine (1 min)"  
ItemSubscriptionArguments3.GroupParameters.RequestedUpdateRate = 1000  
  
Dim ItemSubscriptionArguments4: Set ItemSubscriptionArguments4 =  
CreateObject("OpcLabs.EasyOpc.DataAccess.OperationModel.EasyDAItemSubscriptionArguments")  
  
ItemSubscriptionArguments4.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
ItemSubscriptionArguments4.ItemDescriptor.ItemID = "Simulation.Register_I4"  
ItemSubscriptionArguments4.GroupParameters.RequestedUpdateRate = 1000  
  
Dim arguments(3)  
Set arguments(0) = ItemSubscriptionArguments1  
Set arguments(1) = ItemSubscriptionArguments2
```

```
Set arguments(2) = ItemSubscriptionArguments3
Set arguments(3) = ItemSubscriptionArguments4

Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim handleArray: handleArray = Client.SubscribeMultipleItems(arguments)

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllItems

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

Sub Client_ItemChanged(Sender, e)
    WScript.Echo e.Arguments.ItemDescriptor.ItemId & ":" & e.Vtg
End Sub
```

Examples for OPC UA:

## C#

---

```
// This example shows how to unsubscribe from changes of all items.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class UnsubscribeAllMonitoredItems
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.DataChangeNotification +=
easyUAClient_DataChangeNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeMultipleMonitoredItems(new[]
                {
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10845", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10853", 1000),
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",
                        "nsu=http://test.org/UA/Data/;i=10855", 1000)
                });
                Console.WriteLine("Processing monitored item changed events for 10
seconds...");  
                System.Threading.Thread.Sleep(10 * 1000);
            }
        }
    }
}
```

```
Console.WriteLine("Unsubscribing...");
easyUAClient.UnsubscribeAllMonitoredItems();

Console.WriteLine("Waiting for 5 seconds...");
System.Threading.Thread.Sleep(5 * 1000);
}

static void easyUAClient_DataChangeNotification(object sender,
EasyUADataChangeNotificationEventArgs e)
{
    // Display value
    // Remark: Production code would check e.Exception before accessing
e.AttributeData.
    Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,
e.AttributeData.Value);
}
}
```

# VB.NET

```

' This example shows how to unsubscribe from changes of all items.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class UnsubscribeAllMonitoredItems
        Public Shared Sub Main()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            easyUAClient.SubscribeMultipleMonitoredItems(New
EasyUAMonitoredItemArguments() _
            { _
                New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845", 1000), _
                New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000), _
                New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855", 1000) _
            } _
        }

        Console.WriteLine("Processing monitored item changed events for 10
seconds...")
        Threading.Thread.Sleep(10 * 1000)

        Console.WriteLine("Unsubscribing...")
        easyUAClient.UnsubscribeAllMonitoredItems()

        Console.WriteLine("Waiting for 5 seconds...")
        Threading.Thread.Sleep(5 * 1000)
    End Sub

    Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As Object,
ByVal e As EasyUADataChangeNotificationEventArgs)
        ' Display value
        ' Remark: Production code would check e.Exception before accessing
e.AttributeData.
        Console.WriteLine("{0}: {1}", e.Arguments.NodeDescriptor,

```

```
e.AttributeData.Value)
End Sub
End Class
End Namespace
```

## VBScript

---

```
Rem This example shows how to unsubscribe from changes of all items.
```

```
Option Explicit
```

```
' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim MonitoringParameters: Set MonitoringParameters =
CreateObject("OpcLabs.EasyOpc.UA.UAMonitoringParameters")
MonitoringParameters.SamplingInterval = 1000
Dim MonitoredItemArguments1: Set MonitoredItemArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"
MonitoredItemArguments1.MonitoringParameters = MonitoringParameters
Dim MonitoredItemArguments2: Set MonitoredItemArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"
MonitoredItemArguments2.MonitoringParameters = MonitoringParameters
Dim MonitoredItemArguments3: Set MonitoredItemArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"
MonitoredItemArguments3.MonitoringParameters = MonitoringParameters
Dim arguments(2)
Set arguments(0) = MonitoredItemArguments1
Set arguments(1) = MonitoredItemArguments2
Set arguments(2) = MonitoredItemArguments3
Dim handleArray: handleArray = Client.SubscribeMultipleMonitoredItems(arguments)

Dim i: For i = LBound(handleArray) To UBound(handleArray)
    WScript.Echo "handleArray(" & i & "): " & handleArray(i)
Next

WScript.Echo "Processing monitored item changed events for 10 seconds..."
WScript.Sleep 10 * 1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllMonitoredItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

Sub Client_DataChangeNotification(Sender, e)
    ' Display the data
    Dim display: If e.Exception Is Nothing Then display = e.AttributeData Else display =
```

```
e.ErrorMessageBrief  
    WScript.Echo e.Arguments.NodeDescriptor & ":" & display  
End Sub
```

## Implicit and explicit unsubscribe

If you are no longer using the parent [EasyDAClient](#) or [EasyUAClient](#) object, you do not necessarily have to unsubscribe from the items, but it is highly recommended that you do so; if you don't, be aware of the possible consequences described below.

 In QuickOPC.NET and QuickOPC-UA, the subscriptions will otherwise be internally alive until the .NET CLR (garbage collector) decides to finalize and destroy the parent [EasyDAClient](#) or [EasyUAClient](#) object (if ever); you cannot, however, determine that moment. You can alternatively call the [Dispose](#) method of the [EasyDAClient](#) (or [EasyUAClient](#)) 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.

### 6.1.4.5 Obtaining Subscription Information

If you need to obtain information about a subscription made or modified earlier, you have two methods available in QuickOPC-UA.

Calling [GetMonitoredItemArguments](#) method with a handle obtained when the subscription was made will return an [EasyUAMonitoredItemArguments](#) object with all current parameters.

#### C#

```
// This example shows how to obtain parameters of certain monitored item  
subscription.  
using OpcLabs.EasyOpc.UA;  
using System;  
  
namespace UADocExamples  
{  
    namespace _EasyUAClient  
    {  
        class GetMonitoredItemArguments  
        {  
            public static void Main()  
            {  
                // Instantiate the client object and hook events  
                var easyUAClient = new EasyUAClient();  
                easyUAClient.MonitoredItemChanged +=  
easyUAClient_MonitoredItemChanged;  
  
                Console.WriteLine("Subscribing...");  
                int[] handleArray =  
easyUAClient.SubscribeMultipleMonitoredItems(new []  
                {  
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-  
this.com:51211/UA/SampleServer",  
                        "nsu=http://test.org/UA/Data/;i=10845", 1000),  
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-  
this.com:51211/UA/SampleServer",  
                        "nsu=http://test.org/UA/Data/;i=10853", 1000),  
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-  
this.com:51211/UA/SampleServer",  
                        "nsu=http://test.org/UA/Data/;i=10854", 1000)  
                }  
            }  
        }  
    }  
}
```

# VB.NET

```
' This example shows how to obtain parameters of certain monitored item
subscription.

Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class GetMonitoredItemArguments
        Public Shared Sub Main()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
            Dim handleArray() As Integer =
easyUAClient.SubscribeMultipleMonitoredItems(New EasyUAMonitoredItemArguments() _
{
    {
        New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845", 1000),
        New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000),
        New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855", 1000)
    }
})
Console.WriteLine("Getting monitored item arguments...")
End Sub
End Class
End Namespace
```

```
Dim monitoredItemArguments As EasyUAMonitoredItemArguments =
easyUAClient.GetMonitoredItemArguments(handleArray(2))
    Console.WriteLine("NodeDescriptor: {0}",
monitoredItemArguments.NodeDescriptor)
    Console.WriteLine("SamplingInterval: {0}",
monitoredItemArguments.MonitoringParameters.SamplingInterval)
    Console.WriteLine("PublishingInterval: {0}",
monitoredItemArguments.SubscriptionParameters.PublishingInterval)

    Console.WriteLine("Waiting for 5 seconds...")
    Threading.Thread.Sleep(5 * 1000)

    Console.WriteLine("Unsubscribing...")
    easyUAClient.UnsubscribeAllMonitoredItems()

    Console.WriteLine("Waiting for 5 seconds...")
    Threading.Thread.Sleep(5 * 1000)
End Sub

Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As
Object, ByVal e As EasyUADataChangeNotificationEventArgs)
    ' Your code would do the processing here
End Sub
End Class
End Namespace
```

## VBScript

Rem This example shows how to obtain parameters of certain monitored item subscription.

```
Option Explicit

' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."

Dim MonitoredItemArguments1: Set MonitoredItemArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"

Dim MonitoredItemArguments2: Set MonitoredItemArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"

Dim MonitoredItemArguments3: Set MonitoredItemArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"

Dim arguments(2)
```

```
Set arguments(0) = MonitoredItemArguments1
Set arguments(1) = MonitoredItemArguments2
Set arguments(2) = MonitoredItemArguments3

Dim handleArray: handleArray = Client.SubscribeMultipleMonitoredItems(arguments)

WScript.Echo "Getting monitored item arguments..."
Dim MonitoredItemArguments: Set MonitoredItemArguments =
Client.GetMonitoredItemArguments(handleArray(2))

WScript.Echo "NodeDescriptor: " & MonitoredItemArguments.NodeDescriptor
WScript.Echo "SamplingInterval: " &
MonitoredItemArguments.MonitoringParameters.SamplingInterval
WScript.Echo "PublishingInterval: " &
MonitoredItemArguments.SubscriptionParameters.PublishingInterval

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllMonitoredItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000
```

```
Sub Client_DataChangeNotification(Sender, e)
    ' Your code would do the processing here
End Sub
```

Calling `GetMonitoredItemArgumentsDictionary` obtains information about all monitored item subscriptions on the `EasyUAClient` object. It returns a dictionary of monitored item argument objects (as above). The dictionary is keyed by the monitored item subscription handles, and its values contain information describing each subscription.

## C#

---

```
// This example shows how to obtain dictionary of parameters of all monitored item
subscriptions.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class GetMonitoredItemArgumentsDictionary
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.MonitoredItemChanged +=

easyUAClient_MonitoredItemChanged;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeMultipleMonitoredItems(new[]
                {
                    new EasyUAMonitoredItemArguments(null, "http://opcua.demo-
this.com:51211/UA/SampleServer",

```

# VB.NET

```
' This example shows how to obtain dictionary of parameters of all monitored item
subscriptions.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class GetMonitoredItemArgumentsDictionary
        Public Shared Sub Main()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.DataChangeNotification, AddressOf
easyUAClient_DataChangeNotification

            Console.WriteLine("Subscribing...")
        End Sub
    End Class
End Namespace
```

```

        easyUAClient.SubscribeMultipleMonitoredItems(New
EasyUAMonitoredItemArguments() _
{
    _ New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10845", 1000), _
    _ New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853", 1000), _
    _ New EasyUAMonitoredItemArguments(Nothing, "http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10855", 1000) _
}
)

Console.WriteLine("Getting monitored item arguments dictionary...")
Dim monitoredItemArgumentsDictionary As
EasyUAMonitoredItemArgumentsDictionary = _
    easyUAClient.GetMonitoredItemArgumentsDictionary()

For Each monitoredItemArguments As EasyUAMonitoredItemArguments In
monitoredItemArgumentsDictionary.Values
    Console.WriteLine()
    Console.WriteLine("NodeDescriptor: {0}",
monitoredItemArguments.NodeDescriptor)
    Console.WriteLine("SamplingInterval: {0}",
monitoredItemArguments.MonitoringParameters.SamplingInterval)
    Console.WriteLine("PublishingInterval: {0}",
monitoredItemArguments.SubscriptionParameters.PublishingInterval)
    Next monitoredItemArguments

Console.WriteLine("Waiting for 5 seconds...")
Threading.Thread.Sleep(5 * 1000)

Console.WriteLine("Unsubscribing...")
easyUAClient.UnsubscribeAllMonitoredItems()

Console.WriteLine("Waiting for 5 seconds...")
Threading.Thread.Sleep(5 * 1000)
End Sub

Private Shared Sub easyUAClient_DataChangeNotification(ByVal sender As
Object, ByVal e As EasyUADataChangeNotificationEventArgs)
    ' Your code would do the processing here
End Sub
End Class
End Namespace

```

## VBScript

---

Rem This example shows how to obtain dictionary of parameters of all monitored item subscriptions.

```

Option Explicit

' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."

Dim MonitoredItemArguments1: Set MonitoredItemArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-

```

```
this.com:51211/UA/SampleServer"
MonitoredItemArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10845"

Dim MonitoredItemArguments2: Set MonitoredItemArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10853"

Dim MonitoredItemArguments3: Set MonitoredItemArguments3 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.EasyUAMonitoredItemArguments")
MonitoredItemArguments3.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
MonitoredItemArguments3.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10855"

Dim arguments(2)
Set arguments(0) = MonitoredItemArguments1
Set arguments(1) = MonitoredItemArguments2
Set arguments(2) = MonitoredItemArguments3

Dim handleArray: handleArray = Client.SubscribeMultipleMonitoredItems(arguments)

WScript.Echo "Getting monitored item arguments dictionary..."
Dim MonitoredItemArgumentsDictionary: Set MonitoredItemArgumentsDictionary =
Client.GetMonitoredItemArgumentsDictionary

Dim DictionaryEntry: For Each DictionaryEntry In MonitoredItemArgumentsDictionary
    Dim MonitoredItemArguments: Set MonitoredItemArguments = DictionaryEntry.Value
    WScript.Echo
    WScript.Echo "NodeDescriptor: " & MonitoredItemArguments.NodeDescriptor
    WScript.Echo "SamplingInterval: " &
    MonitoredItemArguments.MonitoringParameters.SamplingInterval
    WScript.Echo "PublishingInterval: " &
    MonitoredItemArguments.SubscriptionParameters.PublishingInterval
    Next

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllMonitoredItems

WScript.Echo "Waiting for 5 seconds..."
WScript.Sleep 5 * 1000

Sub Client_DataChangeNotification(Sender, e)
    ' Your code would do the processing here
End Sub
```

## 6.1.4.6 OPC Classic 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 inside the [Arguments](#) property.

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.

## 6.1.4.7 OPC UA Monitored Item Changed Event or Callback

When there is a significant change related to monitored item you have subscribed to, the [EasyUAClient](#) object generates a [DataChangeNotification](#) event. What constitutes a significant change is given by the data change filter specified when the subscription was created. For subscription mechanism to be useful, you should hook one or more event handlers to this event.

To be more precise, the [DataChangeNotification](#) event is actually generated in other cases, too.

First of all, you always receive at least one [DataChangeNotification](#) 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 [DataChangeNotification](#) event is generated every time the component loses connection to the monitored 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 [DataChangeNotification](#) notification if the status of the item changes (not just its actual data value). In fact, you can influence whether the timestamp or value changes trigger a notification, with the [Trigger](#) property in [UADataChangeFilter](#).

The [DataChangeNotification](#) event notification contains an [EasyUADataChangeNotificationEventArgs](#) 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 in [Arguments](#) (containing also information such as [SubscriptionParameters](#), [MonitoringParameters](#), [NodeDescriptor](#), [EndpointDescriptor](#) 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 [AttributeData](#)

property has the new value, timestamps and status of the item, in form of `UAAttributeData` object. If the `Exception` property is not a null reference, there has been an error related to the item, and the `AttributeData` property is not valid. In such case, the `Exception` property contains information about the problem.

The `DataChangeNotification` event handler is called on a thread determined by the `EasyUAClient` 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.

## 6.1.4.8 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.

Examples for OPC Classic:

### C#

```
// This example shows how subscribe to changes of multiple items and display the
// value of the item with each change,
// using a callback method specified using lambda expression.

using System.Diagnostics;
using OpcLabs.EasyOpc.DataAccess;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyDAClient
    {
        class SubscribeItem
        {
            public static void CallbackLambda()
            {
                // Instantiate the client object
                var easyDAClient = new EasyDAClient();

                Console.WriteLine("Subscribing...");
                // The callback is a lambda expression that displays the value
                easyDAClient.SubscribeItem("", "OPCLabs.KitServer.2",
                "Simulation.Random", 1000,
                (sender, eventArgs) =>
                {

```

```
        Debug.Assert(eventArgs != null);

        if (eventArgs.Exception != null)
            Console.WriteLine(eventArgs.Exception.ToString());
        else
        {
            Debug.Assert(eventArgs.Vtq != null);
            Console.WriteLine(eventArgs.Vtq.ToString());
        }
    });

Console.WriteLine("Processing item changed events for 10
seconds...");
Thread.Sleep(10 * 1000);

Console.WriteLine("Unsubscribing...");
easyDAClient.UnsubscribeAllItems();

Console.WriteLine("Waiting for 2 seconds...");
Thread.Sleep(2 * 1000);
}

}
}
```

## Examples for OPC UA:

C#

```
// This example shows how to subscribe to changes of a single monitored item, and
// display the value of the item with each change
// using a callback method that is provided as lambda expression.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class SubscribeDataChange
        {
            public static void CallbackLambda()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                Console.WriteLine("Subscribing...");
                // The callback is a lambda expression that displays the value
                easyUAClient.SubscribeDataChange(
                    "http://opcua.demo-this.com:51211/UA/SampleServer", // or
                    "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer"
                    "nsu=http://test.org/UA/Data/;i=10853",
                    1000,
                    (sender, eventArgs) =>
                Console.WriteLine(eventArgs.AttributeData.Value));
                // Remark: Production code would check eventArgs.Exception before
                accessing eventArgs.AttributeData.

                Console.WriteLine("Processing data change events for 10
seconds...");
```

```
        Console.WriteLine("Unsubscribing...");  
        easyUAClient.UnsubscribeAllMonitoredItems();  
  
        Console.WriteLine("Waiting for 2 seconds...");  
        System.Threading.Thread.Sleep(2 * 1000);  
    }  
}  
}  
}
```

## VB.NET

```
' This example shows how to subscribe to changes of a single monitored item, and  
display the value of the item with each change  
' using a callback method that is provided as lambda expression.  
Imports OpcLabs.EasyOpc.UA  
  
Namespace _EasyUAClient  
    Partial Friend Class SubscribeDataChange  
        Public Shared Sub CallbackLambda()  
            ' Instantiate the client object  
            Dim easyUAClient = New EasyUAClient()  
  
            Console.WriteLine("Subscribing...")  
            ' The callback is a lambda expression the displays the value  
            easyUAClient.SubscribeDataChange(_  
                "http://opcua.demo-this.com:51211/UA/SampleServer", _  
                "nsu=http://test.org/UA/Data/;i=10853", _  
                1000, _  
                Sub(sender, eventArgs)  
                    Console.WriteLine(eventArgs.AttributeData.Value))  
                    ' Remark: Production code would check eventArgs.Exception before  
accessing eventArgs.AttributeData.  
  
                    Console.WriteLine("Processing monitored item changed events for 10  
seconds...")  
                    Threading.Thread.Sleep(10 * 1000)  
  
                    Console.WriteLine("Unsubscribing...")  
                    easyUAClient.UnsubscribeAllMonitoredItems()  
  
                    Console.WriteLine("Waiting for 2 seconds...")  
                    Threading.Thread.Sleep(2 * 1000)  
                End Sub  
    End Class  
End Namespace
```

For subscription methods that work with multiple subscriptions at once, there is also a [Callback](#) (in OPC Classic) or [DataChangeCallback](#) (in OPC-UA) 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](#) (or [DataChangeCallback](#)) 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](#) or [EasyUAClient](#)) object, and set up event handlers on one instance and use the callback methods on the other instance.

## 6.1.5 Calling Methods in OPC-UA

In OPC-UA, Methods represent the function calls of Objects. Methods are called (invoked) and return after completion, whether successful or unsuccessful. Execution times for Methods may vary, depending on the function they are performing. A Method is always a component of an Object.

### A single method

If you want to call a method on a specific object node in OPC UA, use the `EasyUAClient.CallMethod` method, passing it the endpoint descriptor, object node ID, method node ID, and optionally the method input arguments (if the method has any). The input arguments are specified by an array of objects. The number of arguments and their types must conform to the method's requirements.

If the operation is successful, it returns an array of output arguments of the method call. The number of output arguments and their types are given by the UA method.

#### C++

```
// This example shows how to call a single method, and pass arguments to and from it.

#include "stdafx.h"
#include <atlsafe.h>
#include "_EasyUAClient.CallMethod.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void CallMethod::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            CComSafeArray<VARIANT> inputs(11);
```

```

        inputs.SetAt(0, _variant_t(false));
        inputs.SetAt(1, _variant_t(1));
        inputs.SetAt(2, _variant_t(2));
        inputs.SetAt(3, _variant_t(3));
        inputs.SetAt(4, _variant_t(4));
        inputs.SetAt(5, _variant_t(5));
        inputs.SetAt(6, _variant_t(6));
        inputs.SetAt(7, _variant_t(7));
        inputs.SetAt(8, _variant_t(8));
        inputs.SetAt(9, _variant_t(9));
        inputs.SetAt(10, _variant_t(10));

        CComSafeArray<VARIANT> typeCodes(11);
        typeCodes.SetAt(0, _variant_t(TypeCode_Boolean));
        typeCodes.SetAt(1, _variant_t(TypeCode_SByte));
        typeCodes.SetAt(2, _variant_t(TypeCode_Byte));
        typeCodes.SetAt(3, _variant_t(TypeCode_Int16));
        typeCodes.SetAt(4, _variant_t(TypeCode_UInt16));
        typeCodes.SetAt(5, _variant_t(TypeCode_Int32));
        typeCodes.SetAt(6, _variant_t(TypeCode_UInt32));
        typeCodes.SetAt(7, _variant_t(TypeCode_Int64));
        typeCodes.SetAt(8, _variant_t(TypeCode_UInt64));
        typeCodes.SetAt(9, _variant_t(TypeCode_Single));
        typeCodes.SetAt(10, _variant_t(TypeCode_Double));

        // Instantiate the client object
        _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

        // Perform the operation
        SAFEARRAY* pInputs = inputs;
        SAFEARRAY* pTypeCodes = typeCodes;;
        CComSafeArray<VARIANT> outputs;
        outputs.Attach(ClientPtr->CallMethod(
            L"http://opcua.demo-this.com:51211/UA/SampleServer",
            L"nsu=http://test.org/UA/Data/;i=10755",
            L"nsu=http://test.org/UA/Data/;i=10756",
            &pInputs,
            &pTypeCodes));

        // Display results
        for (int i = outputs.GetLowerBound(); i <= outputs.GetUpperBound(); i++)
        {
            _variant_t output(outputs[i]);
            _variant_t vString;
            vString.ChangeType(VT_BSTR, &output);
            _tprintf(_T("outputs(%d): %s\n"), i, CW2T(_bstr_t)vString));
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}

```

### Free Pascal

---

```

// This example shows how to call a single method, and pass arguments to and
// from it.

class procedure CallMethod.Main;
var
  Client: EasyUAClient;

```

```

I: Cardinal;
Inputs: OleVariant;
Outputs: OleVariant;
TypeCodes: OleVariant;
begin
  Inputs := VarArrayCreate([0, 10], varVariant);
  Inputs[0] := False;
  Inputs[1] := 1;
  Inputs[2] := 2;
  Inputs[3] := 3;
  Inputs[4] := 4;
  Inputs[5] := 5;
  Inputs[6] := 6;
  Inputs[7] := 7;
  Inputs[8] := 8;
  Inputs[9] := 9;
  Inputs[10] := 10;

  TypeCodes := VarArrayCreate([0, 10], varVariant);
  TypeCodes[0] := TypeCode_Boolean;
  TypeCodes[1] := TypeCode_SByte;
  TypeCodes[2] := TypeCode_Byte;
  TypeCodes[3] := TypeCode_Int16;
  TypeCodes[4] := TypeCode_UInt16;
  TypeCodes[5] := TypeCode_Int32;
  TypeCodes[6] := TypeCode_UInt32;
  TypeCodes[7] := TypeCode_Int64;
  TypeCodes[8] := TypeCode_UInt64;
  TypeCodes[9] := TypeCode_Single;
  TypeCodes[10] := TypeCode_Double;

  // Instantiate the client object
  Client := CoEasyUAClient.Create;

  // Perform the operation
  TVarData(Outputs).VType := varArray or varVariant;
  TVarData(Outputs).VArray := PVarArray(Client.CallMethod(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10755',
    'nsu=http://test.org/UA/Data/;i=10756',
    PSafeArray(TVarData(Inputs).VArray),
    PSafeArray(TVarData(TypeCodes).VArray)));

  // Display results
  for I := VarArrayLowBound(Outputs, 1) to VarArrayHighBound(Outputs, 1) do
    try
      WriteLn('outputs(', I, ') : ', Outputs[I]);
    except
      on EVariantError do WriteLn('*** Error displaying the value');
    end;
end;

```

## Object Pascal

---

```

// This example shows how to call a single method, and pass arguments to and
// from it.

class procedure CallMethod.Main;
var
  Client: TEasyUAClient;
  I: Cardinal;

```

```
Inputs: OleVariant;
Outputs: OleVariant;
TypeCodes: OleVariant;
begin
  Inputs := VarArrayCreate([0, 10], varVariant);
  Inputs[0] := False;
  Inputs[1] := 1;
  Inputs[2] := 2;
  Inputs[3] := 3;
  Inputs[4] := 4;
  Inputs[5] := 5;
  Inputs[6] := 6;
  Inputs[7] := 7;
  Inputs[8] := 8;
  Inputs[9] := 9;
  Inputs[10] := 10;

  TypeCodes := VarArrayCreate([0, 10], varVariant);
  TypeCodes[0] := TypeCode_Boolean;
  TypeCodes[1] := TypeCode_SByte;
  TypeCodes[2] := TypeCode_Byte;
  TypeCodes[3] := TypeCode_Int16;
  TypeCodes[4] := TypeCode_UInt16;
  TypeCodes[5] := TypeCode_Int32;
  TypeCodes[6] := TypeCode_UInt32;
  TypeCodes[7] := TypeCode_Int64;
  TypeCodes[8] := TypeCode_UInt64;
  TypeCodes[9] := TypeCode_Single;
  TypeCodes[10] := TypeCode_Double;

  // Instantiate the client object
  Client := TEasyUAClient.Create(nil);

  // Perform the operation
  TVarData(Outputs).VType := varArray or varVariant;
  TVarData(Outputs).VArray := PVarArray(Client.CallMethod(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10755',
    'nsu=http://test.org/UA/Data/;i=10756',
    PSafeArray(TVarData(Inputs).VArray),
    PSafeArray(TVarData(TypeCodes).VArray)));

  // Display results
  for I := VarArrayLowBound(Outputs, 1) to VarArrayHighBound(Outputs, 1) do
    WriteLn('outputs(', I, ') : ', Outputs[I]);
end;
```

## PHP

---

```
// This example shows how to call a single method, and pass arguments to and from it.

$inputs[0] = false;
$inputs[1] = 1;
$inputs[2] = 2;
$inputs[3] = 3;
$inputs[4] = 4;
$inputs[5] = 5;
$inputs[6] = 6;
$inputs[7] = 7;
$inputs[8] = 8;
```

```
$inputs[9] = 9;
$inputs[10] = 10;

$typeCodes[0] = 3;      // TypeCode.Boolean
$typeCodes[1] = 5;      // TypeCode.SByte
$typeCodes[2] = 6;      // TypeCode.Byte
$typeCodes[3] = 7;      // TypeCode.Int16
$typeCodes[4] = 8;      // TypeCode.UInt16
$typeCodes[5] = 9;      // TypeCode.Int32
$typeCodes[6] = 10;     // TypeCode.UInt32
$typeCodes[7] = 11;     // TypeCode.Int64
$typeCodes[8] = 12;     // TypeCode.UInt64
$typeCodes[9] = 13;     // TypeCode.Single
$typeCodes[10] = 14;    // TypeCode.Double

// Instantiate the client object
$Client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");

// Perform the operation
$outputs = $Client->CallMethod(
    "http://opcua.demo-this.com:51211/UA/SampleServer",
    "nsu=http://test.org/UA/Data/;i=10755",
    "nsu=http://test.org/UA/Data/;i=10756",
    $inputs,
    $typeCodes);

// Display results
for ($i = 0; $i < count($outputs); $i++)
{
    printf("outputs[%d]: %s\n", $i, $outputs[$i]);
}
```

## Visual Basic (VB 6.)

---

Rem This example shows how to call a single method, and pass arguments to and from it.

```
Private Sub CallMethod_Main_Command_Click()
    OutputText = ""

    Dim inputs(10)
    inputs(0) = False
    inputs(1) = 1
    inputs(2) = 2
    inputs(3) = 3
    inputs(4) = 4
    inputs(5) = 5
    inputs(6) = 6
    inputs(7) = 7
    inputs(8) = 8
    inputs(9) = 9
    inputs(10) = 10

    Dim typeCodes(10)
    typeCodes(0) = 3      ' TypeCode.Boolean
    typeCodes(1) = 5      ' TypeCode.SByte
    typeCodes(2) = 6      ' TypeCode.Byte
    typeCodes(3) = 7      ' TypeCode.Int16
    typeCodes(4) = 8      ' TypeCode.UInt16
    typeCodes(5) = 9      ' TypeCode.Int32
    typeCodes(6) = 10     ' TypeCode.UInt32
```

```
typeCodes(7) = 11    ' TypeCode.Int64
typeCodes(8) = 12    ' TypeCode.UInt64
typeCodes(9) = 13    ' TypeCode.Single
typeCodes(10) = 14   ' TypeCode.Double

' Instantiate the client object
Dim Client As New EasyUAClient

' Perform the operation
Dim outputs As Variant
outputs = Client.CallMethod(
    "http://opcua.demo-this.com:51211/UA/SampleServer", _
    "nsu=http://test.org/UA/Data/;i=10755", _
    "nsu=http://test.org/UA/Data/;i=10756", _
    inputs, _
    typeCodes)

' Display results
Dim i: For i = LBound(outputs) To UBound(outputs)
    On Error Resume Next
    OutputText = OutputText & "outputs(" & i & ")：" & outputs(i) & vbCrLf
    If Err <> 0 Then OutputText = OutputText & "*** Error" & vbCrLf ' occurs
with types not recognized by VB6
    On Error GoTo 0
Next
End Sub
```

## VBScript

Rem This example shows how to call a single method, and pass arguments to and from it.

```
Option Explicit
```

```
Dim inputs(10)
inputs(0) = False
inputs(1) = 1
inputs(2) = 2
inputs(3) = 3
inputs(4) = 4
inputs(5) = 5
inputs(6) = 6
inputs(7) = 7
inputs(8) = 8
inputs(9) = 9
inputs(10) = 10

Dim typeCodes(10)
typeCodes(0) = 3    ' TypeCode.Boolean
typeCodes(1) = 5    ' TypeCode.SByte
typeCodes(2) = 6    ' TypeCode.Byte
typeCodes(3) = 7    ' TypeCode.Int16
typeCodes(4) = 8    ' TypeCode.UInt16
typeCodes(5) = 9    ' TypeCode.Int32
typeCodes(6) = 10   ' TypeCode.UInt32
typeCodes(7) = 11   ' TypeCode.Int64
typeCodes(8) = 12   ' TypeCode.UInt64
typeCodes(9) = 13   ' TypeCode.Single
typeCodes(10) = 14  ' TypeCode.Double

' Instantiate the client object
```

```
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Perform the operation
Dim outputs: outputs = Client.CallMethod(
    "http://opcua.demo-this.com:51211/UA/SampleServer", _
    "nsu=http://test.org/UA/Data/;i=10755", _
    "nsu=http://test.org/UA/Data/;i=10756", _
    inputs, _
    typeCodes)

' Display results
Dim i: For i = LBound(outputs) To UBound(outputs)
    On Error Resume Next
    WScript.Echo "outputs(" & i & "): " & outputs(i)
    If Err <> 0 Then WScript.Echo "*** Error" ' occurs with types not recognized
by VBScript
    On Error Goto 0
Next
```

## Multiple methods; argument type conversion

For making multiple method calls (on the same or different object nodes) in an efficient manner, use the [EasyUAClient.CallMultipleMethods](#) method. You pass in an array of [UACallArguments](#) objects, each specifying the location of the object node, the method node, and the input arguments to be passed to the method.

If the originating tool or language cannot directly supply the argument in a type that is required for the method by the OPC UA server, you can have the value converted to the desired type internally by QuickOPC. The corresponding type information can be passed in some method overrides, or inside a [UACallArguments](#) object (using the [InputTypeCodes](#), [InputTypeFullName](#) or [InputTypes](#) property).

### C++

```
// This example shows how to call multiple methods, and pass arguments to and from them.

#include "stdafx.h"
#include <atlsafe.h>
#include "_EasyUAClient.CallMultipleMethods.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D" // mscorelible
using namespace mscorelible;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
```

```
rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClient
{
    void CallMultipleMethods::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {

            CComSafeArray<VARIANT> inputs1(11);
            inputs1.SetAt(0, _variant_t(false));
            inputs1.SetAt(1, _variant_t(1));
            inputs1.SetAt(2, _variant_t(2));
            inputs1.SetAt(3, _variant_t(3));
            inputs1.SetAt(4, _variant_t(4));
            inputs1.SetAt(5, _variant_t(5));
            inputs1.SetAt(6, _variant_t(6));
            inputs1.SetAt(7, _variant_t(7));
            inputs1.SetAt(8, _variant_t(8));
            inputs1.SetAt(9, _variant_t(9));
            inputs1.SetAt(10, _variant_t(10));

            CComSafeArray<VARIANT> typeCodes1(11);
            typeCodes1.SetAt(0, _variant_t(TypeCode_Boolean));
            typeCodes1.SetAt(1, _variant_t(TypeCode_SByte));
            typeCodes1.SetAt(2, _variant_t(TypeCode_Byte));
            typeCodes1.SetAt(3, _variant_t(TypeCode_Int16));
            typeCodes1.SetAt(4, _variant_t(TypeCode_UInt16));
            typeCodes1.SetAt(5, _variant_t(TypeCode_Int32));
            typeCodes1.SetAt(6, _variant_t(TypeCode_UInt32));
            typeCodes1.SetAt(7, _variant_t(TypeCode_Int64));
            typeCodes1.SetAt(8, _variant_t(TypeCode_UInt64));
            typeCodes1.SetAt(9, _variant_t(TypeCode_Single));
            typeCodes1.SetAt(10, _variant_t(TypeCode_Double));

            UACallArgumentsPtr CallArguments1Ptr(_uuidof(UACallArguments));
            CallArguments1Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            CallArguments1Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10755";
            CallArguments1Ptr->MethodNodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10756";
            CallArguments1Ptr->InputArguments = inputs1;
            CallArguments1Ptr->InputTypeCodes = typeCodes1;

            CComSafeArray<VARIANT> inputs2(12);
            inputs2.SetAt(0, _variant_t(false));
            inputs2.SetAt(1, _variant_t(1));
            inputs2.SetAt(2, _variant_t(2));
            inputs2.SetAt(3, _variant_t(3));
            inputs2.SetAt(4, _variant_t(4));
            inputs2.SetAt(5, _variant_t(5));
            inputs2.SetAt(6, _variant_t(6));
            inputs2.SetAt(7, _variant_t(7));
            inputs2.SetAt(8, _variant_t(8));
            inputs2.SetAt(9, _variant_t(9));
            inputs2.SetAt(10, _variant_t(10));
            inputs2.SetAt(11, _variant_t(L"eleven")));
    }
}
```

```

    CComSafeArray<VARIANT> typeCodes2(12);
    typeCodes2.SetAt(0, _variant_t(TypeCode_Boolean));
    typeCodes2.SetAt(1, _variant_t(TypeCode_SByte));
    typeCodes2.SetAt(2, _variant_t(TypeCode_Byte));
    typeCodes2.SetAt(3, _variant_t(TypeCode_Int16));
    typeCodes2.SetAt(4, _variant_t(TypeCode_UInt16));
    typeCodes2.SetAt(5, _variant_t(TypeCode_Int32));
    typeCodes2.SetAt(6, _variant_t(TypeCode_UInt32));
    typeCodes2.SetAt(7, _variant_t(TypeCode_Int64));
    typeCodes2.SetAt(8, _variant_t(TypeCode_UInt64));
    typeCodes2.SetAt(9, _variant_t(TypeCode_Single));
    typeCodes2.SetAt(10, _variant_t(TypeCode_Double));
    typeCodes2.SetAt(11, _variant_t(TypeCode_String));

    _UACallArgumentsPtr CallArguments2Ptr(__uuidof(UACallArguments));
    CallArguments2Ptr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
    CallArguments2Ptr->NodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10755";
    CallArguments2Ptr->MethodNodeDescriptor->NodeId->ExpandedText =
L"nsu=http://test.org/UA/Data;/i=10774";
    CallArguments2Ptr->InputArguments = inputs2;
    CallArguments2Ptr->InputTypeCodes = typeCodes2;

    CComSafeArray<VARIANT> arguments(2);
    arguments.SetAt(0, _variant_t((IDispatch*)CallArguments1Ptr));
    arguments.SetAt(1, _variant_t((IDispatch*)CallArguments2Ptr));

    // Instantiate the client object
    _EasyUAClientPtr ClientPtr(__uuidof(EasyUAClient));

    // Perform the operation
    LPSAFEARRAY pArguments = arguments.Detach();
    CComSafeArray<VARIANT> results;
    results.Attach(ClientPtr->CallMultipleMethods(&pArguments));
    arguments.Attach(pArguments);

    // Display results
    for (int i = results.GetLowerBound(); i <= results.GetUpperBound(); i++)
    {
        _tprintf(_T("\n"));
        _tprintf(_T("results(%d):\n"), i);
        _ValueArrayResultPtr ResultPtr = results[i];

        if (ResultPtr->Exception == NULL)
        {
            CComSafeArray<VARIANT> outputs = ResultPtr->ValueArray;
            for (int j = outputs.GetLowerBound(); j <=
outputs.GetUpperBound(); j++)
            {
                _variant_t output(outputs[j]);
                _variant_t vString;
                vString.ChangeType(VT_BSTR, &output);
                _tprintf(_T("    outputs(%d): %s\n"), i,
CW2T((_bstr_t)vString));
            }
        }
        else
            _tprintf(_T("**** Error: %s\n"), CW2T(ResultPtr->Exception-
>ToString()));
    }
}

```

```
// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}

}
```

## Free Pascal

---

```
// This example shows how to call multiple methods, and pass arguments to and
// from them.

class procedure CallMultipleMethods.Main;
var
  Arguments: OleVariant;
  CallArguments1, CallArguments2: _UACallArguments;
  Client: EasyUAClient;
  I, J: Cardinal;
  Inputs1, Inputs2: OleVariant;
  Outputs: OleVariant;
  Result: _ValueArrayResult;
  Results: OleVariant;
  TypeCodes1, TypeCodes2: OleVariant;
begin
  Inputs1 := VarArrayCreate([0, 10], varVariant);
  Inputs1[0] := False;
  Inputs1[1] := 1;
  Inputs1[2] := 2;
  Inputs1[3] := 3;
  Inputs1[4] := 4;
  Inputs1[5] := 5;
  Inputs1[6] := 6;
  Inputs1[7] := 7;
  Inputs1[8] := 8;
  Inputs1[9] := 9;
  Inputs1[10] := 10;

  TypeCodes1 := VarArrayCreate([0, 10], varVariant);
  TypeCodes1[0] := TypeCode_Boolean;
  TypeCodes1[1] := TypeCode_SByte;
  TypeCodes1[2] := TypeCode_Byte;
  TypeCodes1[3] := TypeCode_Int16;
  TypeCodes1[4] := TypeCode_UInt16;
  TypeCodes1[5] := TypeCode_Int32;
  TypeCodes1[6] := TypeCode_UInt32;
  TypeCodes1[7] := TypeCode_Int64;
  TypeCodes1[8] := TypeCode_UInt64;
  TypeCodes1[9] := TypeCode_Single;
  TypeCodes1[10] := TypeCode_Double;

  CallArguments1 := CoUACallArguments.Create;
  CallArguments1.EndpointDescriptor.UrlString := 'http://opcua.demo-
this.com:51211/UA/SampleServer';
  CallArguments1.NodeDescriptor.NodeId.ExpandedText :=
  'nsu=http://test.org/UA/Data/;i=10755';
  CallArguments1.MethodNodeDescriptor.NodeId.ExpandedText :=
  'nsu=http://test.org/UA/Data/;i=10756';
  CallArguments1.InputArguments := PSafeArray(TVarData(Inputs1).VArray);
  CallArguments1.InputTypeCodes := PSafeArray(TVarData(TypeCodes1).VArray);

  Inputs2 := VarArrayCreate([0, 11], varVariant);
  Inputs2[0] := False;
  Inputs2[1] := 1;
```

```

Inputs2[2] := 2;
Inputs2[3] := 3;
Inputs2[4] := 4;
Inputs2[5] := 5;
Inputs2[6] := 6;
Inputs2[7] := 7;
Inputs2[8] := 8;
Inputs2[9] := 9;
Inputs2[10] := 10;
Inputs2[11] := 'eleven';

TypeCodes2 := VarArrayCreate([0, 11], varVariant);
TypeCodes2[0] := TypeCode_Boolean;
TypeCodes2[1] := TypeCode_SByte;
TypeCodes2[2] := TypeCode_Byte;
TypeCodes2[3] := TypeCode_Int16;
TypeCodes2[4] := TypeCode_UInt16;
TypeCodes2[5] := TypeCode_Int32;
TypeCodes2[6] := TypeCode_UInt32;
TypeCodes2[7] := TypeCode_Int64;
TypeCodes2[8] := TypeCode_UInt64;
TypeCodes2[9] := TypeCode_Single;
TypeCodes2[10] := TypeCode_Double;
TypeCodes2[11] := TypeCode_String;

CallArguments2 := CoUACallArguments.Create;
CallArguments2.EndpointDescriptor.UrlString := 'http://opcua.demo-
this.com:51211/UA/SampleServer';
CallArguments2.NodeDescriptor.NodeId.ExpandedText :=
'nsu=http://test.org/UA/Data/;i=10755';
CallArguments2.MethodNodeDescriptor.NodeId.ExpandedText :=
'nsu=http://test.org/UA/Data/;i=10774';
CallArguments2.InputArguments := PSafeArray(TVarData(Inputs2).VArray);
CallArguments2.InputTypeCodes := PSafeArray(TVarData(TypeCodes2).VArray);

Arguments := VarArrayCreate([0, 1], varVariant);
Arguments[0] := CallArguments1;
Arguments[1] := CallArguments2;

// Instantiate the client object
Client := CoEasyUAClient.Create;

// Perform the operation
TVarData(Results).VType := varArray or varVariant;
TVarData(Results).VArray := PVarArray(Client.CallMultipleMethods(
    PSafeArray(TVarData(Arguments).VArray)));

// Display results
for I := VarArrayLowBound(Results, 1) to VarArrayHighBound(Results, 1) do
begin
    WriteLn;
    WriteLn('results(', I, '):');
    Result := IInterface(Results[I]) as _ValueArrayResult;

    if Result.Exception = nil then
        begin
            TVarData(Outputs).VType := varArray or varVariant;
            TVarData(Outputs).VArray := PVarArray(Result.ValueArray);
            for J := VarArrayLowBound(Outputs, 1) to VarArrayHighBound(Outputs, 1) do
                try
                    WriteLn('      ', 'outputs(', J, '):', Outputs[J]);
                except

```

```
        on EVariantError do WriteLn('*** Error displaying the value');
    end;
end
else
  WriteLn('*** Error: ', Result.Exception.ToString);
end;
end;
```

## Object Pascal

---

```
// This example shows how to call multiple methods, and pass arguments to and
// from them.

class procedure CallMultipleMethods.Main;
var
  Arguments: OleVariant;
  CallArguments1, CallArguments2: _UACallArguments;
  Client: TEasyUAClient;
  I, J: Cardinal;
  Inputs1, Inputs2: OleVariant;
  Outputs: OleVariant;
  Result: _ValueArrayResult;
  Results: OleVariant;
  TypeCodes1, TypeCodes2: OleVariant;
begin
  Inputs1 := VarArrayCreate([0, 10], varVariant);
  Inputs1[0] := False;
  Inputs1[1] := 1;
  Inputs1[2] := 2;
  Inputs1[3] := 3;
  Inputs1[4] := 4;
  Inputs1[5] := 5;
  Inputs1[6] := 6;
  Inputs1[7] := 7;
  Inputs1[8] := 8;
  Inputs1[9] := 9;
  Inputs1[10] := 10;

  TypeCodes1 := VarArrayCreate([0, 10], varVariant);
  TypeCodes1[0] := TypeCode_Boolean;
  TypeCodes1[1] := TypeCode_SByte;
  TypeCodes1[2] := TypeCode_Byte;
  TypeCodes1[3] := TypeCode_Int16;
  TypeCodes1[4] := TypeCode_UInt16;
  TypeCodes1[5] := TypeCode_Int32;
  TypeCodes1[6] := TypeCode_UInt32;
  TypeCodes1[7] := TypeCode_Int64;
  TypeCodes1[8] := TypeCode_UInt64;
  TypeCodes1[9] := TypeCode_Single;
  TypeCodes1[10] := TypeCode_Double;

  CallArguments1 := CoUACallArguments.Create;
  CallArguments1.EndpointDescriptor.UrlString := 'http://opcua.demo-
this.com:51211/UA/SampleServer';
  CallArguments1.NodeDescriptor.NodeId.ExpandedText :=
  'nsu=http://test.org/UA/Data/;i=10755';
  CallArguments1.MethodNodeDescriptor.NodeId.ExpandedText :=
  'nsu=http://test.org/UA/Data/;i=10756';
  CallArguments1.InputArguments := PSafeArray(TVarData(Inputs1).VArray);
  CallArguments1.InputTypeCodes := PSafeArray(TVarData(TypeCodes1).VArray);
```

```

Inputs2 := VarArrayCreate([0, 11], varVariant);
Inputs2[0] := False;
Inputs2[1] := 1;
Inputs2[2] := 2;
Inputs2[3] := 3;
Inputs2[4] := 4;
Inputs2[5] := 5;
Inputs2[6] := 6;
Inputs2[7] := 7;
Inputs2[8] := 8;
Inputs2[9] := 9;
Inputs2[10] := 10;
Inputs2[11] := 'eleven';

TypeCodes2 := VarArrayCreate([0, 11], varVariant);
TypeCodes2[0] := TypeCode_Boolean;
TypeCodes2[1] := TypeCode_SByte;
TypeCodes2[2] := TypeCode_Byte;
TypeCodes2[3] := TypeCode_Int16;
TypeCodes2[4] := TypeCode_UInt16;
TypeCodes2[5] := TypeCode_Int32;
TypeCodes2[6] := TypeCode_UInt32;
TypeCodes2[7] := TypeCode_Int64;
TypeCodes2[8] := TypeCode_UInt64;
TypeCodes2[9] := TypeCode_Single;
TypeCodes2[10] := TypeCode_Double;
TypeCodes2[11] := TypeCode_String;

CallArguments2 := CoUACallArguments.Create;
CallArguments2.EndpointDescriptor.UrlString := 'http://opcua.demo-
this.com:51211/UA/SampleServer';
CallArguments2.NodeDescriptor.NodeId.ExpandedText :=
'nsu=http://test.org/UA/Data/;i=10755';
CallArguments2.MethodNodeDescriptor.NodeId.ExpandedText :=
'nsu=http://test.org/UA/Data/;i=10774';
CallArguments2.InputArguments := PSafeArray(TVarData(Inputs2).VArray);
CallArguments2.InputTypeCodes := PSafeArray(TVarData(TypeCodes2).VArray);

Arguments := VarArrayCreate([0, 1], varVariant);
Arguments[0] := CallArguments1;
Arguments[1] := CallArguments2;

// Instantiate the client object
Client := TEasyUAClient.Create(nil);

// Perform the operation
TVarData(Results).VType := varArray or varVariant;
TVarData(Results).VArray := PVarArray(Client.CallMultipleMethods(
  PSafeArray(TVarData(Arguments).VArray)));

// Display results
for I := VarArrayLowBound(Results, 1) to VarArrayHighBound(Results, 1) do
begin
  WriteLn;
  WriteLn('results(' + IntToStr(I) + ')');
  Result := IInterface(Results[I]) as _ValueArrayResult;

  if Result.Exception = nil then
    begin
      TVarData(Outputs).VType := varArray or varVariant;
      TVarData(Outputs).VArray := PVarArray(Result.ValueArray);
      for J := VarArrayLowBound(Outputs, 1) to VarArrayHighBound(Outputs, 1) do
        begin
          WriteLn('outputs[' + IntToStr(J) + ']');
        end;
    end;
end;

```

```
        WriteLn('      ', 'outputs(', J, '): ', Outputs[J]);
    end
else
    WriteLn('*** Error: ', Result.Exception.ToString());
end;
end;
```

## Visual Basic (VB 6.)

---

Rem This example shows how to call multiple methods, and pass arguments to and from them.

```
Private Sub CallMultipleMethods_Main_Command_Click()
    OutputText = ""

    Dim inputs1(10)
    inputs1(0) = False
    inputs1(1) = 1
    inputs1(2) = 2
    inputs1(3) = 3
    inputs1(4) = 4
    inputs1(5) = 5
    inputs1(6) = 6
    inputs1(7) = 7
    inputs1(8) = 8
    inputs1(9) = 9
    inputs1(10) = 10

    Dim typeCodes1(10)
    typeCodes1(0) = 3      ' TypeCode.Boolean
    typeCodes1(1) = 5      ' TypeCode.SByte
    typeCodes1(2) = 6      ' TypeCode.Byte
    typeCodes1(3) = 7      ' TypeCode.Int16
    typeCodes1(4) = 8      ' TypeCode.UInt16
    typeCodes1(5) = 9      ' TypeCode.Int32
    typeCodes1(6) = 10     ' TypeCode.UInt32
    typeCodes1(7) = 11     ' TypeCode.Int64
    typeCodes1(8) = 12     ' TypeCode.UInt64
    typeCodes1(9) = 13     ' TypeCode.Single
    typeCodes1(10) = 14    ' TypeCode.Double

    Dim CallArguments1 As New UAArguments
    CallArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
    CallArguments1.NodeDescriptor.NodeId.expandedText =
    "nsu=http://test.org/UA/Data/;i=10755"
    CallArguments1.MethodNodeDescriptor.NodeId.expandedText =
    "nsu=http://test.org/UA/Data/;i=10756"
    ' Use SetXXXX methods instead of array-type property setters in Visual Basic 6.0
    CallArguments1.SetInputArguments inputs1
    CallArguments1.SetInputTypeCodes typeCodes1

    Dim inputs2(11)
    inputs2(0) = False
    inputs2(1) = 1
    inputs2(2) = 2
    inputs2(3) = 3
    inputs2(4) = 4
    inputs2(5) = 5
    inputs2(6) = 6
    inputs2(7) = 7
```

```

inputs2(8) = 8
inputs2(9) = 9
inputs2(10) = 10
inputs2(11) = "eleven"

Dim typeCodes2(11)
typeCodes2(0) = 3      ' TypeCode.Boolean
typeCodes2(1) = 5      ' TypeCode.SByte
typeCodes2(2) = 6      ' TypeCode.Byte
typeCodes2(3) = 7      ' TypeCode.Int16
typeCodes2(4) = 8      ' TypeCode.UInt16
typeCodes2(5) = 9      ' TypeCode.Int32
typeCodes2(6) = 10     ' TypeCode.UInt32
typeCodes2(7) = 11     ' TypeCode.Int64
typeCodes2(8) = 12     ' TypeCode.UInt64
typeCodes2(9) = 13     ' TypeCode.Single
typeCodes2(10) = 14    ' TypeCode.Double
typeCodes2(11) = 18    ' TypeCode.String

Dim CallArguments2 As New UACallArguments
CallArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
CallArguments2.NodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10755"
CallArguments2.MethodNodeDescriptor.NodeId.expandedText =
"nsu=http://test.org/UA/Data/;i=10774"
' Use SetXXXX methods instead of array-type property setters in Visual Basic 6.0
CallArguments2.SetInputArguments inputs2
CallArguments2.SetInputTypeCodes typeCodes2

Dim arguments(1) As Variant
Set arguments(0) = CallArguments1
Set arguments(1) = CallArguments2

' Instantiate the client object
Dim Client As New EasyUAClient

' Perform the operation
Dim results As Variant
results = Client.CallMultipleMethods(arguments)

' Display results
Dim i: For i = LBound(results) To UBound(results)
    OutputText = OutputText & vbCrLf
    OutputText = OutputText & "results(" & i & "):" & vbCrLf
    Dim Result As ValueArrayResult: Set Result = results(i)

    If Result.Exception Is Nothing Then
        Dim outputs As Variant: outputs = Result.ValueArray
        Dim j: For j = LBound(outputs) To UBound(outputs)
            On Error Resume Next
            OutputText = OutputText & Space(4) & "outputs(" & j & ")：" &
outputs(j) & vbCrLf
                If Err <> 0 Then OutputText = OutputText & Space(4) & "*** Error" &
vbCrLf ' occurs with types not recognized by VB6
                On Error GoTo 0
            Next
        Else
            OutputText = OutputText & "*** Error: " & Result.Exception & vbCrLf
        End If
    Next
End Sub

```

## VBScript

Rem This example shows how to call multiple methods, and pass arguments to and from them.

Option Explicit

```
Dim inputs1(10)
inputs1(0) = False
inputs1(1) = 1
inputs1(2) = 2
inputs1(3) = 3
inputs1(4) = 4
inputs1(5) = 5
inputs1(6) = 6
inputs1(7) = 7
inputs1(8) = 8
inputs1(9) = 9
inputs1(10) = 10

Dim typeCodes1(10)
typeCodes1(0) = 3      ' TypeCode.Boolean
typeCodes1(1) = 5      ' TypeCode.SByte
typeCodes1(2) = 6      ' TypeCode.Byte
typeCodes1(3) = 7      ' TypeCode.Int16
typeCodes1(4) = 8      ' TypeCode.UInt16
typeCodes1(5) = 9      ' TypeCode.Int32
typeCodes1(6) = 10     ' TypeCode.UInt32
typeCodes1(7) = 11     ' TypeCode.Int64
typeCodes1(8) = 12     ' TypeCode.UInt64
typeCodes1(9) = 13     ' TypeCode.Single
typeCodes1(10) = 14    ' TypeCode.Double

Dim CallArguments1: Set CallArguments1 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UACallArguments")
CallArguments1.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
CallArguments1.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10755"
CallArguments1.MethodNodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10756"
CallArguments1.InputArguments = inputs1
CallArguments1.InputTypeCodes = typeCodes1

Dim inputs2(11)
inputs2(0) = False
inputs2(1) = 1
inputs2(2) = 2
inputs2(3) = 3
inputs2(4) = 4
inputs2(5) = 5
inputs2(6) = 6
inputs2(7) = 7
inputs2(8) = 8
inputs2(9) = 9
inputs2(10) = 10
inputs2(11) = "eleven"

Dim typeCodes2(11)
typeCodes2(0) = 3      ' TypeCode.Boolean
typeCodes2(1) = 5      ' TypeCode.SByte
```

```
typeCodes2(2) = 6      ' TypeCode.Byte
typeCodes2(3) = 7      ' TypeCode.Int16
typeCodes2(4) = 8      ' TypeCode.UInt16
typeCodes2(5) = 9      ' TypeCode.Int32
typeCodes2(6) = 10     ' TypeCode.UInt32
typeCodes2(7) = 11     ' TypeCode.Int64
typeCodes2(8) = 12     ' TypeCode.UInt64
typeCodes2(9) = 13     ' TypeCode.Single
typeCodes2(10) = 14    ' TypeCode.Double
typeCodes2(11) = 18    ' TypeCode.String

Dim CallArguments2: Set CallArguments2 =
CreateObject("OpcLabs.EasyOpc.UA.OperationModel.UACallArguments")
CallArguments2.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
CallArguments2.NodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10755"
CallArguments2.MethodNodeDescriptor.NodeId.ExpandedText =
"nsu=http://test.org/UA/Data/;i=10774"
CallArguments2.InputArguments = inputs2
CallArguments2.InputTypeCodes = typeCodes2

Dim arguments(1)
Set arguments(0) = CallArguments1
Set arguments(1) = CallArguments2

' Instantiate the client object
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")

' Perform the operation
Dim results: results = Client.CallMultipleMethods(arguments)

' Display results
Dim i: For i = LBound(results) To UBound(results)
    WScript.Echo
    WScript.Echo "results(" & i & "):""
    Dim Result: Set Result = results(i)

    If Result.Exception Is Nothing Then
        Dim outputs: outputs = Result.ValueArray
        Dim j: For j = LBound(outputs) To UBound(outputs)
            On Error Resume Next
            WScript.Echo Space(4) & "outputs(" & j & "): " & outputs(j)
            If Err <> 0 Then WScript.Echo Space(4) & "*** Error" ' occurs with
types not recognized by VBScript
            On Error Goto 0
        Next
    Else
        WScript.Echo "*** Error: " & Result.Exception
    End If
Next
```

## 6.1.6 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](#) or [EasyUAClient](#) object.

For the [EasyDAClient](#) object, following are its instance properties, i.e. if you have created multiple [EasyDAClient](#)

objects, each will have its own copy of them:

- **Mode**: 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.

and

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

## C#

```
// This example shows how the OPC server can quickly be disconnected after writing a
value into one of its OPC items.

using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyDAClientHoldPeriods
    {
        class TopicWrite
        {
            public static void Main()
            {
                var easyDAClient = new EasyDAClient();

                easyDAClient.InstanceParameters.HoldPeriods.TopicWrite = 100; // in
milliseconds

                easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 12345);
            }
        }
    }
}
```

## VB.NET

```
' This example shows how the OPC server can quickly be disconnected after writing a
value into one of its OPC items.

Imports OpcLabs.EasyOpc.DataAccess

Namespace _EasyDAClientHoldPeriods
    Friend Class TopicWrite
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            easyDAClient.InstanceParameters.HoldPeriods.TopicWrite = 100 ' in
milliseconds

            easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"Simulation.Register_I4", 12345)
        End Sub
    End Class
End Namespace
```

For the [EasyUAClient](#) object, following are its instance properties, i.e. if you have created multiple [EasyUAClient](#) objects, each will have its own copy of them:

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

Instance properties can be modified from your code.

 In QuickOPC.NET and QuickOPC-UA, if you have placed the [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) object on the designer surface, most instance properties can also be directly edited in the Properties window in Visual Studio.



In QuickOPC-COM, your code can override the defaults if needed, by setting the properties accordingly.

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.

Following properties are static, i.e. shared among all instances of [EasyUAClient](#) object:

- EngineParameters: Contains global parameters such as frequencies of internal tasks performed by the component.

Following properties are static unless the [EasyUAClient](#) is configured as isolated:

- HostParameters: Contains parameters that influence the behavior of the component on the host level.
- DiscoveryParameters: Contains parameters that influence the behavior of the component for the discovery.
- MonitoredItemParameters: Contains parameters related to operations that target a specific monitored item (e.g. a retry delay).
- **SessionParameters:** Contains parameters that influence operations that target an OPC-UA session, such as Endpoint Selection Policy, User Identity, and Keep Alive Interval.
- SubscriptionParameters: Contains parameters that influence operations that target an OPC-UA subscription (e.g. a retry delay).

Please use the Reference documentation for details on meaning of various properties and their use.

Static properties can be modified from your code. If you want to modify any of the static properties, you must do it before the first instance of [EasyDAClient](#), [EasyAEClient](#) or [EasyUAObject](#) object is created.

You can also use the following components to set the static properties:

Configuration Component	Sets Static Properties of
<a href="#">EasyAEClientConfiguration</a> component - 	<a href="#">EasyAEClient</a> class
<a href="#">EasyDAClientConfiguration</a> component - 	<a href="#">EasyDAClient</a> class
<a href="#">EasyUAClientConfiguration</a> component - 	<a href="#">EasyUAClient</a> class

In Visual Studio, drag the corresponding component from the Toolbox to the designer surface, select it, and you can then view and modify the settings in the Properties window.

Note that the [EasyXXClientConfiguration](#) (where **XX** is either **AE/DA/UA**) components are just a "peek" to properties

of the [EasyXXClient](#) that are static in nature. For this reason, you should normally have just [one](#) instance of the [EasyXXClientConfiguration](#) component in your application, and you should place it in such a way that it gets instantiated before the first operations on the [EasyXXClient](#) object take place. You can achieve this easily e.g. by placing the [EasyXXClientConfiguration](#) component onto the main (first) form of your application.

The [EasyUAClientConfiguration](#) component also has a [LogEntry](#) event, which functions as a “projection” of the static [EasyUAClient.LogEntry](#) event. You can easily hook an event handler to this event to process log entries generated by the [EasyUAClient](#) component.

## 6.1.6.1 User Identity in QuickOPC-UA

OPC UA Servers may require that the user making a connection from the OPC UA client is authenticated, and reject the connection if the authentication fails. In addition, different users may have different permissions (authorization) for various operations on the OPC UA Server.

You can specify the identity of the user making the connection using the [UserIdentity](#) property. This property is a [UserIdentity](#) object that contains following user token infos:

- [AnonymousTokenInfo](#): An anonymous user.
- [UserNameTokenInfo](#): A user name token (with optional password).
- [X509CertificateTokenInfo](#): A user token represented by an X.509 certificate.
- [KerberosTokenInfo](#): A Kerberos (issued) user token.

Note: The Kerberos token info may represent an explicitly specified user identity (if you set [KerberosTokenInfo.NetworkSecurity.CustomNetworkCredential](#) to [true](#) and specify additional parameters, such as the user name, password, and domain), or it can represent the current user running the code (if [KerberosTokenInfo.NetworkSecurity.CustomNetworkCredential](#) is set to [false](#)).

Zero, one, or more user token infos (of different types) may be specified in the [UserIdentity](#) object. By default, no user token info is specified. The user token infos are always present (i.e. non-null), but they are only used if they are filled in with data. For example, if you leave the [UserName](#) and [Password](#) in the [UserNameTokenInfo](#) empty, the user name token will not be used. If you, however, start putting values into any of the token infos, you need to fill in everything necessary in that token, otherwise an error may occur. For an [AnonymousTokenInfo](#), the anonymous token is used when its [IsConfigured](#) property is set to [true](#).

You can easily create a [UserIdentity](#) with certain user token by one of the following static methods:

- [UserIdentity.CreateAnonymousIdentity](#)
- [UserIdentity.CreateKerberosIdentity](#)
- [UserIdentity.CreateUserNameIdentity](#)
- [UserIdentity.CreateX509Certificate](#)

When QuickOPC-UA makes a connection to the OPC UA server, it selects the user token according to its built-in token selection policy. The OPC UA server is interrogated for user token policies available on the endpoint, and the QuickOPC-UA selects the most appropriate one from them.

When you set the user identity in the above described way, i.e. in the session parameters object, it applies to all sessions (connections) made by that [EasyUAClient](#) object. In addition to this, it is also possible to specify the user identity directly for a specific connection, i.e. on the [UAEndpointDescriptor](#) object. There is a [UserIdentity](#) property on the [UAEndpointDescriptor](#) as well, and the user token infos contained there are merged together with those coming from the session parameters, for each connection made on that endpoint.

## 6.1.6.2 Server Diagnostics in OPC-UA

The [DiagnosticsMasks](#) property on the [SessionParameters](#) allows the developer to specify the types of vendor-specific diagnostics to be returned by the OPC-UA server in the responses. The information from the server responses is then

made available in the [Diagnostics](#) collection of [OperationResult](#) and [EasyDataChangeNotificationEventArgs](#) objects.

## 6.2 Procedural Coding Model for OPC Classic A&E

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

### 6.2.1 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.

#### 6.2.1.1 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.

##### C#

```
// This example shows how to obtain current state information for the condition
// instance corresponding to a Source and
// certain ConditionName.

using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;

namespace DocExamples {
    namespace _EasyAEClient {

        class GetConditionState
        {
            public static void Main()
            {
                var easyAEClient = new EasyAEClient();
                AEConditionState conditionState = easyAEClient.GetConditionState("", "OPCLabs.KitEventServer.2",
                    "Simulation.ConditionState1", "Simulated");

                Console.WriteLine("ConditionState:");
                Console.WriteLine("    .ActiveSubcondition: {0}",
                    conditionState.ActiveSubcondition);
                Console.WriteLine("    .Enabled: {0}", conditionState.Enabled);
                Console.WriteLine("    .Active: {0}", conditionState.Active);
                Console.WriteLine("    .Acknowledged: {0}",
                    conditionState.Acknowledged);
                Console.WriteLine("    .Quality: {0}", conditionState.Quality);
            }
        }
    }
}
```

```
// Remark: IAEConditionState has many more properties
}
}

}

//
```

## VB.NET

```
' This example shows how to obtain current state information for the condition
instance corresponding to a Source and
' certain ConditionName.

Imports OpcLabs.EasyOpc.AlarmsAndEvents

Namespace _EasyAEClient

    Friend Class GetConditionState
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim conditionState As AEConditionState =
easyAEClient.GetConditionState("", "OPCLabs.KitEventServer.2",
"Simulation.ConditionState1", "Simulated")

            Console.WriteLine("ConditionState:")
            Console.WriteLine("    .ActiveSubcondition: {0}",
conditionState.ActiveSubcondition)
            Console.WriteLine("    .Enabled: {0}", conditionState.Enabled)
            Console.WriteLine("    .Active: {0}", conditionState.Active)
            Console.WriteLine("    .Acknowledged: {0}", conditionState.Acknowledged)
            Console.WriteLine("    .Quality: {0}", conditionState.Quality)
            ' Remark: IAEConditionState has many more properties
        End Sub
    End Class
End Namespace
```

## VBScript

```
Rem This example shows how to obtain current state information for the condition
instance corresponding to a Source and
Rem certain ConditionName.

Option Explicit

Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim SourceDescriptor: Set SourceDescriptor =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor.QualifiedName = "Simulation.ConditionState1"

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
Dim ConditionState: Set ConditionState = Client.GetConditionState(ServerDescriptor,
SourceDescriptor, "Simulated", Array())

WScript.Echo "ConditionState:"
With ConditionState
    WScript.Echo Space(4) & ".ActiveSubcondition: " & .ActiveSubcondition
    WScript.Echo Space(4) & ".Enabled: " & .Enabled
End With
```

```
WScript.Echo Space(4) & ".Active: " & .Active
WScript.Echo Space(4) & ".Acknowledged: " & .Acknowledged
WScript.Echo Space(4) & ".Quality: " & .Quality
Rem Note that IAEConditionState has many more properties
End With
```



In QuickOPC.NET, you can alternatively pass in a [ServerDescriptor](#) in place of machine name and server class arguments.

## 6.2.2 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.

### 6.2.2.1 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.

#### C#

```
// This example shows how to acknowledge an event condition in the OPC server.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.DataAccess;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
    {

        class AcknowledgeCondition
        {
            [NotNull]
            static readonly EasyAEClient EasyAEClient = new EasyAEClient();
            [NotNull]
            static readonly EasyDAClient EasyDAClient = new EasyDAClient();

            static volatile bool _done;

            public static void Main()
            {
                var eventHandler = new
                    EasyAENotificationEventHandler(easyAEClient_Notification);
                EasyAEClient.Notification += eventHandler;

                Console.WriteLine("Processing event notifications for 1 minute...");
                var subscriptionFilter = new AESubscriptionFilter
                {
                    Sources = new AENodeDescriptor[] {"Simulation.ConditionState1"}
                };
            }
        }
    }
}
```

VB.NET

' This example shows how to acknowledge an event condition in the OPC server.

```
Imports JetBrains.Annotations  
Imports OpcLabs.EasyOpc.AlarmsAndEvents
```

```
Imports OpcLabs.EasyOpc.DataAccess
Imports System.Threading

Namespace _EasyAEClient

    Friend Class AcknowledgeCondition
        <NotNull()>
        Private Shared ReadOnly EasyAEClient As New EasyAEClient()
        <NotNull()>
        Private Shared ReadOnly EasyDAClient As New EasyDAClient()

        Private Shared _done As Boolean ' volatile

        Public Shared Sub Main()
            Dim eventHandler = New EasyAENotificationEventHandler(AddressOf
easyAEClient_Notification)
            AddHandler EasyAEClient.Notification, eventHandler

            Console.WriteLine("Processing event notifications for 1 minute...")
            Dim subscriptionFilter As New AESubscriptionFilter
            subscriptionFilter.Sources = New AENodeDescriptor()
            {"Simulation.ConditionState1"}
            Dim handle As Integer = EasyAEClient.SubscribeEvents("",,
"OPCLabs.KitEventServer.2", 1000, Nothing, subscriptionFilter)

            ' Give the refresh operation time to complete
            Thread.Sleep(5 * 1000)

            ' Trigger an acknowledgeable event
            EasyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Activate", True)

            _done = False
            Dim endTime As Date = Date.Now + New TimeSpan(0, 0, 5)
            Do While ((Not _done)) AndAlso (Date.Now < endTime)
                Thread.Sleep(1000)
            Loop

            ' Give some time to also receive the acknowledgement notification
            Thread.Sleep(5 * 1000)

            EasyAEClient.UnsubscribeEvents(handle)
        End Sub

        ' Notification event handler
        Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)
            Console.WriteLine()
            Console.WriteLine("Refresh: {0}", e.Refresh)
            Console.WriteLine("RefreshComplete: {0}", e.RefreshComplete)
            If e.EventData IsNot Nothing Then
                Dim eventData As AEEEventData = e.EventData
                Console.WriteLine("EventData.QualifiedSourceName: {0}",
eventData.QualifiedSourceName)
                Console.WriteLine("EventData.Message: {0}", eventData.Message)
                Console.WriteLine("EventData.Active: {0}", eventData.Active)
                Console.WriteLine("EventData.Acknowledged: {0}",
eventData.Acknowledged)
                Console.WriteLine("EventData.AcknowledgeRequired: {0}",
eventData.AcknowledgeRequired)

                If eventData.AcknowledgeRequired Then

```

```
        Console.WriteLine(">>>> ACKNOWLEDGING THIS EVENT")
        EasyAEClient.AcknowledgeCondition("", 
"OPCLabs.KitEventServer.2", "Simulation.ConditionState1", "Simulated",
eventData.ActiveTime, eventData.Cookie)
        Console.WriteLine(">>>> EVENT ACKNOWLEDGED")
        _done = True
    End If
End If
End Sub
End Class
End Namespace
```

## VBScript

Rem This example shows how to acknowledge an event condition in the OPC server.

```
Option Explicit

Dim DAClient: Set DAClient = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim AEClient: Set AEClient =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")

WScript.Echo "Hooking event handler..."
WScript.ConnectObject AEClient, "AEClient_"

Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim SourceDescriptor: Set SourceDescriptor =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor.QualifiedName = "Simulation.ConditionState1"

WScript.Echo "Processing event notifications for 1 minute..."
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.Filter.Sources = Array(SourceDescriptor)
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = AEClient.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

WScript.Echo "Give the refresh operation time to complete: Waiting for 5 seconds..."
WScript.Sleep 5*1000

WScript.Echo "Triggering an acknowledgeable event..."
DAClient.WriteItemValue "", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Activate", True

Dim done: done = False
Dim endTime: endTime = Now() + 5*(1/24/60/60)
While (Not done) And (Now() < endTime)
    WScript.Sleep 1000
WEnd

WScript.Echo "Give some time to also receive the acknowledgement notification:
Waiting for 5 seconds..."
WScript.Sleep 5*1000

WScript.Echo "Unsubscribing events..."
AEClient.UnsubscribeEvents handle
```

```
WScript.Echo "Unhooking event handler..."  
WScript.DisconnectObject AEClient  
  
WScript.Echo "Finished."  
  
Rem Notification event handler  
Sub AEClient_Notification(Sender, e)  
    WScript.Echo  
    WScript.Echo "Refresh: " & e.Refresh  
    WScript.Echo "RefreshComplete: " & e.RefreshComplete  
    If Not (e.EventData Is Nothing) Then  
        With e.EventData  
            WScript.Echo "EventData.QualifiedSourceName: " & .QualifiedSourceName  
            WScript.Echo "EventData.Message: " & .Message  
            WScript.Echo "EventData.Active: " & .Active  
            WScript.Echo "EventData.Acknowledged: " & .Acknowledged  
            WScript.Echo "EventData.AcknowledgeRequired: " & .AcknowledgeRequired  
  
            If .AcknowledgeRequired Then  
                WScript.Echo ">>>> ACKNOWLEDGING THIS EVENT"  
                AEClient.AcknowledgeCondition ServerDescriptor, SourceDescriptor,  
"Simulated", _  
                .ActiveTime, .Cookie, "aUser", ""  
                WScript.Echo ">>>> EVENT ACKNOWLEDGED"  
                done = True  
            End If  
        End With  
    End If  
End Sub
```

Optionally, you can pass in acknowledger ID (who is acknowledging the condition), and a comment string.  
 You can alternatively pass in a [ServerDescriptor](#) in place of machine name and server class arguments.

## 6.2.3 Browsing for Information

QuickOPC 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).

### 6.2.3.1 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).

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.

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.

## C#

```
// This example shows how to obtain all ProgIDs of all OPC Alarms and Events servers
// on the local machine.
using System.Diagnostics;
using OpcLabs.EasyOpc;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;

namespace DocExamples {
namespace _EasyAEClient {

    class BrowseServers
    {
        public static void Main()
        {
            var easyAEClient = new EasyAEClient();
            ServerElementCollection serverElements = easyAEClient.BrowseServers("");

            foreach (ServerElement serverElement in serverElements)
            {
                Debug.Assert(serverElement != null);
                Console.WriteLine("serverElements[{0}].ProgId: {1}",
serverElement.Clsid, serverElement.ProgId);
            }
        }
    }
}
```

## VB.NET

```
' This example shows how to obtain all ProgIDs of all OPC Alarms and Events servers
' on the local machine.
Imports OpcLabs.EasyOpc
Imports OpcLabs.EasyOpc.AlarmsAndEvents

Namespace _EasyAEClient

    Friend Class BrowseServers
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim serverElements As ServerElementCollection =
easyAEClient.BrowseServers("")

            For Each serverElement As ServerElement In serverElements
                Debug.Assert(serverElement IsNot Nothing)
                Console.WriteLine("serverElements[{0}].ProgId: {1}",
serverElement.Clsid, serverElement.ProgId)
            Next serverElement
        End Sub
    End Class
End Namespace
```

## VBScript

```
Rem This example shows how to obtain all ProgIDs of all OPC Alarms and Events
servers on the local machine.
```

## Option Explicit

```
Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
Dim ServerElements: Set ServerElements = Client.BrowseServers("")

Dim ServerElement: For Each ServerElement In ServerElements
    WScript.Echo "ServerElements(\"\"") & ServerElement.UrlString & "\"\").ProgId: " &
    ServerElement.ProgId
Next
```

## 6.2.3.2 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. You will receive an [AENodeElementCollection](#) object. Each [AENodeElement](#) contains information gathered about one sub-area node, such as its name, or an indication whether it has children.

### C#

```
// This example shows how to obtain all areas directly under the root (denoted by
// empty string for the parent).
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace;
using System;
using System.Diagnostics;

namespace DocExamples {
    namespace _EasyAEClient {

        class BrowseAreas
        {
            public static void Main()
            {
                var easyAEClient = new EasyAEClient();
                AENodeElementCollection nodeElements = easyAEClient.BrowseAreas("", 
                    "OPCLabs.KitEventServer.2", "");

                foreach (AENodeElement nodeElement in nodeElements)
                {
                    Debug.Assert(nodeElement != null);

                    Console.WriteLine("nodeElements[\"{0}\"]:", nodeElement.Name);
                    Console.WriteLine("    .QualifiedName: {0}",
                        nodeElement.QualifiedName);
                }
            }
        }
    }
}
```

```
        }
    }
}

}
```

## VB.NET

```
' This example shows how to obtain all areas directly under the root (denoted by
empty string for the parent).

Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace

Namespace _EasyAEClient

    Friend Class BrowseAreas
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim nodeElements As AENodeElementCollection =
easyAEClient.BrowseAreas("", "OPCLabs.KitEventServer.2", "")

            For Each nodeElement As AENodeElement In nodeElements
                Debug.Assert(nodeElement IsNot Nothing)

                Console.WriteLine("nodeElements[""{0}""]:", nodeElement.Name)
                Console.WriteLine("      .Qualifiedname: {0}",
nodeElement.QualifiedName)
            Next nodeElement
        End Sub
    End Class
End Namespace
```

## VBScript

```
Rem This example shows how to obtain all areas directly under the root (denoted by
empty string for the parent).

Option Explicit

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
Dim NodeElements: Set NodeElements = Client.BrowseAreas("", 
"OPCLabs.KitEventServer.2", "")

Dim NodeElement: For Each NodeElement In NodeElements
    WScript.Echo "NodeElements("") & NodeElement.Name & ""):""
    With NodeElement
        WScript.Echo Space(4) & ".Qualifiedname: " & .Qualifiedname
    End With
Next
```

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, 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](#).

## C#

```
// This example shows how to obtain all sources under the "Simulation" area.
using System.Diagnostics;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace;
using System;

namespace DocExamples
{
    namespace _EasyAEClient
    {

        class BrowseSources
        {
            public static void Main()
            {
                var easyAEClient = new EasyAEClient();
                AENodeElementCollection nodeElements =
easyAEClient.BrowseSources("", "OPCLabs.KitEventServer.2", "Simulation");

                foreach (AENodeElement nodeElement in nodeElements)
                {
                    Debug.Assert(nodeElement != null);

                    Console.WriteLine("nodeElements[{0}]:", nodeElement.Name);
                    Console.WriteLine("    .QualifiedName: {0}",
nodeElement.QualifiedName);
                }
            }
        }
    }
}
```

## VB.NET

```
' This example shows how to obtain all sources under the "Simulation" area.

Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace

Namespace _EasyAEClient

    Friend Class BrowseSources
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim nodeElements As AENodeElementCollection =
easyAEClient.BrowseSources("", "OPCLabs.KitEventServer.2", "Simulation")

            For Each nodeElement As AENodeElement In nodeElements
                Debug.Assert(nodeElement IsNot Nothing)

                Console.WriteLine("nodeElements[""{0}""]: ", nodeElement.Name)
                Console.WriteLine("    .QualifiedName: {0}",
nodeElement.QualifiedName)
            Next nodeElement
        End Sub
    End Class
End Namespace
```

## VBScript

```
Rem This example shows how to obtain all sources under the "Simulation" area.
```

```
Option Explicit
```

```
Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
Dim NodeElements: Set NodeElements = Client.BrowseSources("", 
"OPCLabs.KitEventServer.2", "Simulation")

Dim NodeElement: For Each NodeElement In NodeElements
    WScript.Echo "NodeElements("") & NodeElement.Name & ""):""
    With NodeElement
        WScript.Echo Space(4) & ".QualifiedName: " & .QualifiedName
    End With
Next
```

QuickOPC.NET: The most generic address space browsing method is [BrowseNodes](#). It combines the functionality of [BrowseAreas](#) and [BrowseSources](#), and it also allows the widest range of filtering options by passing an argument of type [AEBrowseParameters](#).

### 6.2.3.3 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. You will receive back an [AECategoryElementCollection](#) object.

#### C#

```
// This example shows how to enumerate all event categories provided by the OPC
// server. For each category, it displays its Id
// and description.
using System.Diagnostics;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace;
using System;

namespace DocExamples {
namespace _EasyAEClient {

    class QueryEventCategories
    {
        public static void Main()
        {
            var easyAEClient = new EasyAEClient();
            AECategoryElementCollection categoryElements =
easyAEClient.QueryEventCategories("", "OPCLabs.KitEventServer.2");

            foreach (AECategoryElement categoryElement in categoryElements)
            {
                Debug.Assert(categoryElement != null);
                Console.WriteLine("CategoryElements[" + categoryElement.CategoryId + "].Description: " + categoryElement.Description);
            }
        }
    }
}
```

```
    }  
}  
}
```

## VB.NET

---

```
' This example shows how to enumerate all event categories provided by the OPC  
server. For each category, it displays its Id  
' and description.
```

```
Imports OpcLabs.EasyOpc.AlarmsAndEvents  
Imports OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace  
  
Namespace _EasyAEClient  
  
    Friend Class QueryEventCategories  
        Public Shared Sub Main()  
            Dim easyAEClient = New EasyAEClient()  
            Dim categoryElements As AECategoryElementCollection =  
                easyAEClient.QueryEventCategories("", "OPCLabs.KitEventServer.2")  
  
            For Each categoryElement As AECategoryElement In categoryElements  
                Debug.Assert(categoryElement IsNot Nothing)  
                Console.WriteLine("CategoryElements[""{}""].Description: {}",  
                    categoryElement.CategoryId, categoryElement.Description)  
            Next categoryElement  
        End Sub  
    End Class  
  
End Namespace
```

## VBScript

---

```
Rem This example shows how to enumerate all event categories provided by the OPC  
server. For each category, it displays its Id  
Rem and description.
```

```
Option Explicit  
  
Const AEEventTypes_All = 7  
  
Dim ServerDescriptor: Set ServerDescriptor =  
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")  
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"  
  
Dim Client: Set Client =  
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")  
Dim CategoryElements: Set CategoryElements =  
Client.QueryEventCategories(ServerDescriptor, AEEventTypes_All)  
  
Dim CategoryElement: For Each CategoryElement In CategoryElements  
    WScript.Echo "CategoryElements(" & CategoryElement.CategoryId & ").Description:  
" & CategoryElement.Description  
Next
```

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.

## C#

```
// This example shows information available about OPC event category.
using System.Diagnostics;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace;
using System;

namespace DocExamples {
namespace _AECategoryElement {

    class Properties
    {
        public static void Main()
        {
            var easyAEClient = new EasyAEClient();
            AECategoryElementCollection categoryElements =
easyAEClient.QueryEventCategories("", "OPCLabs.KitEventServer.2");

            foreach (AECategoryElement categoryElement in categoryElements)
            {
                Debug.Assert(categoryElement != null);
                Debug.Assert(categoryElement.AttributeElements.Keys != null);
                Debug.Assert(categoryElement.ConditionElements.Keys != null);

                Console.WriteLine("Information about category {0}:",
categoryElement);
                Console.WriteLine("    .CategoryId: {0}",
categoryElement.CategoryId);
                Console.WriteLine("    .Description: {0}",
categoryElement.Description);
                Console.WriteLine("    .ConditionElements:");
                foreach (string conditionKey in
categoryElement.ConditionElements.Keys)
                    Console.WriteLine("        {0}", conditionKey);
                Console.WriteLine("    .AttributeElements:");
                foreach (long attributeKey in
categoryElement.AttributeElements.Keys)
                    Console.WriteLine("        {0}", attributeKey);
            }
        }
    }
}
```

## VB.NET

```
' This example shows information available about OPC event category.

Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace

Namespace _AECategoryElement

    Friend Class Properties
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim categoryElements As AECategoryElementCollection =
easyAEClient.QueryEventCategories("", "OPCLabs.KitEventServer.2")

            For Each categoryElement As AECategoryElement In categoryElements
```

```
Debug.Assert(categoryElement IsNot Nothing)
Debug.Assert(categoryElement.AttributeElements.Keys IsNot Nothing)
Debug.Assert(categoryElement.ConditionElements.Keys IsNot Nothing)

Console.WriteLine("Information about category {0}:",
categoryElement)
    Console.WriteLine("    .CategoryId: {0}",
categoryElement.CategoryId)
        Console.WriteLine("        .Description: {0}",
categoryElement.Description)
            Console.WriteLine("        .ConditionElements:")
                For Each conditionKey As String In
categoryElement.ConditionElements.Keys
                    Console.WriteLine("            {0}", conditionKey)
                    Next conditionKey
                    Console.WriteLine("            .AttributeElements:")
                        For Each attributeKey As Long In
categoryElement.AttributeElements.Keys
                            Console.WriteLine("                {0}", attributeKey)
                            Next attributeKey
                        Next categoryElement
                    End Sub
                End Class

End Namespace
```

## 6.2.3.4 Querying for OPC Event Conditions on a Category

The [EasyAEClient.QueryCategoryConditions](#) method finds out event conditions supported by given event category.

## 6.2.3.5 Querying for OPC Event Conditions on a Source

The [EasyAEClient.QuerySourceConditions](#) method finds out event conditions associated with the given event source.

### C#

---

```
// This example shows how to enumerate all event conditions associated with the
// specified event source.
using System.Diagnostics;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace;
using System;

namespace DocExamples {
    namespace _EasyAEClient {

        class QuerySourceConditions
        {
            public static void Main()
            {
                var easyAEClient = new EasyAEClient();
```

```
    AEConditionElementCollection conditionElements =
easyAEClient.QuerySourceConditions(
    "", "OPCLabs.KitEventServer.2", "Simulation.ConditionState1");

    foreach (AEConditionElement conditionElement in conditionElements)
{
    Debug.Assert(conditionElement != null);
    Console.WriteLine("ConditionElements[\"{0}\"]: {1} subcondition(s)",

        conditionElement.Name,
conditionElement.SubconditionNames.Length);
}
}

} }
```

## VB.NET

```
' This example shows how to enumerate all event conditions associated with the
specified event source.
```

```
Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports OpcLabs.EasyOpc.AlarmsAndEvents.AddressSpace

Namespace _EasyAEClient

    Friend Class QuerySourceConditions
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim conditionElements As AEConditionElementCollection =
easyAEClient.QuerySourceConditions( _
    "", "OPCLabs.KitEventServer.2", "Simulation.ConditionState1")

            For Each conditionElement As AEConditionElement In conditionElements
                Debug.Assert(conditionElement IsNot Nothing)
                Console.WriteLine("ConditionElements[\"{0}\"]: {1} subcondition(s)",

                    conditionElement.Name,
conditionElement.SubconditionNames.Length)
                Next conditionElement
            End Sub
        End Class
    End Namespace
```

## VBScript

```
Rem This example shows how to enumerate all event conditions associated with the
specified event source.
```

```
Option Explicit

Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim SourceDescriptor: Set SourceDescriptor =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor.QualifiedName = "Simulation.ConditionState1"
```

```
Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
Dim ConditionElements: Set ConditionElements =
Client.QuerySourceConditions(ServerDescriptor, SourceDescriptor)

Dim ConditionElement: For Each ConditionElement In ConditionElements
    WScript.Echo "ConditionElements(" & ConditionElement.Name & "): " &
    (UBound)(ConditionElement.SubconditionNames) + 1) & " subcondition(s)"
Next
```

## 6.2.3.6 Querying for OPC Event Attributes

The [EasyAEClient.QueryCategoryAttributes](#) method finds out event attributes that the server can provide as part of an event notification within a given event category.

## 6.2.4 Subscribing to 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.

### 6.2.4.1 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.

#### C#

---

```
// This example shows how to subscribe to events and display the event message with
// each notification. It also shows how to
// unsubscribe afterwards.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
    {
        partial class SubscribeEvents
        {
            public static void Main()
            {
                using (var easyAEClient = new EasyAEClient())
                {
                    var eventHandler = new
```

```
EasyAENotificationEventHandler(easyAEClient_Notification);
    easyAEClient.Notification += eventHandler;

        int handle = easyAEClient.SubscribeEvents("", 
"OPCLabs.KitEventServer.2", 1000);

        Console.WriteLine("Processing event notifications for 1
minute...");
        Thread.Sleep(60 * 1000);

        easyAEClient.UnsubscribeEvents(handle);
    }
}

// Notification event handler
static void easyAEClient_Notification([NotNull] object sender, [NotNull]
EasyAENotificationEventArgs e)
{
    if (e.EventData != null)
        Console.WriteLine(e.EventData.Message);
}
}
}
```

## VB.NET

```
' This example shows how to subscribe to events and display the event message with
each notification. It also shows how to
' unsubscribe afterwards.

Imports JetBrains.Annotations
Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports System.Threading

Namespace _EasyAEClient
    Partial Friend Class SubscribeEvents
        Public Shared Sub Main()
            Using easyAEClient = New EasyAEClient()
                Dim eventHandler = New EasyAENotificationEventHandler(AddressOf
easyAEClient_Notification)
                    AddHandler easyAEClient.Notification, eventHandler

                    Dim handle As Integer = easyAEClient.SubscribeEvents("", 
"OPCLabs.KitEventServer.2", 1000)

                    Console.WriteLine("Processing event notifications for 1 minute...")
                    Thread.Sleep(60 * 1000)

                    easyAEClient.UnsubscribeEvents(handle)
            End Using
        End Sub

        ' Notification event handler
        Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)
            If e.EventData IsNot Nothing Then
                Console.WriteLine(e.EventData.Message)
            End If
        End Sub
    End Class

```

End Namespace

## Object Pascal

```
// This example shows how to subscribe to events display the event message with each
// notification. It also shows how to
// unsubscribe afterwards.

type
  TClientEventHandlers = class
    // Notification event handler
    procedure OnNotification(
      ASender: TObject;
      sender: OleVariant;
      const eventArgs: _EasyAENotificationEventArgs);
  end;

procedure TClientEventHandlers.OnNotification(
  ASender: TObject;
  sender: OleVariant;
  const eventArgs: _EasyAENotificationEventArgs);
begin
  if eventArgs.EventData <> nil then
    WriteLn(eventArgs.EventData.Message);
end;

class procedure SubscribeEvents.Main;
var
  Client: TEasyAEClient;
  ClientEventHandlers: TClientEventHandlers;
  Handle: Integer;
  ServerDescriptor: _ServerDescriptor;
  State: OleVariant;
  SubscriptionParameters: _AESubscriptionParameters;
begin
  ServerDescriptor := CoServerDescriptor.Create;
  ServerDescriptor.ServerClass := 'OPCLabs.KitEventServer.2';

  // Instantiate the client object and hook events
  Client := TEasyAEClient.Create(nil);
  ClientEventHandlers := TClientEventHandlers.Create;
  Client.OnNotification := ClientEventHandlers.OnNotification;

  WriteLn('Subscribing events...');
  SubscriptionParameters := CoAESubscriptionParameters.Create;
  SubscriptionParameters.NotificationRate := 1000;
  Handle := Client.SubscribeEvents(ServerDescriptor, SubscriptionParameters, true,
  State);

  WriteLn('Processing event notifications for 1 minute...');
  PumpSleep(60*1000);

  WriteLn('Unsubscribing events...');
  Client.UnsubscribeEvents(Handle);

  WriteLn('Finished.');
end;
```

## VBScript

Rem This example shows how to subscribe to events display the event message with

each notification. It also shows how to Rem unsubscribe afterwards.

## Option Explicit

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing events..."
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = Client.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

WScript.Echo "Processing event notifications for 1 minute..."
WScript.Sleep 60*1000

WScript.Echo "Unsubscribing events..."
Client.UnsubscribeEvents handle

WScript.Echo "Finished."

Rem Notification event handler
Sub Client_Notification(Sender, e)
    If Not e.EventData Is Nothing Then WScript.Echo e.EventData.Message
End Sub
```

Optionally, you can specify a subscription filter; it is a separate object of [AESubscriptionFilterType](#) type. 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.Arguments.State](#) object.

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.

## 6.2.4.2 Specifying event filters

Examples:

Filtering by categories:

### C#

---

```
// This example shows how to filter the events by their category.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
    {
        partial class SubscribeEvents
        {
            [NotNull]
            static readonly EasyAEClient EasyAEClient = new EasyAEClient();
            [NotNull]
            static readonly EasyDAClient EasyDaClient = new EasyDAClient();

            public static void FilterByCategories()
            {
                var eventHandler = new
                    EasyAENotificationEventHandler(easyAEClient_Notification_FilterByCategories);
                EasyAEClient.Notification += eventHandler;

                Console.WriteLine("Processing event notifications...");
                var subscriptionFilter = new AESubscriptionFilter
                {
                    Categories = new long[] { 15531778 }
                };
                // You can also filter using event types, severity, areas, and
                sources.
                int handle = EasyAEClient.SubscribeEvents("", "OPCLabs.KitEventServer.2", 1000, null, subscriptionFilter);

                // Allow time for initial refresh
                Thread.Sleep(5 * 1000);

                // Set some events to active state.
                EasyDaClient.WriteItemValue("", "OPCLabs.KitServer.2",
                    "SimulateEvents.ConditionState1.Activate", true);
                EasyDaClient.WriteItemValue("", "OPCLabs.KitServer.2",
                    "SimulateEvents.ConditionState2.Activate", true);

                Thread.Sleep(10 * 1000);

                EasyAEClient.UnsubscribeEvents(handle);
            }

            // Notification event handler
            static void easyAEClient_Notification_FilterByCategories(
                [NotNull] object sender,
```

```
[NotNull] EasyAENotificationEventArgs e)
{
    Console.WriteLine();
    Console.WriteLine(e);
    Console.WriteLine("Refresh: {0}", e.Refresh);
    Console.WriteLine("RefreshComplete: {0}", e.RefreshComplete);
    AEEEventData eventData = e.EventData;
    if (eventData != null)
    {
        Console.WriteLine("Event.CategoryId: {0}",
eventData.CategoryId);
        Console.WriteLine("Event.QualifiedSourceName: {0}",
eventData.QualifiedSourceName);
        Console.WriteLine("Event.Message: {0}", eventData.Message);
        Console.WriteLine("Event.Active: {0}", eventData.Active);
        Console.WriteLine("Event.Acknowledged: {0}",
eventData.Acknowledged);
    }
}
}
```

More examples:

## C#

```
// This example shows how to set the filtering criteria to be used for the event
subscription.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.DataAccess;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _AESubscriptionFilter
    {

        class Properties
        {
            [NotNull]
            static readonly EasyAEClient EasyAEClient = new EasyAEClient();
            [NotNull]
            static readonly EasyDAClient EasyDaClient = new EasyDAClient();

            public static void Main()
            {
                var eventHandler = new
EasyAENotificationEventHandler(easyAEClient_Notification);
                EasyAEClient.Notification += eventHandler;

                Console.WriteLine("Processing event notifications...");
                var subscriptionFilter = new AESubscriptionFilter
                {
                    Sources = new AENodeDescriptor[] {"Simulation.ConditionState1",
"Simulation.ConditionState3"}
                };
                // You can also filter using event types, categories, severity, and
areas.
```

```
        int handle = EasyAEClient.SubscribeEvents("",  
"OPCLabs.KitEventServer.2", 1000, null, subscriptionFilter);  
  
        // Allow time for initial refresh  
        Thread.Sleep(5 * 1000);  
  
        // Set some events to active state.  
        // The activation below will come from a source contained in a  
filter and the notification will arrive.  
        EasyDaClient.WriteItemValue("", "OPCLabs.KitServer.2",  
"SimulateEvents.ConditionState1Activate", true);  
        // The activation below will come from a source that is not  
contained in a filter and the notification will not arrive.  
        EasyDaClient.WriteItemValue("", "OPCLabs.KitServer.2",  
"SimulateEvents.ConditionState2Activate", true);  
  
        Thread.Sleep(10 * 1000);  
  
        EasyAEClient.UnsubscribeEvents(handle);  
    }  
  
    // Notification event handler  
    static void easyAEClient_Notification([NotNull] object sender, [NotNull]  
EasyAENotificationEventArgs e)  
    {  
        Console.WriteLine();  
        Console.WriteLine("Refresh: {0}", e.Refresh);  
        Console.WriteLine("RefreshComplete: {0}", e.RefreshComplete);  
        AEEEventData eventData = e.EventData;  
        if (eventData != null)  
        {  
            Console.WriteLine("Event.QualifiedSourceName: {0}",  
eventData.QualifiedSourceName);  
            Console.WriteLine("Event.Message: {0}", eventData.Message);  
            Console.WriteLine("Event.Active: {0}", eventData.Active);  
            Console.WriteLine("Event.Acknowledged: {0}",  
eventData.Acknowledged);  
        }  
    }  
}  
}
```

## VB.NET

---

' This example shows how to set the filtering criteria to be used for the event subscription.

```
Imports JetBrains.Annotations  
Imports OpcLabs.EasyOpc.AlarmsAndEvents  
Imports OpcLabs.EasyOpc.DataAccess  
Imports System.Threading  
  
Namespace _AESubscriptionFilter  
  
    Friend Class Properties  
        <NotNull()>_  
        Private Shared ReadOnly EasyAEClient As New EasyAEClient()  
        <NotNull()>_  
        Private Shared ReadOnly EasyDaClient As New EasyDAClient()
```

```

    Public Shared Sub Main()
        Dim eventHandler = New EasyAENotificationEventHandler(AddressOf
easyAEClient_Notification)
        AddHandler EasyAEClient.Notification, eventHandler

        Console.WriteLine("Processing event notifications...")
        Dim subscriptionFilter As New AESubscriptionFilter
        subscriptionFilter.Sources = New AENodeDescriptor()
        {"Simulation.ConditionState1", "Simulation.ConditionState3"}
        ' You can also filter using event types, categories, severity, and
areas.
        Dim handle As Integer = EasyAEClient.SubscribeEvents("",

"OPCLabs.KitEventServer.2", 1000, Nothing, subscriptionFilter)

        ' Allow time for initial refresh
        Thread.Sleep(5 * 1000)

        ' Set some events to active state.
        ' The activation below will come from a source contained in a filter and
the notification will arrive.
        EasyDaClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Activate", True)
        ' The activation below will come from a source that is not contained in
a filter and the notification will not arrive.
        EasyDaClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState2.Activate", True)

        Thread.Sleep(10 * 1000)

        EasyAEClient.UnsubscribeEvents(handle)
    End Sub

    ' Notification event handler
    Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)
        Console.WriteLine()
        Console.WriteLine("Refresh: {0}", e.Refresh)
        Console.WriteLine("RefreshComplete: {0}", e.RefreshComplete)
        If e.EventData IsNot Nothing Then
            Dim eventData As AEEventData = e.EventData
            Console.WriteLine("EventData.QualifiedSourceName: {0}",
eventData.QualifiedSourceName)
            Console.WriteLine("EventData.Message: {0}", eventData.Message)
            Console.WriteLine("EventData.Active: {0}", eventData.Active)
            Console.WriteLine("EventData.Acknowledged: {0}",
eventData.Acknowledged)
        End If
    End Sub
End Class
End Namespace

```

## VBScript

Rem This example shows how to set the filtering criteria to be used for the event subscription.

```
Option Explicit
```

```

Dim DAClient: Set DAClient = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim AEClient: Set AEClient =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
```

```
WScript.ConnectObject AEClient, "AEClient_"

WScript.Echo "Processing event notifications..."
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"
Dim SubscriptionFilter: Set SubscriptionFilter =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionFilter")
Dim SourceDescriptor1: Set SourceDescriptor1 =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor1.QualifiedName = "Simulation.ConditionState1"
Dim SourceDescriptor2: Set SourceDescriptor2 =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor2.QualifiedName = "Simulation.ConditionState3"
SubscriptionFilter.Sources = Array(SourceDescriptor1, SourceDescriptor2)
Rem You can also filter using event types, categories, severity, and areas.
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.Filter = SubscriptionFilter
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = AEClient.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

Rem Allow time for initial refresh
WScript.Sleep 5*1000

WScript.Echo "Set some events to active state..."
Rem The activation below will come from a source contained in a filter and the
notification will arrive.
DAClient.WriteItemValue "", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Activate", True
Rem The activation below will come from a source that is not contained in a filter
and the notification will not arrive.
DAClient.WriteItemValue "", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState2.Activate", True

WScript.Sleep 10*1000

WScript.Echo "Unsubscribing..."
AEClient.UnsubscribeEvents handle

Rem Notification event handler
Sub AEClient_Notification(Sender, e)
    WScript.Echo
    WScript.Echo "Refresh: " & e.Refresh
    WScript.Echo "RefreshComplete: " & e.RefreshComplete

    If Not (e.EventData Is Nothing) Then
        With e.EventData
            WScript.Echo "EventData.QualifiedSourceName: " & .QualifiedSourceName
            WScript.Echo "EventData.Message: " & .Message
            WScript.Echo "EventData.Active: " & .Active
            WScript.Echo "EventData.Acknowledged: " & .Acknowledged
        End With
    End If
End Sub
```

### 6.2.4.3 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).

#### C#

```
// This example shows how to subscribe to events display the event message with each
// notification. It also shows how to
// unsubscribe afterwards.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
    {

        class ChangeEventSubscription
        {
            public static void Main()
            {
                using (var easyAEClient = new EasyAEClient())
                {
                    var eventHandler = new
EasyAENotificationEventHandler(easyAEClient_Notification);
                    easyAEClient.Notification += eventHandler;

                    Console.WriteLine("Subscribing...");
                    int handle = easyAEClient.SubscribeEvents("",

"OPCLabs.KitEventServer.2", 500);

                    Console.WriteLine("Waiting for 10 seconds...");
                    Thread.Sleep(10 * 1000);

                    Console.WriteLine("Changing subscription...");
                    easyAEClient.ChangeEventSubscription(handle, 5 * 1000);

                    Console.WriteLine("Waiting for 50 seconds...");
                    Thread.Sleep(50 * 1000);

                    easyAEClient.UnsubscribeEvents(handle);
                }
            }

            // Notification event handler
            static void easyAEClient_Notification([NotNull] object sender, [NotNull]
EasyAENotificationEventArgs e)
            {
                if (e.EventData != null)
                    Console.WriteLine(e.EventData.Message);
            }
        }
    }
}
```

## VB.NET

```
' This example shows how to subscribe to events display the event message with each
' notification. It also shows how to
' unsubscribe afterwards.

Imports JetBrains.Annotations
Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports System.Threading

Namespace _EasyAEClient

    Friend Class ChangeEventSubscription
        Public Shared Sub Main()
            Using easyAEClient = New EasyAEClient()
                Dim eventHandler = New EasyAENotificationEventHandler(AddressOf
easyAEClient_Notification)
                    AddHandler easyAEClient.Notification, eventHandler

                    Console.WriteLine("Subscribing...")
                    Dim handle As Integer = easyAEClient.SubscribeEvents("",,
(OPCLabs.KitEventServer.2", 500)

                    Console.WriteLine("Waiting for 10 seconds...")
                    Thread.Sleep(10 * 1000)

                    Console.WriteLine("Changing subscription...")
                    easyAEClient.ChangeEventSubscription(handle, 5 * 1000)

                    Console.WriteLine("Waiting for 50 seconds...")
                    Thread.Sleep(50 * 1000)

                    easyAEClient.UnsubscribeEvents(handle)
            End Using
        End Sub

        ' Notification event handler
        Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)
            If e.EventData IsNot Nothing Then
                Console.WriteLine(e.EventData.Message)
            End If
        End Sub
    End Class
End Namespace
```

## VBScript

Rem This example shows how to change the notification rate of an existing subscription.

```
Option Explicit

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"
```

```
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.NotificationRate = 500
Dim handle: handle = Client.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Changing subscription..."
Client.ChangeEventSubscription handle, 5*1000, SubscriptionParameters.Filter, True

WScript.Echo "Waiting for 50 seconds..."
WScript.Sleep 50*1000

Client.UnsubscribeEvents handle

Rem Notification event handler
Sub Client_Notification(Sender, e)
    If Not e.EventData Is Nothing Then WScript.Echo e.EventData.Message
End Sub
```

## 6.2.4.4 Unsubscribing from OPC Events

### A single subscription

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.

#### C#

```
// This example shows how to unsubscribe from specific event notifications.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
    {

        class UnsubscribeEvents
        {
            public static void Main()
            {
                using (var easyAEClient = new EasyAEClient())
                {
                    var eventHandler = new
EasyAENotificationEventHandler(easyAEClient_Notification);
                    easyAEClient.Notification += eventHandler;

                    Console.WriteLine("Subscribing...");
                }
            }
        }
    }
}
```

```
        int handle = easyAEClient.SubscribeEvents("",  
"OPCLabs.KitEventServer.2", 1000);  
  
        Console.WriteLine("Waiting for 10 seconds...");  
        Thread.Sleep(10 * 1000);  
  
        Console.WriteLine("Unsubscribing...");  
        easyAEClient.UnsubscribeEvents(handle);  
  
        Console.WriteLine("Waiting for 10 seconds...");  
        Thread.Sleep(10 * 1000);  
    }  
}  
  
// Notification event handler  
static void easyAEClient_Notification([NotNull] object sender, [NotNull]  
EasyAENotificationEventArgs e)  
{  
    if (e.EventData != null)  
        Console.WriteLine(e.EventData.Message);  
}  
}  
}  
}
```

## VB.NET

```
' This example shows how to unsubscribe from specific event notifications.
```

```
Imports JetBrains.Annotations  
Imports OpcLabs.EasyOpc.AlarmsAndEvents  
Imports System.Threading  
  
Namespace _EasyAEClient  
  
    Friend Class UnsubscribeEvents  
        Public Shared Sub Main()  
            Using easyAEClient = New EasyAEClient()  
                Dim eventHandler = New EasyAENotificationEventHandler(AddressOf  
easyAEClient_Notification)  
                AddHandler easyAEClient.Notification, eventHandler  
  
                Console.WriteLine("Subscribing...")  
                Dim handle As Integer = easyAEClient.SubscribeEvents("",  
"OPCLabs.KitEventServer.2", 1000)  
  
                Console.WriteLine("Waiting for 10 seconds...")  
                Thread.Sleep(10 * 1000)  
  
                Console.WriteLine("Unsubscribing...")  
                easyAEClient.UnsubscribeEvents(handle)  
  
                Console.WriteLine("Waiting for 10 seconds...")  
                Thread.Sleep(10 * 1000)  
            End Using  
        End Sub  
  
        ' Notification event handler  
        Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As  
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)  
            If e.EventData IsNot Nothing Then
```

```
    Console.WriteLine(e.EventData.Message)
End If
End Sub
End Class
End Namespace
```

## VBScript

Rem This example shows how to unsubscribe from specific event notifications.

```
Option Explicit
```

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = Client.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

WScript.Echo "Unsubscribing..."
Client.UnsubscribeEvents handle

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000
```

```
Rem Notification event handler
Sub Client_Notification(Sender, e)
    If Not e.EventData Is Nothing Then WScript.Echo e.EventData.Message
End Sub
```

## All subscriptions

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

### C#

```
// This example shows how to unsubscribe from all event notifications.
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
```

```
{  
  
    class UnsubscribeAllEvents  
    {  
        public static void Main()  
        {  
            using (var easyAEClient = new EasyAEClient())  
            {  
                var eventHandler = new  
EasyAENotificationEventHandler(easyAEClient_Notification);  
                easyAEClient.Notification += eventHandler;  
  
                Console.WriteLine("Subscribing...");  
                easyAEClient.SubscribeEvents("", "OPCLabs.KitEventServer.2",  
1000);  
  
                Console.WriteLine("Waiting for 10 seconds...");  
                Thread.Sleep(10 * 1000);  
  
                Console.WriteLine("Unsubscribing...");  
                easyAEClient.UnsubscribeAllEvents();  
  
                Console.WriteLine("Waiting for 10 seconds...");  
                Thread.Sleep(10 * 1000);  
            }  
        }  
  
        // Notification event handler  
        static void easyAEClient_Notification([NotNull] object sender, [NotNull]  
EasyAENotificationEventArgs e)  
        {  
            if (e.EventData != null)  
                Console.WriteLine(e.EventData.Message);  
        }  
    }  
}
```

## VB.NET

```
' This example shows how to unsubscribe from all event notifications.  
  
Imports JetBrains.Annotations  
Imports OpcLabs.EasyOpc.AlarmsAndEvents  
Imports System.Threading  
  
Namespace _EasyAEClient  
  
    Friend Class UnsubscribeAllEvents  
        Public Shared Sub Main()  
            Using easyAEClient = New EasyAEClient()  
                Dim eventHandler = New EasyAENotificationEventHandler(AddressOf  
easyAEClient_Notification)  
                AddHandler easyAEClient.Notification, eventHandler  
  
                Console.WriteLine("Subscribing...")  
                easyAEClient.SubscribeEvents("", "OPCLabs.KitEventServer.2", 1000)  
  
                Console.WriteLine("Waiting for 10 seconds...")  
                Thread.Sleep(10 * 1000)  
            End Sub  
    End Class  
End Namespace
```

```
Console.WriteLine("Unsubscribing...")
easyAEClient.UnsubscribeAllEvents()

Console.WriteLine("Waiting for 10 seconds...")
Thread.Sleep(10 * 1000)
End Using
End Sub

' Notification event handler
Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)
    If e.EventData IsNot Nothing Then
        Console.WriteLine(e.EventData.Message)
    End If
End Sub
End Class
End Namespace
```

## VBScript

Rem This example shows how to unsubscribe from all event notifications.

```
Option Explicit
```

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.NotificationRate = 1000
Client.SubscribeEvents ServerDescriptor, SubscriptionParameters, True, Nothing

WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000
```

```
WScript.Echo "Unsubscribing..."
Client.UnsubscribeAllEvents
```

```
WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000
```

```
Rem Notification event handler
Sub Client_Notification(Sender, e)
    If Not e.EventData Is Nothing Then WScript.Echo e.EventData.Message
End Sub
```

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.

## 6.2.4.5 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.

## VBScript

```
Rem This example shows how to force a refresh for all active conditions and inactive,
unacknowledged conditions.
```

```
Option Explicit
```

```
Dim DAClient: Set DAClient = CreateObject("OpcLabs.EasyOpc.DataAccess.EasyDAClient")
Dim AEClient: Set AEClient =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
WScript.ConnectObject AEClient, "AEClient_"

WScript.Echo "Processing event notifications..."
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"
Dim SourceDescriptor1: Set SourceDescriptor1 =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor1.QualifiedName = "Simulation.ConditionState1"
Dim SourceDescriptor2: Set SourceDescriptor2 =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor2.QualifiedName = "Simulation.ConditionState1"
Dim SourceDescriptor3: Set SourceDescriptor3 =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AENodeDescriptor")
SourceDescriptor3.QualifiedName = "Simulation.ConditionState1"
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.Filter.Sources = Array(SourceDescriptor1, SourceDescriptor2,
SourceDescriptor3)
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = AEClient.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

Rem The component will perform auto-refresh at this point, give it time to happen
WScript.Echo "Waiting for 10 seconds..."
WScript.Sleep 10*1000

Rem Set some events to active state, which will cause them to appear in refresh
WScript.Echo "Activating conditions and waiting for 10 seconds..."
DAClient.WriteItemValue "", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Activate", True
DAClient.WriteItemValue "", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState2.Activate", True
WScript.Sleep 10*1000

WScript.Echo "Refreshing subscription and waiting for 10 seconds..."
AEClient.RefreshEventSubscription handle
WScript.Sleep 10*1000

AEClient.UnsubscribeEvents handle
```

```
Rem Notification event handler
Sub AEClient_Notification(Sender, e)
    WScript.Echo
    WScript.Echo "Refresh: " & e.Refresh
    WScript.Echo "RefreshComplete: " & e.RefreshComplete

    If Not (e.EventData Is Nothing) Then
        With e.EventData
            WScript.Echo "EventData.QualifiedSourceName: " & .QualifiedSourceName
            WScript.Echo "EventData.Message: " & .Message
            WScript.Echo "EventData.Active: " & .Active
            WScript.Echo "EventData.Acknowledged: " & .Acknowledged
        End With
    End If
End Sub
```

## 6.2.4.6 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 [EventData](#) 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 [EventData](#) 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 [EventData](#) 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 e.g. [Arguments](#) (including [Arguments.State](#)). Other properties, such as [EventData](#), contain null references when there is no associated information for them. When the [EventData](#) property is not a null reference, it contains an [AEEEventData](#) 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 [EventData](#) 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 [EventData](#) property is a null reference). If none of the previous applies, the [EventData](#) property contains a valid [AEEEventData](#) 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.EventData is null **then**

*Subscription has been successfully connected or re-connected, handle it (or ignore)*

**else**

*Handle the OPC event, details are in notificationEventArgs.EventData. 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".

## C#

---

```
// This example shows how to subscribe to events with specified event attributes,
// and obtain the attribute values in event
// notifications.
using System;
using System.Collections.Generic;
using System.Threading;
using JetBrains.Annotations;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using OpcLabs.EasyOpc.DataAccess;

namespace DocExamples
{
    namespace _EasyAENotificationEventArgs
    {
        class AttributeValues
        {
            public static void Main()
            {
                var easyAEClient = new EasyAEClient();
                var easyDAClient = new EasyDAClient();

                var eventHandler = new
                    EasyAENotificationEventHandler(easyAEClient_Notification);
                easyAEClient.Notification += eventHandler;

                // Inactivate the event condition (we will later activate it and
                // receive the notification)
                easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
                    "SimulateEvents.ConditionState1.Inactivate", true);

                var subscriptionFilter = new AESubscriptionFilter
                {
                    Sources = new AENodeDescriptor[] {"Simulation.ConditionState1"}
                };

                // Prepare a dictionary holding requested event attributes for each
                // event category
                // The event category IDs and event attribute IDs are hard-coded
                // here, but can be obtained from the OPC
                // server by querying as well.
                var returnedAttributesByCategory = new AEAttributeSetDictionary();
                returnedAttributesByCategory[0x00ECFF02] = new long[] {0x00EB0003,
                    0x00EB0008};

                Console.WriteLine("Subscribing to events...");
                int handle = easyAEClient.SubscribeEvents("", "OPCLabs.KitEventServer.2", 1000, null, subscriptionFilter,
                    returnedAttributesByCategory);
```

# VB.NET

```
' This example shows how to subscribe to events with specified event attributes, and  
' obtain the attribute values in event  
' notifications.
```

```
Imports System.Threading
Imports JetBrains.Annotations
Imports OpcLabs.EasyOpc.AlarmsAndEvents
Imports OpcLabs.EasyOpc.DataAccess

Namespace _EasyAENotificationEventArgs
    Friend Class AttributeValues
        Public Shared Sub Main()
            Dim easyAEClient = New EasyAEClient()
            Dim easyDAClient = New EasyDAClient()

            Dim eventHandler = New EasyAENotificationEventHandler(AddressOf
easyAEClient_Notification)
            AddHandler easyAEClient.Notification, eventHandler
```

```
' Inactivate the event condition (we will later activate it and receive
the notification)
easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Inactivate", True)

Dim subscriptionFilter As New AESubscriptionFilter
subscriptionFilter.Sources = New AENodeDescriptor()
{"Simulation.ConditionState1"}

' Prepare a dictionary holding requested event attributes for each event
category
' The event category IDs and event attribute IDs are hard-coded here,
but can be obtained from the OPC
' server by querying as well.
Dim returnedAttributesByCategory = New AEAttributeSetDictionary()
returnedAttributesByCategory(&HECFF02) = New Long() {&HEB0003, &HEB0008}

Console.WriteLine("Subscribing to events...")
Dim handle As Integer = easyAEClient.SubscribeEvents("", 
"OPCLabs.KitEventServer.2", 1000, Nothing, subscriptionFilter,
returnedAttributesByCategory)

' Give the refresh operation time to complete
Thread.Sleep(5 * 1000)

' Trigger an event carrying specified attributes (activate the
condition)
easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.AttributeValues.15400963", 123456)
easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.AttributeValues.15400968", "Some string value")
easyDAClient.WriteItemValue("", "OPCLabs.KitServer.2",
"SimulateEvents.ConditionState1.Activate", True)

Console.WriteLine("Processing event notifications for 10 seconds...")
Thread.Sleep(10 * 1000)

easyAEClient.UnsubscribeEvents(handle)
End Sub

' Notification event handler
Private Shared Sub easyAEClient_Notification(<NotNull()> ByVal sender As
Object, <NotNull()> ByVal e As EasyAENotificationEventArgs)
    If (Not e.Refresh) AndAlso (e.EventData IsNot Nothing) Then
        ' Display all received event attribute IDs and their corresponding
values
        Console.WriteLine("Event attribute count: {0}",
e.EventData.AttributeValues.Count)
        For Each pair As KeyValuePair(Of Long, Object) In
e.EventData.AttributeValues
            Console.WriteLine("      {0}: {1}", pair.Key, pair.Value)
        Next pair
    End If
End Sub
End Class
End Namespace
```

## VBScript

Rem This example subscribe to events, and displays rich information available with

each event notification.

## Option Explicit

```
Dim ServerDescriptor: Set ServerDescriptor =
CreateObject("OpcLabs.EasyOpc.ServerDescriptor")
ServerDescriptor.ServerClass = "OPCLabs.KitEventServer.2"

Dim Client: Set Client =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.EasyAEClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Dim SubscriptionParameters: Set SubscriptionParameters =
CreateObject("OpcLabs.EasyOpc.AlarmsAndEvents.AESubscriptionParameters")
SubscriptionParameters.NotificationRate = 1000
Dim handle: handle = Client.SubscribeEvents(ServerDescriptor,
SubscriptionParameters, True, Nothing)

WScript.Echo "Processing event notifications for 1 minute..."
WScript.Sleep 60*1000

Client.UnsubscribeEvents handle

Rem Notification event handler
Sub Client_Notification(Sender, e)
On Error Resume Next
WScript.Echo
WScript.Echo "e.Exception.Message: " & e.Exception.Message
WScript.Echo "e.Exception.Source: " & e.Exception.Source
WScript.Echo "e.Exception.ErrorCode: " & e.Exception.ErrorCode
WScript.Echo "e.Arguments.State: " & e.Arguments.State
WScript.Echo "e.Arguments.ServerDescriptor.MachineName: " &
e.Arguments.ServerDescriptor.MachineName
WScript.Echo "e.Arguments.ServerDescriptor.ServerClass: " &
e.Arguments.ServerDescriptor.ServerClass
WScript.Echo "e.Arguments.SubscriptionParameters.Active: " &
e.Arguments.SubscriptionParameters.Active
WScript.Echo "e.Arguments.SubscriptionParameters.NotificationRate: " &
e.Arguments.SubscriptionParameters.NotificationRate
Rem IMPROVE: Display Arguments.SubscriptionParameters.Filter details
WScript.Echo "e.Arguments.SubscriptionParameters.Filter: " &
e.Arguments.SubscriptionParameters.Filter
Rem IMPROVE: Display
Arguments.SubscriptionParameters.ReturnedAttributesByCategory details
WScript.Echo "e.Arguments.SubscriptionParameters.ReturnedAttributesByCategory: " &
e.Arguments.SubscriptionParameters.ReturnedAttributesByCategory
WScript.Echo "e.Refresh: " & e.Refresh
WScript.Echo "e.RefreshComplete: " & e.RefreshComplete
WScript.Echo "e.EnabledChanged: " & e.EnabledChanged
WScript.Echo "e.ActiveChanged: " & e.ActiveChanged
WScript.Echo "e.AcknowledgedChanged: " & e.AcknowledgedChanged
WScript.Echo "e.QualityChanged: " & e.QualityChanged
WScript.Echo "e.SeverityChanged: " & e.SeverityChanged
WScript.Echo "e.SubconditionChanged: " & e.SubconditionChanged
WScript.Echo "e.MessageChanged: " & e.MessageChanged
WScript.Echo "e.AttributeChanged: " & e.AttributeChanged
WScript.Echo "e.EventData.QualifiedSourceName: " &
e.EventData.QualifiedSourceName
WScript.Echo "e.EventData.Time: " & e.EventData.Time
```

```
WScript.Echo "e.EventData.TimeLocal: " & e.EventData.TimeLocal
WScript.Echo "e.EventData.Message: " & e.EventData.Message
WScript.Echo "e.EventData.EventType: " & e.EventData.EventType
WScript.Echo "e.EventData.CategoryId: " & e.EventData.CategoryId
WScript.Echo "e.EventData.Severity: " & e.EventData.Severity
Rem IMPROVE: Display EventData.AttributeValues details
WScript.Echo "e.EventData.AttributeValues: " & e.EventData.AttributeValues
WScript.Echo "e.EventData.ConditionName: " & e.EventData.ConditionName
WScript.Echo "e.EventData.SubconditionName: " & e.EventData.SubconditionName
WScript.Echo "e.EventData.Enabled: " & e.EventData.Enabled
WScript.Echo "e.EventData.Active: " & e.EventData.Active
WScript.Echo "e.EventData.Acknowledged: " & e.EventData.Acknowledged
WScript.Echo "e.EventData.Quality: " & e.EventData.Quality
WScript.Echo "e.EventData.AcknowledgeRequired: " &
e.EventData.AcknowledgeRequired
WScript.Echo "e.EventData.ActiveTime: " & e.EventData.ActiveTime
WScript.Echo "e.EventData.ActiveTimeLocal: " & e.EventData.ActiveTimeLocal
WScript.Echo "e.EventData.Cookie: " & e.EventData.Cookie
WScript.Echo "e.EventData.ActorId: " & e.EventData.ActorId
End Sub
```

## 6.2.4.7 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.

### C#

```
// This example shows how to subscribe to events and display the event message with
// each notification, using a callback method
// specified using lambda expression.

using System.Diagnostics;
using OpcLabs.EasyOpc.AlarmsAndEvents;
using System;
using System.Threading;

namespace DocExamples
{
    namespace _EasyAEClient
    {
        partial class SubscribeEvents
        {
            public static void CallbackLambda()
            {
                // Instantiate the client object
                var easyAEClient = new EasyAEClient();

                Console.WriteLine("Subscribing...");
                // The callback is a lambda expression the displays the event
message
                easyAEClient.SubscribeEvents("", "OPCLabs.KitEventServer.2", 1000,
                    (sender, eventArgs) =>
                {
                    Debug.Assert(eventArgs != null);
                    if (eventArgs.EventData != null)
```

```
        Console.WriteLine(eventArgs.EventData.Message);
    });

    Console.WriteLine("Processing event notifications for 20
seconds...");
    Thread.Sleep(20 * 1000);

    Console.WriteLine("Unsubscribing...");
    easyAEClient.UnsubscribeAllEvents();

    Console.WriteLine("Waiting for 2 seconds...");
    Thread.Sleep(2 * 1000);
}

}

}
```

## 6.2.5 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.

Following are instance properties, i.e. if you have created multiple [EasyAEClient](#) object, each will have its own copy of them:

- [Mode](#): 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, your code can set different values to these properties 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.

Please use the Reference documentation for details on meaning of various properties and their use.

## 6.3 Procedural Coding Model for OPC UA Alarms & Conditions

This chapter gives you guidance in how to implement the common tasks that are needed when dealing with OPC UA

Alarms & Conditions vents server from the client side. You achieve these tasks by calling methods on the EasyUAClient object (the same object as for OPC UA Data Access).

Note: Functionality in this chapter that has to do specifically with Alarms & Conditions is described in Part 9 of the OPC Unified Architecture Specification, however many related and relevant pieces are scattered around other parts of the specification as well.

## 6.3.1 Obtaining Information

For obtaining information related to Alarms & Conditions (such as the state of conditions), OPC UA does not introduce any new mechanism over what we have already described for OPC Data (in OPC UA). Please refer to the "Obtaining Information" under "Procedural Coding Model for OPC Data (Class and UA)" for all necessary information.

Methods (and extension methods) such as [EasyUAClient.Read](#), [ReadMultiple](#), [ReadValue](#), [ReadMultipleValues](#) can be used.

## 6.3.2 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.

The operations performed by the methods described here are actually implemented inside the OPC server. The OPC specification defines OPC UA methods for that, and as such, you could use e.g. one of the [EasyUAClient.CallMethod](#) overloads in order to achieve the same outcome. You would have to, however, look up and specify the Node ID of the desired method, and properly assemble the array of input arguments (with proper types), and so on. The methods described below make it easier, and do the boring part for you. They are all implemented as extension methods on the [IEasyUAClient](#) interface which means that (at least in languages such as C# or VB.NET) you can use them as if they were normal methods on the [EasyUAClient](#) object.

### 6.3.2.1 Conditions - Disabling/Enabling, and Applying Comments

In order to change a condition instance to the Disabled state, use the [IEasyUAClient.Disable](#) extension method.

In order to change a condition instance to the enabled state, use the [IEasyUAClient.Enable](#) extension method.

In order to apply a comment to a specific state of a condition instance, use the [IEasyUAClient.AddComment](#) extension method.

### 6.3.2.2 Dialogs - Responding

In order to respond to pass the selected response option and end the dialog, use the [IEasyUAClient.Respond](#) extension method.

### 6.3.2.3 Acknowledgeable Conditions – Acknowledging and Confirmation

In order to acknowledge an event notification for a condition instance state (where AckedState is FALSE), use the

[IEasyUAClient.Acknowledge](#) extension method.

## C#

```
// This example shows how to acknowledge an event.
using System;
using System.Threading;
using OpcLabs.EasyOpc.UA;
using OpcLabs.EasyOpc.UA.AddressSpace;
using OpcLabs.EasyOpc.UA.AlarmsAndConditions;
using OpcLabs.EasyOpc.UA.AlarmsAndConditions.Extensions;
using OpcLabs.EasyOpc.UA.Filtering;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        class Acknowledge
        {
            public static void Main()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                UANodeId nodeId = null;
                byte[] eventId = null;
                var anEvent = new ManualResetEvent(initialState: false);

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeEvent(
                    "opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer",
                    UAObjectIds.Server,
                    1000,
                    new UAEventFilterBuilder(
                        UAFilterElements.Equals(
                            UABaseEventObject.Operands.NodeId,
                            new UANodeId(expandedText:
                            "nsu=http://opcfoundation.org/Quickstarts/AlarmCondition;ns=2;s=1:Colours/EastTank?
Yellow")),
                    UABaseEventObject.AllFields),
                    (sender, eventArgs) =>
                {
                    if (!eventArgs.Succeeded)
                    {
                        Console.WriteLine(eventArgs.ErrorMessageBrief);
                        return;
                    }
                    if (eventArgs.EventData != null)
                    {
                        UABaseEventObject baseEventObject =
eventArgs.EventData.BaseEvent;
                        Console.WriteLine(baseEventObject);

                        // Make sure we do not catch the event more than once
                        if (anEvent.WaitOne(0))
                            return;

                        nodeId = baseEventObject.NodeId;
                        eventId = baseEventObject.EventId;
                    }
                });
            }
        }
    }
}
```

```
        anEvent.Set();
    }
},
state:null);
Console.WriteLine("Waiting for an event for 30 seconds...");
if (!anEvent.WaitOne(30*1000))
{
    Console.WriteLine("Event not received");
    return;
}

Console.WriteLine("Acknowledging an event...");
easyUAClient.Acknowledge(
    "opc.tcp://opcuademo-
this.com:62544/Quickstarts/AlarmConditionServer",
    nodeId,
    eventId,
    "Acknowledged by an automated example code");

Console.WriteLine("Waiting for 5 seconds...");
Thread.Sleep(5 * 1000);

Console.WriteLine("Unsubscribing...");
easyUAClient.UnsubscribeAllMonitoredItems();

Console.WriteLine("Waiting for 5 seconds...");
Thread.Sleep(5 * 1000);
}
}
}
}
```

In order to confirm an event notification for a condition instance state (where ConfirmedState is FALSE), use the [IEasyUAClient.Confirm](#) extension method.

## 6.3.2.4 Alarms – Shelving and Unshelving

In order to set the alarm condition to the Unshelved state, use the [IEasyUAClient.Unshelve](#) extension method.

In order to set the alarm condition to the TimedShelved state, use the [IEasyUAClient.TimedShelve](#) extension method.

In order to set the alarm condition to the OneShotShelved state, use the [IEasyUAClient.OneShotShelve](#) extension method.

## 6.3.3 Browsing for Information

QuickOPC contains methods that allow your application to retrieve and enumerate information about OPC UA Alarm & Conditions 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).

### 6.3.3.1 Discovering OPC UA Servers

Please refer to the “Discovering OPC UA Servers” chapter under “Procedural Coding Model for OPC Data (Class and

UA)" for all necessary information. OPC Unified Architecture, as the name suggests, allows to treat multiple functionality areas commonly, and the same OPC Servers that provide Data Access functionality are (or can be) also Alarms & Conditions servers.

## 6.3.3.2 Browsing for Event Sources

In order to browse for event sources, you can use the pre-made browse parameters available in the [UABrowseParameters.EventSources](#) static property, and call the [IEasyUAClient.BrowseNodes](#) method with them. Alternatively, you can use a dedicated extension method, [IEasyUAClient.BrowseEventSources](#), to perform browsing for event sources easily.

## 6.3.3.3 Browsing for Notifiers

In order to browse for notifiers, you can use pre-made browse parameters available in the [UABrowseParameters.Notifiers](#) static property, and call the [IEasyUAClient.BrowseNodes](#) method. Alternatively, you can use a dedicated extension method, [IEasyUAClient.BrowseNotifiers](#), to perform browsing for notifiers easily.

## 6.3.4 Subscribing to Information

If your application needs to be informed about events occurring in the process and provided by the OPC UA Alarms & Conditions 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.

### 6.3.4.1 Subscribing to OPC Events

Generally speaking, you subscribe to OPC UA Events similarly to data changes, but by using monitoring parameters that contain an event filter instead of data change filter. Besides the [UAMonitoringParameters.DataChangeFilter](#) property, the [UAMonitoringParameters](#) class also has an [EventFilter](#) property (of [UAEventFilter](#) type). The developer can specify either a data change filter, or an event filter, with monitoring parameters.

As the [UAMonitoringParameters](#) are part of the information that gets passed (directly or indirectly) to any [EasyUAClient.SubscribeMonitoredItem](#) or [SubscribeMultipleMonitoredItems](#) call, the introduction of the [EventFilter](#) property and the [EventNotification](#) event (both described above) are, in fact, the only core features that already in itself allow the Alarms & Conditions support to work. Many other features simply provide better developer experience above this.

#### A single event

Instead of putting together the monitoring parameters with an event filter, you can subscribe to events a bit easier using the [IEasyUAClient.SubscribeEvent](#) extension method, with several overloads. Similarly to the [IEasyUAClient.SubscribeDataChange](#) extension method (for data changes), the [SubscribeEvent](#) method provides a simplified way to subscribe to events.

##### C#

```
// This example shows how to subscribe to event notifications and display each incoming event.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class SubscribeEvent
        {
```

```
public static void Overload1()
{
    // Instantiate the client object and hook events
    var easyUAClient = new EasyUAClient();
    easyUAClient.EventNotification += easyUAClient_EventNotification;

    Console.WriteLine("Subscribing...");
    easyUAClient.SubscribeEvent(
        "opc.tcp://opcuademo-this.com:62544/Quickstarts/AlarmConditionServer",
        UAObjectIds.Server,
        1000);

    Console.WriteLine("Processing event notifications for 30 seconds...");
    System.Threading.Thread.Sleep(30 * 1000);

    Console.WriteLine("Unsubscribing...");
    easyUAClient.UnsubscribeAllMonitoredItems();

    Console.WriteLine("Waiting for 5 seconds...");
    System.Threading.Thread.Sleep(5 * 1000);
}

static void easyUAClient_EventNotification(object sender,
EasyUAEEventNotificationEventArgs e)
{
    // Display the event
    Console.WriteLine(e);
}
}
```

## VB.NET

```
' This example shows how to subscribe to event notifications and display each incoming event.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.AddressSpace
Imports OpcLabs.EasyOpc.UA.OperationModel

Namespace _EasyUAClient
    Friend Class SubscribeEvent
        Public Shared Sub Overload1()
            ' Instantiate the client object and hook events
            Dim easyUAClient = New EasyUAClient()
            AddHandler easyUAClient.EventNotification, AddressOf
easyUAClient_EventNotification

            Console.WriteLine("Subscribing...")
            easyUAClient.SubscribeEvent(
                "opc.tcp://opcuademo-this.com:62544/Quickstarts/AlarmConditionServer", _
                UAObjectIds.Server, _
                1000)

            Console.WriteLine("Processing event notifications for 10 seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Unsubscribing...")
            easyUAClient.UnsubscribeAllMonitoredItems()

            Console.WriteLine("Waiting for 5 seconds...")
            Threading.Thread.Sleep(5 * 1000)
        End Sub

        Private Shared Sub easyUAClient_EventNotification(ByVal sender As Object, ByVal e As
EasyUAEEventNotificationEventArgs)
            ' Display the event
            Console.WriteLine(e)
        End Sub
    End Class
End Namespace
```

```
End Namespace
```

## Visual Basic (VB 6.)

Rem This example shows how to subscribe to event notifications and display each incoming event.

```
' The client object, with events
'Public WithEvents Client1 As EasyUAClient

Private Sub SubscribeEvent_Main_Command_Click()
    OutputText = ""

    Set Client1 = New EasyUAClient

    OutputText = OutputText & "Subscribing..." & vbCrLf
    Call Client1.SubscribeEvent("opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer", "nsu=http://opcfoundation.org/UA;/i=2253",
1000)

    OutputText = OutputText & "Processing event notifications for 30 seconds..." & vbCrLf
    Pause 30000

    OutputText = OutputText & "Unsubscribing..." & vbCrLf
    Client1.UnsubscribeAllMonitoredItems

    OutputText = OutputText & "Waiting for 5 seconds..." & vbCrLf
    Pause 5000

    Set Client1 = Nothing
End Sub

Private Sub Client1_EventNotification(ByVal sender As Variant, ByVal eventArgs As
EasyUAEventNotificationEventArgs)
    ' Display the event
    OutputText = OutputText & eventArgs & vbCrLf
End Sub
```

## VBScript

Rem This example shows how to subscribe to event notifications and display each incoming event.

```
Option Explicit

Const UAObjectIds_Server = "nsu=http://opcfoundation.org/UA;/i=2253"

' Instantiate the client object and hook events
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
WScript.ConnectObject Client, "Client_"

WScript.Echo "Subscribing..."
Client.SubscribeEvent "opc.tcp://opcua.demo-this.com:62544/Quickstarts/AlarmConditionServer",
UAObjectIds_Server, 1000

WScript.Echo "Processing event notifications for 30 seconds..."
WScript.Sleep 30*1000

Sub Client_EventNotification(Sender, e)
    ' Display the event
    WScript.Echo e
End Sub
```

## Multiple events

Multiple events are more efficiently subscribed using the [SubscribeMultipleMonitoredItems](#) method:

## C#

```
// This example shows how to subscribe to multiple events.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class SubscribeMultipleMonitoredItems
        {
            public static void Events()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.EventNotification += easyUAClient_EventNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeMultipleMonitoredItems(new []
                {
                    new EasyUAMonitoredItemArguments("firstState",
                        "opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer",
                        UAObjectIds.Server,
                        new UAMonitoringParameters(1000, new UAEEventFilterBuilder(
UAFilterElements.GreaterThanOrEqualTo(UABaseEventObject.Operands.Severity, 500),
                        UABaseEventObject.AllFields))
                    { AttributeId = UAAttributeId.EventNotifier },
                    new EasyUAMonitoredItemArguments("secondState",
                        "opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer",
                        UAObjectIds.Server,
                        new UAMonitoringParameters(2000, new UAEEventFilterBuilder(
UAFilterElements.Equals(
                        UABaseEventObject.Operands.SourceNode,
                        new
UANodeId("nsu=http://opcfoundation.org/Quickstarts/AlarmCondition;ns=2;s=1:Metals/SouthMotor"))),
                        UABaseEventObject.AllFields)))
                    { AttributeId = UAAttributeId.EventNotifier },
                });
                Console.WriteLine("Processing event notifications for 30 seconds...");
                System.Threading.Thread.Sleep(30 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeAllMonitoredItems();

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }

            static void easyUAClient_EventNotification(object sender,
EasyUAEventArgs e)
            {
                // Display the event
                Console.WriteLine(e);
            }
        }
    }
}
```

## 6.3.4.2 Specifying Event Filters

The [UAEventFilter](#) object (used for specifying the event filters) is probably the most complicated part of the OPC UA Alarms & Condition features. It has two main parts: Select clauses (contained in the [SelectClauses](#) property, of [UAAttributeFieldCollection](#) type), and a Where clause (contained in the [WhereClause](#) property, of [UAContentFilterElement](#) type). The Select clauses contain a collection of the attribute fields to return with each event in a notification. The Where clause contains the criteria limiting the notifications.

In order to make the creation of some event filters appear shorter in the code, a [UAEventFilterBuilder](#) class is provided. Instances of this builder can be implicitly converted to the event filter itself (hence you can create just the [UAEventFilterBuilder](#) objects, and use them at all places where the [UAEventFilter](#) object is expected). The builder contains a Where clause, and is a collection of Select clauses. You can therefore use the C# collection initializer to specify the Select clauses.

In COM (tools like VB6, Delphi etc.), creation of OPC UA Event Filters requires longer code. For related information, see [http://kb.quickopc.com/Creating\\_an\\_OPc\\_UA\\_event\\_filter\\_in\\_COM](http://kb.quickopc.com/Creating_an_OPc_UA_event_filter_in_COM).

### 6.3.4.2.1 The Select clauses

Each Select clause of the event filter is a [UAAttributeField](#) object, which contains an [Operand](#) property. You can optionally set its the [State](#) property to an object of your choice, and the same object will then be made available to you in the event notification containing the operand. An [Operand](#) is of the [UASimpleAttributeOperand](#) type, and specifies primarily a [Typeld](#) node in the server, and a [QualifiedNames](#) collection (of [UAQualifiedNameCollection](#) type), which is a so-called simple relative path (inside that type) that leads to the required node. A simple relative path can only contain forward, "any hierarchical" references.

Conversion methods, and conversion operators exist between the [UAQualifiedNameCollection](#) (for simple relative paths) and [UABrowsePathElementCollection](#) (for general relative paths). Conversion from [UAQualifiedNameCollection](#) to [UABrowsePathElementCollection](#) is always possible and has an implicit conversion operator, conversion from [UABrowsePathElementCollection](#) to [UAQualifiedNameCollection](#) is only possible for simple relative path, and has an explicit conversion operator instead.

In order to make the specification of the most commonly used Select clauses easier, it is possible to use pre-defined clauses provided in the static [UABaseEventObject.Operands](#) class (for example, [UABaseEventObject.Operands.Message](#)). The [UABaseEventObject.AllFields](#) property can be used to obtain Select clauses for all fields of the UA base event type at once.

#### C#

---

```
// This example shows how to select fields for event notifications.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _UAEVENTFILTER
    {
        class SelectClauses
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.EventNotification += easyUAClient_EventNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeEvent(
                    "opc.tcp://opcua.demo-this.com:62544/Quickstarts/AlarmConditionServer",
                    UAObjectIds.Server,
                    1000,
                    new UAAttributeFieldCollection
                    {
                        // Select specific fields using standard operand symbols
                        UABaseEventObject.Operands.NodeId,
                        UABaseEventObject.Operands.SourceNode,
                        UABaseEventObject.Operands.SourceName,
                        UABaseEventObject.Operands.Time,

                        // Select specific fields using an event type ID and a simple relative path
                        UAFilterElements.SimpleAttribute(UAOBJECTIDS.BaseEventType, "/Message"),
                        UAFilterElements.SimpleAttribute(UAOBJECTIDS.BaseEventType, "/Severity"),
                    });
                Console.WriteLine("Processing event notifications for 30 seconds...");
            }
        }
    }
}
```

```

        System.Threading.Thread.Sleep(30 * 1000);

        Console.WriteLine("Unsubscribing...");
        easyUAClient.UnsubscribeAllMonitoredItems();

        Console.WriteLine("Waiting for 5 seconds...");
        System.Threading.Thread.Sleep(5 * 1000);
    }

    static void easyUAClient_EventNotification(object sender, EasyUAEEventNotificationEventArgs e)
    {
        Console.WriteLine();

        // Display the event
        if (e.EventData == null)
        {
            Console.WriteLine(e);
            return;
        }
        Console.WriteLine("All fields:");
        foreach (KeyValuePair<UAAttributeField, ValueResult> pair in e.EventData.FieldResults)
        {
            UAAttributeField attributeField = pair.Key;
            ValueResult valueResult = pair.Value;
            Console.WriteLine(" {0} -> {1}", attributeField, valueResult);
        }
        // Extracting a specific field using a standard operand symbol
        Console.WriteLine("Source name: {0}",
            e.EventData.FieldResults[UABaseEventObject.Operands.SourceName]);
        // Extracting a specific field using an event type ID and a simple relative path
        Console.WriteLine("Message: {0}",
            e.EventData.FieldResults[UAFilterElements.SimpleAttribute(UAObjectTypeIds.BaseEventType,
"/Message")));
    }
}
}

```

## Free Pascal

---

```

// This example shows how to select fields for event notifications.

type
  TClientEventHandlers4 = class
    procedure OnEventNotification(
      Sender: TObject;
      sender0: OleVariant;
      eventArgs: _EasyUAEEventNotificationEventArgs);
  end;

procedure TClientEventHandlers4.OnEventNotification(
  Sender: TObject;
  sender0: OleVariant;
  eventArgs: _EasyUAEEventNotificationEventArgs);

function ToUAAttributeField(Operand: UASimpleAttributeOperand): UAAttributeField;
var
  AttributeField: UAAttributeField;
begin
  AttributeField := CoUAAttributeField.Create;
  AttributeField.Operand := Operand;
  ToUAAttributeField := AttributeField;
end;

function UAFilterElements_SimpleAttribute(TypeId: UANodeId; SimpleRelativeBrowsePathString: string): UASimpleAttributeOperand;
var
  Operand: UASimpleAttributeOperand;
  BrowsePathParser: UABrowsePathParser;
begin
  BrowsePathParser := CoUABrowsePathParser.Create;
  Operand := CoUASimpleAttributeOperand.Create;
  Operand.TypeId.NodeId := TypeId;
  Operand.QualifiedNames :=
BrowsePathParser.ParseRelative(SimpleRelativeBrowsePathString).ToUAQualifiedNameCollection;
  UAFilterElements_SimpleAttribute := Operand;
end;

function ObjectTypeIds_BaseEventType: UANodeId;
var
  NodeId: UANodeId;

```

```

begin
    NodeId := CoUANodeId.Create;
    NodeId.StandardName := 'BaseEventType';
    ObjectTypeIds_BaseEventType := NodeId;
end;

var
    Count: Cardinal;
    Element: OleVariant;
    EntryEnumerator: IEnumVariant;
    Entry: DictionaryEntry2;
    AttributeField: UAAttributeField;
    AValueResult: ValueResult;
begin
    begin
        WriteLn;

        // Display the event
        if eventArgs.EventData = nil then
            begin
                WriteLn(eventArgs.ToString);
                Exit;
            end;
        WriteLn('All fields:');
        EntryEnumerator := eventArgs.EventData.FieldResults.GetEnumerator;
        while EntryEnumerator.Next(1, Element, Count) = S_OK do
            begin
                Entry := IUnknown(Element) as DictionaryEntry2;
                AttributeField := IUnknown(Entry.key) as UAAttributeField;
                AValueResult := IUnknown(Entry.Value) as ValueResult;
                WriteLn(' ', AttributeField.ToString, ' -> ', AValueResult.ToString);
            end;
        // Extracting a specific field using an event type ID and a simple relative path
        WriteLn('Source name: ', 

eventArgs.EventData.FieldResults[ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
    '/SourceName'))].ToString);
        WriteLn('Message: ', 

eventArgs.EventData.FieldResults[ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
    '/Message'))].ToString);
    end;

class procedure SelectClauses.Main;

    function ToUAAttributeField(Operand: UASimpleAttributeOperand): UAAttributeField;
    var
        AttributeField: UAAttributeField;
    begin
        AttributeField := CoUAAAttributeField.Create;
        AttributeField.Operand := Operand;
        ToUAAttributeField := AttributeField;
    end;

    function ObjectTypeIds_BaseEventType: UANodeId;
    var
        NodeId: UANodeId;
    begin
        NodeId := CoUANodeId.Create;
        NodeId.StandardName := 'BaseEventType';
        ObjectTypeIds_BaseEventType := NodeId;
    end;

    function UABaseEventObject_Operands_NodeId: UASimpleAttributeOperand;
    var
        Operand: UASimpleAttributeOperand;
    begin
        Operand := CoUASimpleAttributeOperand.Create;
        Operand.TypeId.NodeId.StandardName := 'BaseEventType';
        Operand.AttributeId := UAAttributeId_NodeId;
        UABaseEventObject_Operands_NodeId := Operand;
    end;

    function UAFilterElements_SimpleAttribute(TypeId: UANodeId; SimpleRelativeBrowsePathString: string): 
UASimpleAttributeOperand;
    var
        Operand: UASimpleAttributeOperand;
        BrowsePathParser: UABrowsePathParser;
    begin
        BrowsePathParser := CoUABrowsePathParser.Create;
        Operand := CoUASimpleAttributeOperand.Create;
        Operand.TypeId.NodeId := TypeId;
    end;

```

```

Operand.QualifiedNames :=
BrowsePathParser.ParseRelative(SimpleRelativeBrowsePathString).ToUAQualifiedNameCollection;
UAFilterElements_SimpleAttribute := Operand;
end;

var
Arguments: OleVariant;
EvsClient: TEvsEasyUAClient;
Client: EasyUAClient;
ClientEventHandlers: TClientEventHandlers4;
Handle: Cardinal;
HandleArray: OleVariant;
MonitoredItemArguments: EasyUAMonitoredItemArguments;
MonitoringParameters: UAMonitoringParameters;
EventFilter: UAEEventFilter;
SelectClauses: UAAttributeFieldCollection;
begin
// Instantiate the client object and hook events
EvsClient := TEvsEasyUAClient.Create(nil);
Client := EvsClient.ComServer;
ClientEventHandlers := TClientEventHandlers4.Create;
EvsClient.OnEventNotification := @ClientEventHandlers.OnEventNotification;

WriteLn('Subscribing...');

SelectClauses := CoUAAttributeFieldCollection.Create;
// Select specific fields using an event type ID and a simple relative path
SelectClauses.Add(ToUAAttributeField(UABaseEventObject_Operands_NodeId));
SelectClauses.Add(ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
'/SourceNode')));
SelectClauses.Add(ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
'/SourceName')));
SelectClauses.Add(ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/Time')));
SelectClauses.Add(ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
'/Message')));
SelectClauses.Add(ToUAAttributeField(UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
'/Severity')));

EventFilter := CoUAEEventFilter.Create;
EventFilter.SelectClauses := SelectClauses;

MonitoringParameters := CoUAMonitoringParameters.Create;
MonitoringParameters.SamplingInterval := 1000;
MonitoringParameters.EventFilter := EventFilter;
MonitoringParameters.QueueSize := 1000;

MonitoredItemArguments := CoEasyUAMonitoredItemArguments.Create;
MonitoredItemArguments.EndpointDescriptor.UrlString := 'opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer';
MonitoredItemArguments.NodeDescriptor.NodeId.StandardName := 'Server';
MonitoredItemArguments.MonitoringParameters := MonitoringParameters;
//MonitoredItemArguments.SubscriptionParameters.PublishingInterval := 0;
MonitoredItemArguments.AttributeId := UAAttributeId_EventNotifier;

Arguments := VarArrayCreate([0, 0], varVariant);
Arguments[0] := MonitoredItemArguments;

TVarData(HandleArray).VType := varArray or varVariant;
TVarData(HandleArray).VArray := PVarArray(
Client.SubscribeMultipleMonitoredItems(PSafeArray(TVarData(Arguments).VArray)));

WriteLn('Processing event notifications for 30 seconds...');
PumpSleep(30*1000);

WriteLn('Unsubscribing...');
Client.UnsubscribeAllMonitoredItems;

WriteLn('Waiting for 5 seconds...');
PumpSleep(5*1000);

WriteLn('Done...');
end;

```

### 6.3.4.2.2 The Where clause

The Where clause of the event filter is represented using the [UAContentFilterElement](#) object. This objects contains the filter operand to be evaluated (the [FilterOperator](#) property, which is a [UAFilterOperator](#) enumeration), and the operands used by the selected operator (the [FilterOperands](#) property, which is a [UAFilterOperandCollection](#)). The [UAFilterOperator](#) enumeration contains all operators supported by OPC UA (Equals, IsNull, GreaterThan, LessThan, GreaterThanOrEqual, LessThanOrEqual, Like, Not, Between, InList, And, Or, Cast, InView, OfType, RelatedTo, BitwiseAnd, BitwiseOr).

The `UAFilterOperandCollection` is a collection of content filter operands (`UAFilterOperand`).

The `UAFilterOperand` is a base class for four different types of filter operands: `UAAttributeOperand` (an operand consisting of an attribute of a node in a type, with optional relative path and alias), `UAContentFilterElement` (essentially, an operator with operands), `UALiteralOperand` (an operand consisting of a literal value), or `UASimpleAttributeOperand` (an operand consisting of an attribute of a node in a type, with optional simple relative path, and no alias).

In order to make the creation of Where clauses appear shorter in the code, a `UAContentFilterElementBuilder` class is provided. Instances of this builder can be implicitly converted to the content filter element itself (hence you can create just the `UAContentFilterElementBuilder` objects, and use them at all places where the `UAContentFilterElement` object is expected). The builder contains an operator and is a collection of operands; you can use the C# collection initializer to specify the operands.

For further simplification of the coding, the `UAFilterElements` static class provides an easy way to construct basic filter elements (literal, attribute, or a simple attribute), and elements with various operators. Using the `UAFilterElements` class is for most cases the recommended way to create the Where clause of the event filter. For example, `UAFilterElements.GreaterThanOrEqual` creates a Where clause with the ' $\geq$ ' operator. There are methods for each of the OPC UA operators, with various overloads. The `UAFilterElements.Attribute`, `UAFilterElements.Literal` and `UAFilterElements.SimpleAttribute` methods can be used to create other types of filter elements. Moreover, useful overloads exist that assure that you can actually use .NET objects directly as values of operands, and they will be interpreted as literal elements. With use of all these code simplifying features, you can write, for example, the following Where clause:

```
UAFilterElements.GreaterThanOrEqual(UABaseEventObject.Operands.Severity, 500).
```

## C#

---

```
// This example shows how to specify criteria for event notifications.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _UAEEventFilter
    {
        class WhereClause
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.EventNotification += easyUAClient_EventNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeEvent(
                    "opc.tcp://opcuademo-this.com:62544/Quickstarts/AlarmConditionServer",
                    UAObjectIds.Server,
                    1000,
                    new UAEEventFilterBuilder(
                        // Either the severity is >= 500, or the event comes from a specified source node
                        UAFilterElements.Or(
                            UAFilterElements.GreaterThanOrEqual(UABaseEventObject.Operands.Severity, 500),
                            UAFilterElements.Equals(
                                UABaseEventObject.Operands.SourceNode,
                                new
UANodeId("nsu=http://opcfoundation.org/Quickstarts/AlarmCondition;ns=2;s=1:Metals/SouthMotor"))),
                    UABaseEventObject.AllFields));

                Console.WriteLine("Processing event notifications for 30 seconds...");
                System.Threading.Thread.Sleep(30 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeAllMonitoredItems();

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }

            static void easyUAClient_EventNotification(object sender, EasyUAEEventNotificationEventArgs e)
            {
                // Display the event
                Console.WriteLine(e);
            }
        }
    }
}
```

## Free Pascal

---

```
// This example shows how to specify criteria for event notifications.

type
  TClientEventHandlers3 = class
  procedure OnEventNotification(
    Sender: TObject;
    sender0: OleVariant;
    eventArgs: _EasyUAEEventNotificationEventArgs);
  end;

procedure TClientEventHandlers3.OnEventNotification(
  Sender: TObject;
  sender0: OleVariant;
```

```

    eventArgs: _EasyUAEEventNotificationEventArgs);
begin
    // Display the event
    WriteLn(eventArgs.ToString());
end;

class procedure WhereClause.Main;

function ToUAAttributeField(Operand: UASimpleAttributeOperand): UAAttributeField;
var
    AttributeField: UAAttributeField;
begin
    AttributeField := CoUAAttributeField.Create;
    AttributeField.Operand := Operand;
    ToUAAttributeField := AttributeField;
end;

function ObjectTypeIds_BaseEventType: UANodeId;
var
    NodeId: UANodeId;
begin
    NodeId := CoUANodeId.Create;
    NodeId.StandardName := 'BaseEventType';
    ObjectTypeIds_BaseEventType := NodeId;
end;

function UAFilterElements_SimpleAttribute(TypeId: UANodeId; SimpleRelativeBrowsePathString: string): UASimpleAttributeOperand;
var
    Operand: UASimpleAttributeOperand;
    BrowsePathParser: UABrowsePathParser;
begin
    BrowsePathParser := CoUABrowsePathParser.Create;
    Operand := CoUASimpleAttributeOperand.Create;
    Operand.TypeId.NodeId := TypeId;
    Operand.QualifiedNames := BrowsePathParser.ParseRelative(SimpleRelativeBrowsePathString).ToUAQualifiedNameCollection;
    UAFilterElements_SimpleAttribute := Operand;
end;

function UABaseEventObject_Operands_NodeId: UASimpleAttributeOperand;
var
    Operand: UASimpleAttributeOperand;
begin
    Operand := CoUASimpleAttributeOperand.Create;
    Operand.TypeId.NodeId.StandardName := 'BaseEventType';
    Operand.AttributeId := UAAttributeId_NodeId;
    UABaseEventObject_Operands_NodeId := Operand;
end;

function UABaseEventObject_Operands_EventId: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_EventId := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/EventId');
end;

function UABaseEventObject_Operands_EventType: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_EventType := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/EventType');
end;

function UABaseEventObject_Operands_SourceNode: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_SourceNode := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
    '/SourceNode');
end;

function UABaseEventObject_Operands_SourceName: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_SourceName := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
    '/SourceName');
end;

function UABaseEventObject_Operands_Time: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_Time := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/Time');
end;

function UABaseEventObject_Operands_ReceiveTime: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_ReceiveTime := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType,
    '/ReceiveTime');
end;

function UABaseEventObject_Operands_LocalTime: UASimpleAttributeOperand;
begin
    UABaseEventObject_Operands_LocalTime := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/LocalTime');
end;

function UABaseEventObject_Operands_Message: UASimpleAttributeOperand;

```

```

begin
  UABaseEventObject_Operands_Message := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/Message');
end;

function UABaseEventObject_Operands_Severity: UASimpleAttributeOperand;
begin
  UABaseEventObject_Operands_Severity := UAFilterElements_SimpleAttribute(ObjectTypeIds_BaseEventType, '/Severity');
end;

function UABaseEventObject_AllFields: UAAttributeFieldCollection;
var
  Fields: UAAttributeFieldCollection;
begin
  Fields := CoUAAttributeFieldCollection.Create;

  Fields.Add(ToUAAttributeField(UABaseEventObject_NodeId));

  Fields.Add(ToUAAttributeField(UABaseEventObject_EventId));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_EventType));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_SourceNode));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_SourceName));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_Time));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_ReceiveTime));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_LocalTime));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_Message));
  Fields.Add(ToUAAttributeField(UABaseEventObject_Operands_Severity));

  UABaseEventObject_AllFields := Fields;
end;

var
  Arguments: OleVariant;
  EvsClient: TEvsEasyUAClient;
  Client: EasyUAClient;
  ClientEventHandlers: TClientEventHandlers3;
  Handle: Cardinal;
  HandleArray: OleVariant;
  MonitoredItemArguments: EasyUAMonitoredItemArguments;
  MonitoringParameters: UAMonitoringParameters;
  EventFilter: UAEEventFilter;
  WhereClause: UAContentFilterElement;
  Operand1: UASimpleAttributeOperand;
  Operand2: UALiteralOperand;
  Operand3: UASimpleAttributeOperand;
  Operand4: UALiteralOperand;
  SourceNodeId: UANodeId;
  Element1, Element2: UAContentFilterElement;
  BaseEventType: UANodeId;
begin
  // Instantiate the client object and hook events
  EvsClient := TEvsEasyUAClient.Create(nil);
  Client := EvsClient.ComServer;
  ClientEventHandlers := TClientEventHandlers3.Create;
  EvsClient.OnEventNotification := @ClientEventHandlers.OnEventNotification;

  WriteLn('Subscribing...');

  WhereClause := CoUAContentFilterElement.Create;
  BaseEventType := CoUaNodeId.Create;
  BaseEventType.StandardName := 'BaseEventType';

  // Either the severity is >= 500, or the event comes from a specified source node
  Operand1 := UABaseEventObject_Operands_Severity;
  Operand2 := CoUALiteralOperand.Create;
  Operand2.Value := 500;
  Element1 := CoUAContentFilterElement.Create;
  Element1.FilterOperator := UAFilterOperator_GreaterThanOrEqual;
  Element1.FilterOperands.Add(Operand1);
  Element1.FilterOperands.Add(Operand2);
  Operand3 := UABaseEventObject_Operands_SourceNode;
  SourceNodeId := CoUANodeId.Create;
  SourceNodeId.ExpandedText := 'nsu=http://opcfoundation.org/Quickstarts/AlarmCondition;ns=2;s=1:Metals/SouthMotor';
  Operand4 := CoUALiteralOperand.Create;
  Operand4.Value := SourceNodeId;
  Element2 := CoUAContentFilterElement.Create;
  Element2.FilterOperator := UAFilterOperator_Equals;
  Element2.FilterOperands.Add(Operand3);
  Element2.FilterOperands.Add(Operand4);
  WhereClause.FilterOperator := UAFilterOperator_Or;
  WhereClause.FilterOperands.Add(Element1);
  WhereClause.FilterOperands.Add(Element2);

  EventFilter := CoUAEEventFilter.Create;
  EventFilter.SelectClauses := UABaseEventObject_AllFields;
  EventFilter.WhereClause := WhereClause;

  MonitoringParameters := CoUAMonitoringParameters.Create;
  MonitoringParameters.SamplingInterval := 1000;

```

```
MonitoringParameters.EventFilter := EventFilter;
MonitoringParameters.QueueSize := 1000;

MonitoredItemArguments := CoEasyUAMonitoredItemArguments.Create;
MonitoredItemArguments.EndpointDescriptor.UrlString := 'opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer';
MonitoredItemArguments.NodeDescriptor.NodeId.StandardName := 'Server';
MonitoredItemArguments.MonitoringParameters := MonitoringParameters;
//MonitoredItemArguments.SubscriptionParameters.PublishingInterval := 0;
MonitoredItemArguments.AttributeId := UAAttributeId_EventNotifier;

Arguments := VarArrayCreate([0, 0], varVariant);
Arguments[0] := MonitoredItemArguments;

TVarData(HandleArray).VType := varArray or varVariant;
TVarData(HandleArray).VArray := PVarArray(
    Client.SubscribeMultipleMonitoredItems(PSafeArray(TVarData(Arguments).VArray)));

WriteLn('Processing event notifications for 30 seconds...');
PumpSleep(30*1000);

WriteLn('Unsubscribing...');
Client.UnsubscribeAllMonitoredItems;

WriteLn('Waiting for 5 seconds...');
PumpSleep(5*1000);

WriteLn('Done...');
end;
```

## 6.3.4.3 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 sampling interval, or the event filter. Instead, change the parameters by calling the [ChangeMonitoredItemSubscription](#) method, passing it the subscription handle, and the new parameters.

For changing parameters of multiple subscriptions in an efficient manner, call the [ChangeMultipleMonitoredItemSubscriptions](#) method.

## 6.3.4.4 Unsubscribing from OPC Events

If you no longer want to receive event notifications, you need to unsubscribe from them. To unsubscribe from a single monitored item, call the [UnsubscribeMonitoredItem](#) method, passing it the (event) subscription handle. To unsubscribe from events in an efficient manner, call the [UnsubscribeMultipleMonitoredItems](#) method (instead of calling [UnsubscribeMonitoredItem](#) in a loop), passing it an array of (event) subscription handles.

You can unsubscribe from all monitored items you have previously subscribed to (on the same instance of [EasyUAClient](#) object) by calling the [UnsubscribeAllMonitoredItems](#) method.

## 6.3.4.5 Refreshing Condition States

The [UAMonitoredItemArguments](#) class contains a [Boolean AutoConditionRefresh](#) property. When set (this is the default), a UA ConditionRefresh will be automatically performed when needed to keep the condition information up-to-date.

The component handles special events reserved for condition refresh automatically, and translates them to event notifications with a special form of event arguments. Event notifications ([EasyUAEventNotificationArgs](#)) that relate to condition refresh have their [Refresh](#) property set to '[true](#)'. In addition, a start of condition refresh is indicated by a notification with event arguments that have their [RefreshInitiated](#) property set to '[true](#)'. Correspondingly, an end of condition refresh is indicated by a notification with event arguments that have their [RefreshComplete](#) property set to '[true](#)'.

## 6.3.4.6 Notification Event

When an OPC UA Alarms & Conditions server generates an event, the [EasyUAClient](#) object generates an [EventNotification](#) event. For subscription mechanism to be useful, you should hook one or more event handlers to this event. This event is raised for every event generated by a subscribed OPC monitored item.

To be more precise, the [EventNotification](#) 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 [EventNotification](#) when a successful connection (or re-connection) is made. In this case, the [Exception](#) and [EventData](#) properties of the event arguments are null references.
2. You receive the [EventNotification](#) 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 [EventData](#) property is a null reference.
3. You receive one additional [EventNotification](#) when the component is about to send you notifications for condition refresh, and one additional [EventNotification](#) after the component has sent you all notifications for the condition refresh. In these cases, the [RefreshInitiated](#) or [RefreshComplete](#) property of the event arguments is set to '[true](#)', and the [Exception](#) and [EventData](#) properties contain null references.

The notification for the [EventNotification](#) event contains an [EasyUAEVENTNotificationEventArgs](#) argument. You will find all kind of relevant data in this object. Some properties in this object contain valid information under all circumstances - e.g. [Arguments](#) (including [Arguments.State](#)). Other properties, such as [EventData](#), contain null references when there is no associated information for them. When the [EventData](#) property is not a null reference, it contains a [UAEventData](#) object describing the detail of the actual OPC event received from the OPC UA Alarms & Conditions 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 [EventData](#) 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 condition refresh is being initiated or is complete (in this case, the [RefreshInitiated](#) or [RefreshComplete](#) property is '[true](#)'), or that an event subscription has been successfully connected or re-connected (in this case, the [EventData](#) property is a null reference). If none of the previous applies, the [EventData](#) property contains a valid [UAEventData](#) object with details about the actual OPC event generated by the OPC server.

Pseudo-code for the full [EventNotification](#) 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.RefreshInitiated then
    A "refresh" has been initiated; handle it
else if notificationEventArgs.RefreshComplete then
    A "refresh" is complete; handle it
else if notificationEventArgs.EventData is null then
    Subscription has been successfully connected or re-connected, handle it (or ignore)
else
    Handle the OPC event, details are in notificationEventArgs.EventData. You may use
    notificationEventArgs.Refresh flag for distinguishing refreshes from original notifications.
```

The [EventNotification](#) event handler is called on a thread determined by the [EasyUAClient](#) component. For details, please refer to "Multithreading and Synchronization" chapter under "Advanced Topics".

The arguments of each event notification ([EasyUAEVENTNotificationEventArgs](#)) have an [EventData](#) property (of type [UAEventData](#), described further below), which contains details about the OPC UA event. In addition, common information such as the copy of the input arguments to the subscription, i.e. identification of the endpoint and node (monitored item) which has generated the event, is included. The [Exception](#) property (and several related properties)

indicates success (when it is a null reference), or a failure (in which case it contains an exception object describing the error).

The `UAEventData` object contains details about an OPC UA event, i.e. the information that the OPC UA server sends with the event notification. Primarily, it has a `FieldResults` property, which is a dictionary containing results for each of the fields in the select clauses of the event filter. Each field results is a `ValueResult` object, i.e. it can either contain the actual value, or an error indication.

## C#

```
// This example shows how to display all fields of incoming events, or extract specific
// fields.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _UAEventData
    {
        class FieldResults
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.EventNotification += easyUAClient_EventNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeEvent(
                    "opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer",
                    UAObjectIds.Server,
                    1000);

                Console.WriteLine("Processing event notifications for 30 seconds...");
                System.Threading.Thread.Sleep(30 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeAllMonitoredItems();

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }

            static void easyUAClient_EventNotification(object sender,
EasyUAEEventArgs e)
            {
                Console.WriteLine();

                // Display the event
                if (e.EventData == null)
                {
                    Console.WriteLine(e);
                    return;
                }
                Console.WriteLine("All fields:");
                foreach (KeyValuePair<UAAttributeField, ValueResult> pair in
e.EventData.FieldResults)
                {
                    UAAttributeField attributeField = pair.Key;
                    ValueResult valueResult = pair.Value;
                    Console.WriteLine(" {0} -> {1}", attributeField, valueResult);
                }
            }
        }
    }
}
```

```
        }
        // Extracting a specific field using a standard operand symbol
        Console.WriteLine("Source name: {0}",
            e.EventData.FieldResults[UABaseEventObject.Operands.SourceName]);
        // Extracting a specific field using an event type ID and a simple
relative path
        Console.WriteLine("Message: {0}",
e.EventData.FieldResults[UAFilterElements.SimpleAttribute(UAObjectTypeIds.BaseEventType,
"/Message"))];
    }
}
```

In order to make access to at least some of these results easier (without having to go through indexing an element in the dictionary), the [UAEventData](#) object also contains a [BaseEvent](#) property (of [UABaseEventObject](#) type). The [UABaseEventObject](#) class contains properties that directly correspond to information contained in the base event type (from which all UA events are derived), such as [SourceName](#), [Time](#), [Message](#), [Severity](#), etc. In case of an error related to a particular field, the corresponding property will contain its default value (e.g. an empty string for a [Message](#)).

C#

```
// This example shows how to display specific fields of incoming events that are
derived from base event type.
using OpcLabs.EasyOpc.UA;
using System;

namespace UADocExamples
{
    namespace _UAEventData
    {
        class BaseEvent
        {
            public static void Main()
            {
                // Instantiate the client object and hook events
                var easyUAClient = new EasyUAClient();
                easyUAClient.EventNotification += easyUAClient_EventNotification;

                Console.WriteLine("Subscribing...");
                easyUAClient.SubscribeEvent(
                    "opc.tcp://opcuademo-
this.com:62544/Quickstarts/AlarmConditionServer",
                    UAObjectIds.Server,
                    1000);

                Console.WriteLine("Processing event notifications for 30 seconds...");
                System.Threading.Thread.Sleep(30 * 1000);

                Console.WriteLine("Unsubscribing...");
                easyUAClient.UnsubscribeAllMonitoredItems();

                Console.WriteLine("Waiting for 5 seconds...");
                System.Threading.Thread.Sleep(5 * 1000);
            }

            static void easyUAClient_EventNotification(object sender,
EasyUAEVENTIFICATIONEventArgs e)
            {
                Console.WriteLine();
            }
        }
    }
}
```

```
// Display the event
if (e.EventData == null)
{
    Console.WriteLine(e);
    return;
}
UABaseEventObject baseEventObject = e.EventData.BaseEvent;
Console.WriteLine("Source name: {0}", baseEventObject.SourceName);
Console.WriteLine("Message: {0}", baseEventObject.Message);
Console.WriteLine("Severity: {0}", baseEventObject.Severity);
}
}
}
```

It should be noted that the [FieldResults](#) directory always contains results for the fields you have specified in Select clause of the event filter, but it can contain some additional fields as well. This happens when these fields are necessary for internal processing inside the component. Your code should be able to ignore the results that it has not requested.

Note: Some data returned use complex data types defined by OPC UA, which QuickOPC transforms to its own .NET types. For example, the [UATimeZoneData](#) class contains data returned in the [UABaseEventObject.LocalTime](#) property (defines the local time that may or may not take daylight saving time into account).

## 6.3.4.7 Using Callback Methods Instead of Event Handler

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 UA Alarms & Conditions as well.

The [EasyUAMonitoredItemArguments](#) object has an [EventCallback](#) property. This property specifies an optional method to be invoked for each event notification generated by the monitored item.

There is also an [IEasyUAClient.SubscribeMonitoredItem](#) extension method overload that accepts an event callback ([EasyUAEventNotificationEventHandler](#)).

### C#

```
// This example shows how to subscribe to event notifications and display each
incoming event
// using a callback method that is provided as lambda expression.
using OpcLabs.EasyOpc.UA;
using System;
using OpcLabs.EasyOpc.UA.AddressSpace;

namespace UADocExamples
{
    namespace _EasyUAClient
    {
        partial class SubscribeEvent
        {
            public static void CallbackLambda()
            {
                // Instantiate the client object
                var easyUAClient = new EasyUAClient();

                Console.WriteLine("Subscribing...");
                // The callback is a lambda expression the displays the event
                easyUAClient.SubscribeEvent(
```

```
        "opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer",
UAObjectIds.Server,
1000,
(sender, eventArgs) => Console.WriteLine(eventArgs));
// Remark: Production code would check e.Exception before accessing
e.EventData.

Console.WriteLine("Processing event notifications for 30
seconds...");
System.Threading.Thread.Sleep(30 * 1000);

Console.WriteLine("Unsubscribing...");
easyUAClient.UnsubscribeAllMonitoredItems();

Console.WriteLine("Waiting for 2 seconds...");
System.Threading.Thread.Sleep(2 * 1000);
}
}
}
```

## VB.NET

---

```
' This example shows how to subscribe to event notifications and display each
incoming event
' using a callback method that is provided as lambda expression.
Imports OpcLabs.EasyOpc.UA
Imports OpcLabs.EasyOpc.UA.AddressSpace

Namespace _EasyUAClient
    Partial Friend Class SubscribeEvent
        Public Shared Sub CallbackLambda()
            ' Instantiate the client object
            Dim easyUAClient = New EasyUAClient()

            Console.WriteLine("Subscribing...")
            ' The callback is a lambda expression the displays the event
            easyUAClient.SubscribeEvent(_
                "opc.tcp://opcua.demo-
this.com:62544/Quickstarts/AlarmConditionServer", _
                UAObjectIds.Server, _
                1000, _
                Sub(sender, eventArgs) Console.WriteLine(eventArgs))
            ' Remark: Production code would check e.Exception before accessing
            e.EventData.

            Console.WriteLine("Processing event notifications for 10 seconds...")
            Threading.Thread.Sleep(10 * 1000)

            Console.WriteLine("Unsubscribing...")
            easyUAClient.UnsubscribeAllMonitoredItems()

            Console.WriteLine("Waiting for 2 seconds...")
            Threading.Thread.Sleep(2 * 1000)
        End Sub
    End Class
End Namespace
```

## 6.3.5 Setting Parameters

As the Alarms & Conditions support for OPC UA uses the same [EasyUAClient](#) object, the same properties and objects are used for setting the parameters as for OPC Data. Please refer to the "Setting Parameters" chapter under "Procedural Coding Model for OPC Data (Class and UA)" for all necessary information.

## 6.4 Operation Monitoring and Control

The [EasyUAClient](#) object provides a [ServerConditionChanged](#) event. This event is raised for a change in the condition of an OPC server. Specifically, you receive a notification whenever the connection to the server is established, or whenever the connection is terminated. Whenever there is an error (exception) associated with the server condition, it is also available in the event arguments.

The event notification passes to you an [EasyUAServerConditionChangedEventArgs](#) object, which contains data about the new server condition. The [EndpointDescriptor](#) property identifies the server the notification applies to.

The [Boolean Connected](#) property indicates whether the server is now connected. The [Exception](#) property contains a null reference in case of no error, or it contains an exception object in case of problems.

Additional properties of the event arguments distinguish the change further. For example, the [ConnectionState](#) property contains an enumerated value ( [ConnectionState Enumeration](#)) with members [Disconnected](#), [Connecting](#), [Connected](#), and [Disconnecting](#). The [RetrialDelay](#) property (valid in the [Disconnected](#) state) indicates the time (in milliseconds) to the next reconnection attempt.

A corresponding event is not currently available for OPC Classic.

Note that the information provided by this event should be used for oversight, informational and diagnostics purposes only. Specifically, you should not use it to attempt to implement any error recovery mechanism, because error recovery is already built into the QuickOPC software, and your own mechanism would inevitably cause conflicts.

## 7 Live Mapping Model

### 7.1 Live Mapping Model for OPC Data (Classic and UA)



The live mapping development model allows you to write objects that correspond logically to a functionality provided by whatever is "behind" the OPC data. For example, if part of your application works with a boiler, it is natural for you to create a boiler object in the code. You then describe how your objects and their members correspond to OPC data – this is done using attributes to annotate your objects. Using Live Mapping methods, you then map your objects to OPC data, and perform OPC operations on them.

For example, when you subscribe to OPC items (in OPC Classic) or OPC attributes (in OPC-UA), incoming OPC notifications about changes can directly set corresponding properties in your objects, without any further coding. You can then focus on writing the code that handles these property changes, abstracting away the fact how they come in.

Note: Some mapping technologies for .NET already exist, from Microsoft or otherwise. These technologies are mainly focused on mapping database information (ORM – Object-relational mapping), and they are not well suitable for automation purposes like OPC. We therefore had to implement a new mapping model. We use the term "live mapping" so that it is clearly distinguished from other mapping technologies used; it also reflects the fact that this kind of mapping is well-suited for real-time, "live" data.

Do not confuse the live mapping model with live binding, also provided by QuickOPC. The main differences between the two are that

- live mapping is primarily non-visual (both in design and in runtime), and
- live mapping has the ability to work with object types, not just individual members.

With live mapping model, you generally need to do following steps in order:

1. Write classes that are a logical model of your application. If you are following object-oriented principles in your design, chances are that your code already has such classes anyway.
2. Annotate the classes with attributes for live mapping. The main purpose of these attributes is to describe how the members of your classes correspond to nodes in the OPC address space.
3. Create instance(s) of your classes, as usually.
4. Perform the actual mapping, which associates your objects with OPC data.
5. Call mapping operations as needed, e.g. execute OPC reads, writes, or subscribe to changes in OPC values.

#### 7.1.1 Live Mapping Example

It is best to start with Live Mapping by going through a simple example.

##### Define the mapping

Our example works with a hypothetical boiler, which is an object composed from other structured objects, such as controllers, a drum, level indicator, and pipes. Some of these objects are composed as well. The precise structure is described in the code comments below.

Note: The code snippets demonstrated here are for OPC Classic. In OPC-UA, the principles of Live Mapping are the same, but the actual attributes and pieces of code are slightly different. If you want to use Live Mapping with OPC-UA, please use the explanation here as a guidance, and refer to the UAConsoleLiveMapping example in the product instead.

The simulated boiler data are provided by the sample OPC server that ships with the QuickOPC product.

The code below is taken from the ConsoleLiveMapping example that ships with the product (file **Boiler.cs**):

## Define Live Mapping

```
using System;
using OpcLabs.BaseLib.LiveMapping;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.LiveMapping;
using OpcLabs.EasyOpc.DataAccess.LiveMapping.Extensions;

namespace ConsoleLiveMapping
{
    // The Boiler and its constituents are described in our application
    // domain terms, the way we want to work with them.
    // Attributes are used to describe the correspondence between our
    // types and members, and OPC nodes.

    // This is how the boiler looks in OPC address space:
    // - Boiler #1
    //     - CC1001           (CustomController)
    //         - ControlOut
    //         - Description
    //         - Input1
    //         - Input2
    //         - Input3
    //     - Drum1001          (BoilerDrum)
    //         - LIX001          (LevelIndicator)
    //             - Output
    //     - FC1001           (FlowController)
    //         - ControlOut
    //         - Measurement
    //         - SetPoint
    //     - LC1001           (LevelController)
    //         - ControlOut
    //         - Measurement
    //         - SetPoint
    //     - Pipe1001          (BoilerInputPipe)
    //         - FTX001          (FlowTransmitter)
    //             - Output
    //     - Pipe1002          (BoilerOutputPipe)
    //         - FTX002          (FlowTransmitter)
    //             - Output

    [DAType]
    class Boiler
    {
        // Specifying BrowsePath-s here only because we have named the
        // class members differently from OPC node names.

        [DANode(BrowsePath = "Pipe1001")]
        public BoilerInputPipe InputPipe = new BoilerInputPipe();

        [DANode(BrowsePath = "Drum1001")]
        public BoilerDrum Drum = new BoilerDrum();
    }
}
```

```
[DANode(BrowsePath = "Pipe1002")]
public BoilerOutputPipe OutputPipe = new BoilerOutputPipe();

[DANode(BrowsePath = "FC1001")]
public FlowController FlowController = new FlowController();

[DANode(BrowsePath = "LC1001")]
public LevelController LevelController = new LevelController();

[DANode(BrowsePath = "CC1001")]
public CustomController CustomController = new CustomController();
}

[DAType]
class BoilerInputPipe
{
    // Specifying BrowsePath-s here only because we have named the
    // class members differently from OPC node names.

    [DANode(BrowsePath = "FTX001")]
    public FlowTransmitter FlowTransmitter1 = new FlowTransmitter();

    [DANode(BrowsePath = "ValveX001")]
    public Valve Valve = new Valve();
}

[DAType]
class BoilerDrum
{
    // Specifying BrowsePath-s here only because we have named the
    // class members differently from OPC node names.

    [DANode(BrowsePath = "LIX001")]
    public LevelIndicator LevelIndicator = new LevelIndicator();
}

[DAType]
class BoilerOutputPipe
{
    // Specifying BrowsePath-s here only because we have named the
    // class members differently from OPC node names.

    [DANode(BrowsePath = "FTX002")]
    public FlowTransmitter FlowTransmitter2 = new FlowTransmitter();
}

[DAType]
class FlowController : GenericController
{
```

```
[DAType]
class LevelController : GenericController
{
}

[DAType]
class CustomController
{
    [DANode, DAItem]
    public double Input1 { get; set; }

    [DANode, DAItem]
    public double Input2 { get; set; }

    [DANode, DAItem]
    public double Input3 { get; set; }

    [DANode, DAItem(Operations = DAItemMappingOperations.ReadAndSubscribe)] // no
OPC writing
    public double ControlOut { get; set; }

    [DANode, DAItem]
    public string Description { get; set; }
}

[DAType]
class FlowTransmitter : GenericSensor
{
}

[DAType]
class Valve : GenericActuator
{
}

[DAType]
class LevelIndicator : GenericSensor
{
}

[DAType]
class GenericController
{
    [DANode, DAItem(Operations = DAItemMappingOperations.ReadAndSubscribe)] // no
OPC writing
    public double Measurement { get; set; }

    [DANode, DAItem]
    public double SetPoint { get; set; }
```

```
[DANode, DAIItem(Operations =
    DAIItemMappingOperations.ReadAndSubscribe)] // no OPC writing
public double ControlOut { get; set; }

}

[DAType]
class GenericSensor
{
    // Meta-members are filled in by information collected during
    // mapping, and allow access to it later from your code.
    // Alternatively, you can derive your class from DAMappedNode,
    // which will bring in many meta-members automatically.
    [MetaMember("NodeDescriptor")]
    public DANodeDescriptor NodeDescriptor { get; set; }

    [DANode, DAIItem(Operations = DAIItemMappingOperations.ReadAndSubscribe)] // no
    OPC writing
    public double Output
    {
        get { return _output; }
        set
        {
            _output = value;
            Console.WriteLine("Sensor \"\{0\}\" output is now \{1\}.",
NodeDescriptor, value);
        }
    }

    private double _output;
}

[DAType]
class GenericActuator
{
    [DANode, DAIItem]
    public double Input { get; set; }
}
```

Notice the setter for the `GenericSensor.Output` property in the code above. It contains your application logic for “what to do when sensor’s output changes” (in our example, we simply display the sensor’s identification and the new value). Similarly, you can write logic that handles other OPC item changes, or – in the opposite direction – provides values to be written to the OPC items.

## Perform operations

Let us now use the above mapping definitions to actually perform some meaningful tasks.

First, we create an instance of our logical “boiler” object, and using the `DAClientMapper` object (for mapping to OPC Data Access), we map it to OPC. The type mapping definition for the `Boiler` does not contain information about which OPC server to use, and where in its address space (tree of items) can the boiler items be found. This is intentional - although such information could have been specified directly on the `Boiler` class, it would hamper the reusability of

such class. We therefore provide that information at the moment of mapping, by specifying it inside the new [DAMappingContext](#) passed to the mapper. Similarly, we specify the requested update rate.

The code pieces below are taken from the `ConsoleLiveMapping` example that ships with the product (file `Program.cs`).

## Perform Live Mapping

```
varboiler1 = new Boiler();
varmapper = new DAClientMapper();
mapper.Map(boiler1, new DAMappingContext
{
    ServerDescriptor = "OPCLabs.KitServer.2", // local OPC server
    // The NodeDescriptor below determines where in the OPC address
    // space we want to map our data to.
    NodeDescriptor = new DANodeDescriptor
    {
        BrowsePath = "/Boilers/Boiler #1" },
    // Requested update rate (for subscriptions):
    GroupParameters = 1000,
});
```

With the mapping already in place, we can easily do various operations on the mapped objects. For example, we can read in all data of the boiler, and the values will appear directly in the properties of our Boiler object:

## Read data with Live Mapping

```
Console.WriteLine();
Console.WriteLine("Reading all data of the boiler...");
mapper.Read();
Console.WriteLine("Drum level is: {0}",
    boiler1.Drum.LevelIndicator.Output);
```

You can also modify properties of the level controller, e.g. change its setpoint by setting the `SetPoint` property, and then send all writeable values to level controller through OPC:

## Write data with Live Mapping

```
Console.WriteLine();
Console.WriteLine("Writing new setpoint value...");
boiler1.LevelController.SetPoint = 50.0;
mapper.WriteTarget(boiler1.LevelController, /*recurse:*/false);
```

If you want to subscribe to changes occurring in the boiler, and have the properties in your object automatically updates with new values, you can do it as below. When, e.g., an output of any sensor changes, the `Output` property will be set on the corresponding object, and the code that (in our example) displays the new value will run.

## Subscribe to data with Live Mapping

```
Console.WriteLine();
Console.WriteLine("Subscribing to boiler data changes...");
mapper.Subscribe(/*active:*/true);

Thread.Sleep(30 * 1000);

Console.WriteLine();
Console.WriteLine("Unsubscribing from boiler data changes...");
```

```
mapper.Subscribe(/*active:*/false);
```

## Benefits

You can already see some of the benefits of the live mapping in this simple example. The code that you write can focus on the logic of the application, and works directly with members of your .NET objects. You do not have to clutter your code with OPC-specific information such as server and item IDs. It also gets very easy to reuse the mapped types: If you had more boilers that are represented identically in the OPC address space, you can simply create more Boiler objects and map them all, without a need to change anything in the definition of the [Boiler](#) class.

## 7.1.2 Live Mapping Overview

In live mapping model, you end up creating Mappings that relate Mapping Sources with Mapping Targets. These concepts are explained further below.

The live mapping is orchestrated by the mapper object, described in a separate chapter (see [The Mapper Object](#)).

### 7.1.2.1 Mapping Sources

The mapping source is a data provider and/or consumer external to your application code.

For OPC Data Access, there are two types of mapping sources that you can use:

- The [DAClientItemSource](#) represents an OPC item.
- The [DAClientPropertySource](#) represents an OPC property.

For OPC Unified Architecture, following type of mapping source exists:

- The [UAClientDataMappingSource](#) represents an attribute of an OPC-UA node.

One mapping source can be (and often is) shared by multiple mappings. This happens e.g. when you map multiple members to the same source, using different Mapping Kinds (for example, you may map separate members to the value, timestamp, and quality information of the OPC item).

When you use the attributes to define the live mappings, mapping source objects are created and maintained together with their Mappings, so normally you do not deal with them directly.

### 7.1.2.2 Mapping Targets

The mapping target is an element in your code from which data can be obtained or that can accept data. Most typically, it is a property of your object, but it can also be a field, method, or an event.

The actual mapping target is represented by the [ObjectMemberLinkingTarget](#) class. When you use the attributes to define the live mappings, mapping target objects are created and maintained together with their Mappings, so normally you do not deal with them directly.

### 7.1.2.3 Mappings

The mapping is a relation between a mapping source and a mapping target. Besides specifying which mapping source and mapping target belong together, it also specifies how and when the data is being transferred between the two.

For example, a mapping for OPC Data Access items ([DAClientItemMapping](#)) may specify that the data is only transferred (from the source to the target) when a "Read" operation is requested (see [Mapping Operations](#)) and that

the data will be in form of the value-timestamp-quality triple, i.e. a [DAVtq](#) object (see Mapping Kinds). With OPC-UA and [UAClientDataMapping](#), the "Read" operation exists as well, and an example form of transfer may be the [UAAttributeData](#), which contains the attribute value, status code, and timestamps. Or, the transferred data may simple be the OPC value itself.

When you annotate your objects using attributes, and then map your objects using the mapper object, the necessary mappings are created for all members of your objects that are annotated in such way. You therefore do not normally need to create mappings individually.

For OPC Data Access, there are two types of mappings that you can use:

- The [DAClientItemMapping](#) relates a [DAClientItemSource](#) to some mapping target. It refers to an OPC item, and you can read it or write it, or subscribe to it. These mappings are created with the use of the [DALitem](#) attribute (see further below).
- The [DAClientPropertyMapping](#) relates a [DAClientPropertySource](#) to some mapping target. It refers to an OPC property, and you can get its value. These mappings are created with the use of the [DAProperty](#) attribute (see further below).

For OPC Unified Architecture, following type of mapping exists:

- The [UAClientDataMapping](#) relates a [UAClientDataMappingSource](#) to some mapping target. It refers to an attribute of an OPC-UA node , and you can read it or write it, or subscribe to it. These mappings are created with the use of the [UAData](#) attribute (see further below).

## 7.1.3 Attributes for Live Mapping

By annotating your types and members by attributes, you provide information to the live mapping system about the correspondences between the source OPC system and the target .NET objects.

For example, using the combination of [DANode](#) and [DALitem](#) attributes on a property of your object, you denote that the property corresponds to a node in OPC-DA address space, specify the name of that node (if different from property name), and mark the node as OPC item, i.e. capable of reading, writing or subscriptions. Using other attributes, you can also specify parameters for these mapping operations (such as the requested update rate). In OPC-UA, you would use a similar combination of [UANode](#) and [UAData](#) attributes.

Attributes in .NET are classes that, by convention, are named so that they end in the word "**Attribute**". Some languages, such as C#, allow you to omit the final **Attribute** word, when the attribute is specified. For example, in C#, you can write [\[DALitem\]](#) instead of [\[DALitemAttribute\]](#). In this document, we will mostly use the "shortened" named of attribute classes. Remember that in some languages, you may have to use the full name of the attribute class, and therefore append the word "**Attribute**" to the attribute name listed here.

The following table shows the Live Mapping attributes common for OPC Classic and OPC-UA, and the type of elements they can be applied to.

Attribute	Purpose	on Class or Struct	on Property, Method, or Field	on Event
<a href="#">MappingTag</a>	Member mapping		✓	✓
<a href="#">MetaMember</a>	Member mapping		✓	

The following table shows the attributes available for OPC Classic, and the type of elements they can be applied to.

Attribute	Purpose	on Class or Struct	on Property, Method, or Field	on Event
<a href="#">DALitem</a>	Member mapping		✓	✓

Attribute	Purpose	on Class or Struct	on Property, Method, or Field	on Event
DALItemIdTemplate	Propagated parameter	✓		
DAMember	Member mapping		✓	
DANode	Member mapping		✓	✓
DAProperty	Member mapping		✓	
DARead	Propagated parameter	✓	✓	✓
DASubscription	Propagated parameter	✓	✓	✓
DAType	Type mapping	✓		
Server	Propagated parameter		✓	✓

The following table shows the attributes available for OPC Unified Architecture, and the type of elements they can be applied to.

Attribute	Purpose	on Class or Struct	on Property, Method, or Field	on Event
UAData	Member mapping		✓	✓
UADataChangeFilter	Propagated parameter	✓	✓	✓
UAEndpoint	Propagated parameter		✓	✓
UAMember	Member mapping		✓	
UAMonitoring	Propagated parameter	✓	✓	✓
UANamespace	Propagated parameter	✓		
UANode	Member mapping		✓	✓
UANodeIdTemplate	Propagated parameter	✓		
UARead	Propagated parameter	✓	✓	✓
UASubscription	Propagated parameter	✓	✓	✓
UAType	Type mapping	✓		

The meaning of these attributes will be explained in subsequent chapters, and can be found in the reference documentation as well.

## 7.1.4 Propagated Parameters

Attributes that are marked with “propagated parameter” in the above table do not create a new type definition or member mapping. Instead, when used, they establish or change parameter values that will be used when the actual mapping is created, using attributes for Type Mapping or Member Mapping.

The parameters set in this way also propagate further down in the mapping hierarchy. E.g. if you put a propagated parameter attribute on a type, it will apply to that type and all its mapped members, and if any such member refer to a mapped type, the parameters will propagate recursively to it. If, however, any of these down-level types or members is annotated with the same attribute, the values from this new attribute will replace the parameters that were set higher in the hierarchy.

## 7.1.5 Type Mapping

Types involved in the live mapping for OPC Data need to be annotated with the [DAType](#) attribute (OPC Classic) or the [UAType](#) attribute (OPC-UA). If you derive your types from one of the Mapped Node Classes (such as [DAMappedNode](#) or [UAMappedNode](#)), this attribute is inherited and you do not have to explicitly specify it again.

Both reference and value types are allowed in the live mapping.

In addition to the [DAType](#) (or [UAType](#)) attribute, you can use other (“propagated parameter”) attributes on your type, as seen from the table in the Attributes for Live Mapping chapter. These additional attributes do not have direct effect on the type itself, but influence all members of that type. For example, if you specify the [DASubscription](#) attribute on your type, and specify the requested update rate, this update rate will be used with all members of that type, unless overridden again by the [DASubscription](#) attribute on any member. Similarly, in OPC Unified Architecture, use can specify the sampling interval using the [UAMonitoring](#) attribute, and it will be propagated further down.

## 7.1.6 Member Mapping

The member mapping has its specifics depending whether you apply the mapping attribute to a property or field (which is most common), to a method, or to an event. In the following three chapters, we will explain the mapping behavior with these different types of members.

The subsequent chapters then explain how the actual mappings are created with different attributes, and what is their meaning and usage.

### 7.1.6.1 Mapping Property and Field Members

This is the most obvious and straightforward member mapping. The property’s or field’s value is the actual mapped data.

The property or field type must be compatible with the type of data, as given by the mapping (see Mapping Kinds for details on types for different kinds of mappings). Note that it is generally preferred to use properties rather than fields for the mapping purposes.

Indexed properties and fields are not currently supported. I.e. you can have an array contained in the mapped property or field, but the array has to be treated as a whole.

### 7.1.6.2 Mapping Method Members

When you map a method, it has to conform to one of the following:

- A method (function) that has no arguments, and returns a value. This method can be used to obtain values

from the target. It is equivalent to a property getter.

- A method that has one argument (return value is ignored and can be void). This method can be used to modify values in the target. It is equivalent to a property setter.

The method return type (in first case), or the method's argument type (in second case) must be compatible with the type of data, as given by the mapping (see Mapping Kinds for details on types for different kinds of mappings).

## 7.1.6.3 Mapping Event Members

An event can only serve for delivering data from the source to the target. When you map an event, the event handler has to accept two arguments. The first argument is the sender – the target object itself. The second argument contains the mapped data, and is of type `DataEventArgs<TData>` (if the target type is not known) or `DataEventArgs<TData,TTarget>` (for target of type `TTarget`).

`TData` is the type of data, as given by the mapping (see Mapping Kinds for details on types for different kinds of mappings).

## 7.1.6.4 Map-through, OPC Nodes and OPC Data

Following attributes are used to create or describe the actual member mappings for OPC Classic (Data Access):

- `DAMember`: Causes the mapping to continue into the value of the mapped member (map-through).
- `DANode`: Marks the member that represents a node in the OPC-DA address space (no new mapping is directly created just by using this attribute), and maps through it.
- `DAItem`: Gives information about OPC Data Access item, and creates a mapping for it.
- `DAProperty`: Gives information about OPC Data Access property, and creates a mapping for it.

Following attributes are used to create or describe the actual member mappings for data with OPC Unified Architecture (OPC-UA):

- `UAMember`: Causes the mapping to continue into the value of the mapped member (map-through).
- `UANode`: Marks the member that represents a node in the OPC-UA address space (no new mapping is directly created just by using this attribute), and maps through it.
- `UAData`: Gives information about OPC-UA data provided by an attribute on an OPC-UA node, and creates a mapping for it.

In order to describe what precisely happens when these attributes and their combinations are used, it is important to understand that when the mapping is done, there is a “current” OPC node maintained by the mapper object, and a “current” object being mapped. When the mapping is started, the current OPC node is given to the mapper (or, if omitted, starts at the root of the address space), and the current object is the object to be mapped, also passed to the mapper.

As the mapping progresses, the current OPC node is only changed by the `DANode` (or `UANode`) attribute. That is, if you want any of your members be mapped to a node different from the parent node, you need to annotate that member with the `DANode` (or `UANode`) attribute. The current OPC node is thus changed for that member and anything below it in the mapping hierarchy.

The `DAMember` and `DANode` (or the `UAMember` and `UANode`) attributes both function as map-through. This means that when they are encountered during mapping, the value of the annotated member is obtained, and the mapping continues into the object given by the value (if the value is a null reference, it is simply ignored). You can therefore have a structure of objects where object's properties or fields contain other objects, and the whole structure can be mapped at once.

The difference between the `DAMember` and `DANode` (or `UAMember` and `UANode`) is that with `DAMember` (`UAMember`), the current OPC node is not changed. This means that members of the child object will be mapped to the same OPC node as the parent object. With `DANode` (`UANode`), however, the members of the child object will be

mapped to an OPC node different from the parent. Which node precisely is used for the child object's members is described in a subsequent chapter, Browse Path and Node Id Resolution. Here, it is enough to say that if you don't specify any argument to the [DANode \(UANode\)](#) attribute, the current OPC node will be changed to a node with the same name as the member you are annotating. So, if your property is named e.g. [Output](#), it will correspond to an [Output](#) node in the OPC address space.

The [DAMember](#) and [DANode](#) (or [UAMember](#) and [UANode](#)) attributes do not create any OPC data mappings that can actually have some effect on the underlying OPC system. They are just helper attributes that allow you to define the structure of mappings. In order to create OPC data mappings, you need to annotate your member with either [DALitem](#) or [DAProperty](#) attribute (in OPC Classic), or with the [UADATA](#) attribute (in OPC-UA).

With the [DALitem](#) attribute, you specify that the annotated member should be mapped to an OPC Data Access item, and you optionally provide additional information about how the mapping should be done. The arguments that you can use with the [DALitem](#) attribute allow you to optionally specify:

- [AccessPath](#): An optional data path suggestion to the server.
- [RequestedDataType](#): The data type requested (VarTypes.Empty for the server's canonical data type).
- [Kind](#): Specifies how the item will be mapped (see Mapping Kinds).
- [Operations](#): The item mapping operations in which the mapping will participate (see Mapping Operations).
- [ItemType](#): The type of OPC item.

Note that with the [DALitem](#) attribute, you do not specify information about which OPC node (item) should be mapped: This information comes from the usage of [DANode](#) attribute(s).

With the [DAProperty](#) attribute, you specify that the annotated member should be mapped to an OPC Data Access property (on an item), specify the numerical ID of the property, and optionally additional information about how the mapping should be done.

With the [UADATA](#) attribute, you specify that the annotated member should be mapped to an OPC-UA attribute, and you optionally provide additional information about how the mapping should be done. The arguments that you can use with the [UADATA](#) attribute allow you to optionally specify:

- [Attributeld](#): Identifies an attribute of a node.
- [ValueType](#): The type of the OPC-UA Value attribute.
- [IndexRangeList](#): List of index ranges for individual array dimensions.
- [Kind](#): Specifies how the item will be mapped (see 7.1.9Mapping Kinds).
- [Operations](#): The item mapping operations in which the mapping will participate (see 7.1.8Mapping Operations).

Note that with the [UADATA](#) attribute, you do not specify information about which OPC-UA node should be mapped: This information comes from the usage of [UANode](#) attribute(s).

Typically, when the structure of your objects follows the structure of the OPC address space, you will annotate object members as follows:

- Members that correspond to OPC branches (folders): In OPC Classic, use the [DANode](#) attribute. In OPC-UA, use the [UANode](#) attribute.
- Members that correspond to OPC leaves (items, variables): In OPC Classic, use the [DANode](#) and [DALitem](#) attributes together. In OPC-UA, use the [UANode](#) and [UADATA](#) attributes together.

It is, however, perfectly possible (and indeed, practical) that your objects may have a structure that differs from a structure of the OPC address space – either just at some places, or globally. In such case, you can still make your objects "fit" the address space, by proper placing of all attributes described here.

## 7.1.6.5 Browse Path and Node Id Resolution

Items in OPC Data Access are ultimately identified by their item IDs (node IDs), which are strings defined by the server. Their syntax is not standardized, and although they commonly look like a sequence of names or numbers with some separators interspersed (such as "Boiler.Drum1001.LIX001.Output"), there is no general way to construct the item IDs

without browsing the OPC server's address space.

In OPC-UA, nodes have their Node Ids, which are a bit more complex (they are qualified by a namespace, and can be of several types, not just strings) – but effectively, they behave similarly. Their values are not generally standardized; they are completely determined by the concrete OPC-UA server you are connecting to.

When mapping, you could specify the precise node ID with the [DANode](#) (or [UANode](#)) attribute. This may work in some scenarios, but has severe limitations. The OPC address space (and your object structure) typically contains repeated patterns of nodes that are just placed differently. You want to define and map object types that match these patterns, but because the OPC node IDs will differ based on the placement of the pattern in the OPC address space, you won't be able to tell a unique item ID for each member.

This problem can be solved by using browse paths – sequences of item names that start from a known place in the OPC address space (e.g. in its root). The names in browse paths are "short" names that are used when browsing and sometimes displayed to the user, and the "short" node name typically remains the same if the node with the same meaning (a part of node pattern) is placed differently in the OPC address space. The mapping can translate the browse paths to item IDs when they are actually needed. For more on browse paths, see 4.11.7Browse Paths.

In order to specify nodes using a browse path, use the [BrowsePath](#) argument of the [DANode](#) (or [UANode](#)) attribute. It is a string that can contain an absolute or relative browse path. An absolute browse path string has similar limitations as the item ID; you may find it useful to define the starting point for the mapping (always needed in OPC-UA), or if the mapping does not start at the root of the OPC address space (in OPC Classic), though.

With a relative browse path string, you can specify that the member annotated with [DANode](#) (or [UANode](#)) attribute will belong to an OPC node that is based on the current OPC node, but appended with the relative browse path given. The relative browse path string can contain a single "short" item name, or multiple separated names forming a longer path.

For example, if the current OPC node (for the parent) in OPC Classic is given by browse path '/Boiler/Drum1001/LIX001', and your member is annotated with [DANode](#) attribute with [BrowsePath](#) = "Output", the OPC node for the member will be given by a browse path '/Boiler/Drum1001/LIX001/Output', which can then be translated to a full item ID. When the same type containing your member is used within a different context (at different place in the OPC address space), the resulting browse path will be different, but still contain "Output" node at the end. This is how repeated node patterns can be mapped without having to specify the individual item IDs.

By default, when the [DANode](#) (or [UANode](#)) attribute is used without further arguments, the name of the member being annotated is used as the relative browse path string for the [DANode](#) (or [UANode](#)) attribute. This means that if you name your members the same as the nodes in the OPC address space, you do not have to provide any additional arguments to the [DANode](#) (or [UANode](#)) attribute. This is a common practice, but if some members cannot follow it, you can still specify a different browse path just for them.

In some cases, the use of browse paths is not possible or practical, but you can define a "rule" specifying how the node IDs are constructed for the concrete OPC server and concrete part of its address space. In such case, you can use the [DAItemIdTemplate](#) (or [UANoditemIdTemplate](#)) attribute to annotate your type, giving it an [ItemIdTemplateString](#) (or a [NoditemIdTemplateString](#), in OPC-UA) argument that specifies how a node ID of a node is made of the parent node Id, and browse name of the node being mapped (a member name, typically).

Following macros can be used in template strings for [DAItemIdTemplate](#) attribute:

- **\$(ParentItemId)**: The OPC Item Id of the parent node.
- **\$(ParentNodePath)**: The OPC node path of the parent node (only in OPC XML-DA).
- **\$(BrowseName)**: The browse name of current node.

Following macros can be used in template strings for [UANoditemIdTemplate](#) attribute:

- **\$(ParentNodeId)**: The node Id of the parent OPC-UA node.
- **\$(BrowseName)**: The browse name of current node.

Final note: In OPC Classic, it is possible to specify combinations of [BrowsePath](#) and [ItemId](#) arguments of the [DANode](#) attribute, and further combine it with the [DAItemIdTemplate](#) attribute. In OPC-UA, it is possible to specify combinations of [BrowsePath](#) and [NoditemId](#) arguments of the [UANode](#) attribute, and further combine it with the

[UANodeIdTemplate](#) attribute.

In such cases, the information specified at the “deepest” level prevails. If both the browse path and node ID are specified for a node, then both are passed to the underlying OPC client object for processing. Which one of them will be finally used depends on many factors, including the concrete capabilities of the server and the version(s) of OPC specifications it conforms to, so in such case, you are responsible for guaranteeing that both the browse path and the item ID represent the same node.

## 7.1.6.6 Meta- Members

Sometimes you may have a need to obtain additional information that comes from the mapping process; such as what were the resulting values of propagated parameters, or what is the precise node the current object is being mapped to (this is useful when the object is used repeatedly for patterns of nodes in the OPC address space). In order to have this information stored into members of your objects, you can use *meta-members*.

Annotating a member (field, method or property) with the [MetaMember](#) attribute causes it to be set to named information gathered during mapping (a meta-information).

The [MetaMember](#) attribute has a [Name](#) argument that denotes which meta-information will be used.

Meta-members in the following table are available for both OPC Classic and OPC-UA:

Name	Type	Description
Mappings	<a href="#">IEnumerable&lt;AbstractMapping&gt;</a>	An enumeration of mappings defined directly on the target object.
Parent	<a href="#">Object</a>	A reference to the parent target object; <b>null</b> if none.

In addition, meta-members in the following table are available for OPC Classic:

Name	Type	Description
GroupParameters	<a href="#">DAGroupParameters</a>	An object containing subscription parameters, such as the requested update rate.
NodeDescriptor	<a href="#">DANodeDescriptor</a>	The descriptor of the OPC node involved in the operation.
ReadParameters	<a href="#">DAReadParameters</a>	The read parameters (such as data source or value age).
ServerDescriptor	<a href="#">ServerDescriptor</a>	An OPC server involved in the operation.

In addition, meta-members in the following table are available for OPC-UA:

Name	Type	Description
EndpointDescriptor	<a href="#">UAEndpointDescriptor</a>	An OPC-UA server involved in the operation.
MonitoringParameters	<a href="#">UAMonitoringParameters</a>	An object containing monitoring parameters, such as the sampling interval.
NodeDescriptor	<a href="#">UANodeDescriptor</a>	The descriptor of the OPC-UA node

Name	Type	Description
		involved in the operation.
ReadParameters	<a href="#">UAReadParameters</a>	The read parameters (such as the maximum age of the value).
SubscriptionParameters	<a href="#">UASubscriptionParameters</a>	An object containing subscription parameters, such as the publishing interval.

All meta-information is constant after being established, and changing it later has no effect on the underlying mappings.

## 7.1.6.7 Mapping Tags

Mapped members can be tagged with one or more tags. When performing operations, the developer can specify that only members with certain tag, or combination of tags, will be affected.

A mapping tag is any non-empty string.

In order to place a mapping tag on any mapped member, use the [MappingTag](#) attribute with it. You can repeat the [MappingTag](#) attribute on the member, if you wish the member be tagged with multiple tags.

If you are creating library objects that can be shared with other projects or developers, you are responsible for making sure that the same tag is not accidentally used for different purposes. It is therefore recommended to design a scheme that prevents such conflicts. For example, you may use a namespace approach, and have tags such as "MyCompany.MyProduct.MyComponent.MyTag".

## 7.1.6.8 Deferred Mapping

With deferred mapping, the mapped node object does not have to exist at the time of mapping, but is determined later, when it is actually needed. This is useful e.g. if it is costly to create the full object graph upfront, either because it is very large, or because it depends on dynamic factors that are not known at the time of mapping.

In order to use deferred mapping, you need to:

1. Specify [Deferred](#) = [true](#) with the [DANode](#) or [UANode](#) attribute. This causes the mapped member be set to a function ("deferred mapping function") that returns the mapped object (normally, the mapped member contains the mapped object itself).
2. Set the [DeferredMapNodeFunction](#) of the [DAClientMapper](#) or [UAClientMapper](#) to a function that returns the deferred mapping function (see above) for a given mapping context.

The deferred mapping is used in OPC-UA Modelling to dynamically provide node objects whose types depend on the information model of the OPC-UA server.

## 7.1.7 Mapped Node Classes

The [DAMappedNode](#) class is a pre-made base class for mapping OPC Data Access nodes. The [UAMappedNode](#) class is a pre-made base class for mapping OPC-UA data nodes.

You may (but do not have to) use this class as a base class for your objects that you map to OPC Data Access (or OPC-UA data) nodes. The class is already annotated with the [DAType](#) (or [UAType](#)) attribute, and contains properties that are mapped to useful meta-members, such as the [NodeDescriptor](#) property.

Sometimes it is not possible to use these mapped node classes, usually because you already have a class hierarchy

where your objects are derived from a different base that you cannot change. In other cases, however, the [DAMappedNode](#) (or [UAMappedNode](#)) class serves as a handy shortcut that can make your code shorter.

## 7.1.8 Mapping Operations

There are several things you can do with OPC Data: you can read from them, write to them, or subscribe to their changes. Not all mappings should be involved in all operations, and you therefore need to specify the operations that are of interest to you.

For [DAClientItemMapping](#) (OPC Data Access items), doing this means properly setting the [Operations](#) argument of the [DALitem](#) attribute. You can set it to one of the following values, or their combination:

- [DALitemMappingOperations.Read](#): A Read operation.
- [DALitemMappingOperations.Write](#): A Write operation.
- [DALitemMappingOperations.Subscribe](#): A Subscribe operation.

If you want a combination, you can use a logical OR of the individual values, or use a predefined symbolic name such as [ReadAndWrite](#), [ReadAndSubscribe](#), etc. By default, the [Operations](#) argument is set to [DALitemMappingOperations.All](#), meaning that the mapping will be involved in all operations.

There is just one mapping operation for [DAClientPropertyMapping](#) (OPC Data Access properties) – the “Get” operation that obtains the value of the OPC property. For OPC-DA property mappings, it is therefore neither necessary nor possible to specify the operation(s) that certain mapping should be involved in.

For [UAClientDataMapping](#) (in OPC-UA), you specify the operations by properly setting the [Operations](#) argument of the [UAData](#) attribute. You can set it to one of the following values, or their combination:

- [UADataMappingOperations.Read](#): A Read operation.
- [UADataMappingOperations.Write](#): A Write operation.
- [UADataMappingOperations.Subscribe](#): A Subscribe operation.

The operations do not get actually executed until you write and run a code that does it. See [Invoking the Operations](#) for details.

## 7.1.9 Mapping Kinds

When you describe that certain mapping target (e.g. a property) should be mapped to certain mapping source (e.g. an OPC item), there is still a piece of information missing: What exactly is the content of that mapping target? For example, with OPC-DA item, does the mapped element contain the value of the OPC item itself, or the quality, or does it contain the value/timestamp/quality combination, or even something else?

In order to distinguish between these, the live mapping has a concept of a mapping kind. You can choose which kind of mapping to use for certain mapping target. You can also map the same source to multiple targets, each time with a different mapping kind. Using this approach, you can e.g. map the value, timestamp, and quality, each to a separate property.

The mapping kind is specified using the [Kind](#) property of the [DALitem](#) attribute (for OPC-DA item mappings) on the target member or on the [UAData](#) attribute (OPC UA data mappings).

The following table lists all possible mapping for OPC-DA item mappings. When no mapping kind is specified, the default mapping kind of [Value](#) is used.

Kind	Type	Description
Result	<a href="#">OperationResult</a> or a derived type	Map the operation result.

Kind	Type	Description
ErrorCode	Int32	Map the error code. Zero for success, negative values for errors, and positive values for warnings.
Exception	Exception or a derived type	Map the exception. null if the operation has been successful.
ErrorMessage	String	Map the error message. An empty string if the operation has been successful.
StatusInfo	StatusInfo	Map the status information.
Vtq	DAVtq or DAVtq<T>	Map the item value/timestamp/quality combination. See Note 1.
Value	Object or T	Map the item value. This is the default mapping kind. See Note 1.
Timestamp	DateTime	Map the timestamp. In UTC. See Note 1.
TimestampLocal	DateTime	Map the timestamp. In local time. See Note 1.
Quality	DAQuality	Map the OPC quality. See Note 1.

For OPC-DA property mappings, the mapping kind is specified using the [Kind](#) property of the [DAProperty](#) attribute on the target member. The following table lists all possible mapping for OPC-DA property mappings. When no mapping kind is specified, the default mapping kind of [Value](#) is used.

Kind	Type	Description
Result	OperationResult or a derived type	Map the operation result.
ErrorCode	Int32	Map the error code. Zero for success, negative values for errors, and positive values for warnings.
Exception	Exception or a derived type	Map the exception. null if the operation has been successful.
ErrorMessage	String	Map the error message. An empty string if the operation has been successful.
StatusInfo	StatusInfo	Map the status information.
Value	Object or T	Map the property value. This is the default mapping kind. See Note 1.

The following table lists all possible mapping for OPC-UA data mappings. When no mapping kind is specified, the default mapping kind of [Value](#) is used.

Kind	Type	Description
Result	OperationResult or a derived type	Map the operation result.
ErrorCode	Int32	Map the error code. Zero for success, negative values for errors, and positive values for warnings.
Exception	Exception or a derived type	Map the exception. null if the operation has been successful.
ErrorMessage	String	Map the error message. An empty string if the operation

Kind	Type	Description
		has been successful.
StatusInfo	StatusInfo	Map the status information.
AttributeData	UAAttributeData or UAAttributeData <T>	Map the attribute value/source timestamp/server timestamp/status code combination. See Note 1.
Value	Object or T	Map the item value. This is the default mapping kind. See Note 1.
ServerTimestamp	DateTime	Map the server timestamp. In UTC. See Note 1.
ServerTimestampLocal	DateTime	Map the server timestamp. In local time. See Note 1.
SourceTimestamp	DateTime	Map the source timestamp. In UTC. See Note 1.
SourceTimestampLocal	DateTime	Map the source timestamp. In local time. See Note 1.
StatusCode	UAStatusCode	Map the status code. See Note 1.

Note 1: The mapping target value is not changed if the operation resulted in an exception. This means that when an error occurs, the mapping target with this mapping kind will not be reset to null or other default value; in fact it won't be influenced at all. E.g. if an item's value (a property annotated with mapping kind of [Value](#)) is successfully obtained once, and then an error occurs in a subsequent operation, the target property will remain unchanged. You can detect the error by additional mappings with different mapping kinds (e.g. [Exception](#)).

## 7.1.10 Mapping Arguments and Phases

When a mapping operation, such as Read, Write is performed, the execution is made in certain phases. First, the arguments for the operation are obtained from the targets, the actual operation then takes places, and after that, the results are stored to the targets again. For operations such as Subscribe, when the mapping source generates an asynchronous event, similar phases exist during the processing of the notification as well.

For the purpose of clarity, we will call the individual phases as follows:

- **Input:** Before the mapping operation is executed, values are obtained from the mapping target (and then send to the mapping source).
- **Output:** After the mapping operation has been executed, values (received from the mapping source) are stored to the mapping target.
- **Notification Input:** When an asynchronous notification arrives from the mapping source, the values contained in the notification are stored to the mapping target.
- **Notification Output:** When an asynchronous notification from the mapping source is finalized, it is updated with the values obtained from the mapping target. Not currently used.

You can map some of Outputs and Notification Inputs, or none, or all of them – all these approaches are valid, because it is up to you how you handle the data that comes from the mapping source. With Inputs (and Notification Outputs), you must, however, obey the rules that exist for every mapping operation, and map the proper combination of them, so that the mapping source can receive the data it expects.

The following table shows how (depending on the item mapping kind) the [Read](#), [Write](#) and [Subscribe](#) operations treat their mapping arguments for OPC-DA item mappings.

Mapping Kind	Mapping Operation		
	Read	Write	Subscribe
Result	Output	Output	Notification Input

ErrorCode	Output	Output	Notification Input
Exception	Output	Output	Notification Input
ErrorMessage	Output	Output	Notification Input
StatusInfo	Output	Output	Notification Input
Vtq	Output	Input	Notification Input
Value	Output	Input	Notification Input
Timestamp	Output	Input	Notification Input
TimestampLocal	Output	Input	Notification Input
Quality	Output	Input	Notification Input

The Write operation in OPC-DA can either write just a value, or a value/timestamp/quality triple. Only following combinations of inputs are therefore valid for the OPC-DA Write operation:

Vtq	Value	Timestamp or TimestampLocal	Quality	Outcome
✓				Value/timestamp/quality triple is written.
	✓	✓	✓	
	✓			Just the value is written.

For OPC-DA property mappings, there is just one mapping operation ([Get](#)), and the corresponding table is trivial:

Mapping Kind	Mapping Operation
Result	<a href="#">Get</a>
ErrorCode	Output
Exception	Output
ErrorMessage	Output
StatusInfo	Output
Value	Output

The following table shows how (depending on the item mapping kind) the [Read](#), [Write](#) and [Subscribe](#) operations treat their mapping arguments for OPC-UA data mappings.

Mapping Kind	Mapping Operation		
Mapping Kind	Read	Write	Subscribe
Result	Output	Output	Notification Input
ErrorCode	Output	Output	Notification Input
Exception	Output	Output	Notification Input
ErrorMessage	Output	Output	Notification Input
StatusInfo	Output	Output	Notification Input

AttributeData	Output	Input	Notification Input
Value	Output	Input	Notification Input
ServerTimestamp	Output	Input	Notification Input
ServerTimestampLocal	Output	Input	Notification Input
SourceTimestamp	Output	Input	Notification Input
SourceTimestampLocal	Output	Input	Notification Input
StatusCode	Output	Input	Notification Input

The Write operation in OPC-UA can either write just a value, or a value/server timestamp/source timestamp/status code quadruple. Only following combinations of inputs are therefore valid for the OPC-UA Write operation:

AttributeData	Value	ServerTimestamp or ServerTimestampLocal	SourceTimestamp or SourceTimestampLocal	StatusCodes	Outcome
✓					Value/server timestamp/source timestamp/status code quadruple is written.
	✓	✓	✓	✓	Just the value is written.

## 7.1.11 The Mapper Object

The mapper object is at the center of the live mapping mode. You will typically use it for two main purposes:

1. Establish the correspondences between the source OPC data, and you target .NET objects: see [Mapping Your Objects](#).
2. Perform various operations using the mappings established earlier: [Invoking the Operations](#).

For mapping of OPC Data Access (OPC-DA) sources, the actual mapper class is [DAClientMapper](#). You can create an instance of [DAClientMapper](#) simply by instantiating it with one of its constructor overloads. You can also use a static [DAClientMapper.SharedInstance](#) object and avoid the need of instantiation.

For mapping of OPC Unified Architecture (OPC-UA) sources, the actual mapper class is [UAClientMapper](#). You can create an instance of [UAClientMapper](#) simply by instantiating it with one of its constructor overloads. You can also use a static [UAClientMapper.SharedInstance](#) object and avoid the need of instantiation.

### 7.1.11.1 Mapping Your Objects

In order to create the mappings, i.e. associations between mapping sources (OPC data) and mapping targets (your object members), you typically call one of the [Map](#) method overrides on the mapper object ([DAClientMapper](#), for OPC Data Access).

For the mapping, you can either have the mapper use a copy of its mapping context template (defined by its [MappingContextTemplate](#) property), or pass in your own Mapping Context object (see further below for details of the mapping context). In both cases, you need to specify the object to be mapped. The mapper then uses its knowledge about the object type (as given by the mapping attributes) to create and add to itself a set of mappings, one mapping for each mapped member in your object structure.

It is also possible to "undo" the mapping. To do so, you typically call one of the Unmap method overrides on the mapper object ([DAClientMapper](#), for OPC Data Access). You can either un-map all mappings that currently exist in the mapper (if you pass no arguments), or un-map specific mappings.

## 7.1.11.2 Mapping Context

The mapping context holds information about the state of the mapping process. In OPC Data Access (OPC-DA), the mapping context is represented by an instance of [DAMappingContext](#) class. In OPC Unified Architecture (OPC-UA), the mapping context is represented by an instance of [UAMappingContext](#) class.

The state of mapping changes in the mapping context, as the mapping process proceeds from your given "starting" object, to its possible constituent objects.

For example, the mapping context contains the information about the OPC server and current OPC node, and parameters for reading and subscription. As the mapping proceeds to deeper level, the properties in the mapping context are propagated to the deeper levels as well, and they change accordingly. For example, if a member is mapped to an OPC-DA node, the [NodeDescriptor](#) property of the [DAMappingContext](#) changes to reflect the information about the node. If the member represents an OPC branch and contains another object, this new [NodeDescriptor](#) will be the basis for mapping the members of the inner object.

Refer to Propagated Parameters for descriptions of attributes that are maintained in the mapping context.

You need to make sure that enough information is provided for the mapping, either through the "starting" mapping context used with the Map method, or by means of the mapping attributes. Typically, for example, the OPC server's computer and ProgID (or endpoint URL) will not be specified through the mapping attributes, but will come from your code, so that it can be modified in run-time. In OPC-DA, you will therefore create a new [DAMappingContext](#) with properly filled [ServerDescriptor](#), and pass it to the [Map](#) method. In OPC-UA, you will create a new [UAMappingContext](#) with properly filled [EndpointDescriptor](#), and pass it to the [Map](#) method.

## 7.1.11.3 Invoking the Operations

Once you have your objects mapped, you can invoke operations on them. The operations available depend on the type of mapper object.

For OPC Data Access (the [DAClientMapper](#) class), following methods exist for invoking operations:

- The [Get](#) method executes the "Get", i.e. gets the values of OPC-DA properties.
- The [Read](#) method executes the "Read", i.e. reads the data of OPC-DA items.
- The [Subscribe](#) method executes the "Subscribe"; depending on its Boolean arguments, it either subscribes to or unsubscribes from changes of OPC-DA items.
- The [Write](#) method executes the "Write", i.e. writes data into the OPC-DA items.

For OPC Unified Architecture (the [UAClientMapper](#) class), following methods exist for invoking operations:

- The [Read](#) method executes the "Read", i.e. reads the data of OPC-UA node attributes.
- The [Subscribe](#) method executes the "Subscribe"; depending on its Boolean arguments, it either subscribes to or unsubscribes from changes of OPC-UA node attributes (monitored items).
- The [Write](#) method executes the "Write", i.e. writes data into the OPC-UA node attributes.

Each operation can be invoked either on all applicable mappings currently existing in the mapper, or on a specified subset of mappings. For example, the [Read](#) method on [DAClientMapper](#) has two overloads, one with no arguments (reads all item mappings that have [DAMappingOperations.Read](#) included in their [Operations](#) property), and one with the [IEnumerable<IDAClientMapping>](#) argument (reads just the specified mappings).

Since you typically do not have direct access to the individual mappings created for your objects (except with Type-less Mapping), you might be asking how to get access to the subset of mappings that you need. For example, you may need to perform an OPC Read on just certain mapped object, somewhere inside the whole hierarchy.

The way to obtain the set of mapping is to query the mapper itself. The [Mappings](#) property of the mapper contains an [IEnumerable](#)<> of all mappings currently maintained by the mapper. You can filter these mappings in any way you like, testing whether they fulfill your application-defined criteria, and then perform the operation just with the filtered subset.

In order to make this task easier, some extension method exist. Following methods are available for all OPC specifications:

- [AbstractMappingExtension.GetTargetObject](#) method determines the target object of the mapping. You can use it to test whether the mapping is for "your" object.
- [AbstractMappingExtension.BelongsTo](#) method determines whether a given target object belongs to this mapping, either directly, or recursively (to any of the parent mappings).

For OPC Data Access, you can use following extension methods:

- [DAClientMapperExtension.GetTarget](#) method executes the OPC-DA "Get" operation on member of given target object, directly or recursively.
- [DAClientMapperExtension.ReadTarget](#) method executes the OPC-DA "Read" operation on member of given target object, directly or recursively.
- [DAClientMapperExtension.SubscribeTarget](#) method executes the OPC-DA "Subscribe" operation on member of given target object, directly or recursively.
- [DAClientMapperExtension.WriteTarget](#) method executes the OPC-DA "Write" operation on member of given target object, directly or recursively.

For OPC Unified Architecture, you can use the following extension methods:

- [UAMapperExtension.ReadTarget](#) method executes the OPC-UA "Read" operation on member of given target object, directly or recursively.
- [UAMapperExtension.SubscribeTarget](#) method executes the OPC-UA "Subscribe" operation on member of given target object, directly or recursively.
- [UAMapperExtension.WriteTarget](#) method executes the OPC-UA "Write" operation on member of given target object, directly or recursively.

With each of the extension methods on the [DAClientMapperExtension](#) or [UAMapperExtension](#), you can also specify that just members with given mapping tag, or with tags fulfilling certain condition (predicate) will be affected.

The [DAClientMapper](#) and [UAClientMapper](#) objects have a [Boolean QueuedExecution](#) property (defaults to false). When set to 'true', the operations on the mapper are executed outside the calling thread, but still in the order of arrival. This allows non-blocking operation calls on the mappers.

## 7.1.11.4 Type-less Mapping

While less common, you can actually using the live mapping mechanism for establishing a direct correspondence between mapping sources and mapping targets, without having to map the types. This may be beneficial e.g. if your application just needs to pick a small number of "scattered" items to be mapped from a large OPC address space, and at the same time you won't gain much from defining the types for objects that exist in the OPC address space.

In order to use the type-less mapping, you need to call the [DefineMapping](#) method on the mapper object, passing it the three essential pieces of information: The mapping source, the mapping itself (not yet associated with the source and target), and the mapping target. The [DefineMapping](#) method associates the mapping with its source and target, and adds the mapping to the mapper.

Following example shows how to define a new type-less mapping which maps a specified OPC-DA item to the [Value](#) member of your target [MyClass](#) object. The mapper is then instructed to invoke the OPC read operation, which will in turn store the OPC item's value to your target object.

Type-less mapping (OPC UA)

```
var mapper = new DAClientMapper();
var target = new MyClass();

// Define a type-less mapping.

mapper.DefineMapping(
    new DAClientItemSource("OPCLabs.KitServer.2", "Simulation.Register_I4",
    DADataSource.Cache),
    new DAClientItemMapping(typeof(Int32)),
    new ObjectMemberLinkingTarget(target.GetType(), target, "Value"));

// Perform a read operation.
mapper.Read();
```

Following example shows similar code for OPC Unified Architecture (OPC-UA):

## Type-less mapping (OPC UA)

```
var mapper = new UAClientMapper();
var target = new MyClass();

// Define a type-less mapping.

var memberInfo = target.GetType().GetSingleMember("Value");
Debug.Assert(memberInfo != null);

mapper.DefineMapping(
    new UAClientDataMappingSource(
        "opc.tcp://opcua.demo-this.com:51210/UA/SampleServer",
        "nsu=http://test.org/UA/Data/;i=10389",
        UAAttributeId.Value,
        UAIndexRangeList.Empty,
        UAReadParameters.CacheMaximumAge),
    new UAClientDataMapping(typeof(Int32)),
    new ObjectMemberLinkingTarget(target.GetType(), target, memberInfo));

// Perform a read operation.
mapper.Read();
```

In order to remove the mapping added in this way, call the [UDefineMapping](#) method on the mapper object.

Following example shows how to subscribe and unsubscribe:

## Subscribe and subscribe with type-less mapping

```
// This example for OPC DA type-less mapping shows how to define a mapping and
// perform subscribe and unsubscribe operations.
using System;
using System.Threading;
using OpcLabs.BaseLib.ComponentModel.Linking;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.LiveMapping;
```

```
namespace DocExamples
{
    namespace _DAClientMapper
    {
        partial class DefineMapping
        {
            class MyClassSubscribe
            {
                public Double Value
                {
                    set
                    {
                        // Display the incoming value
                        Console.WriteLine(value);
                    }
                }
            }

            public static void Subscribe()
            {
                var mapper = new DAClientMapper();
                var target = new MyClassSubscribe();

                // Define a type-less mapping.

                mapper.DefineMapping(
                    new DAClientItemSource("OPCLabs.KitServer.2", "Demo.Ramp", 1000,
DADDataSource.Cache),
                    new DAClientItemMapping(typeof(Double)),
                    new ObjectMemberLinkingTarget(target.GetType(), target,
"Value"));

                // Perform a subscribe operation.
                mapper.Subscribe(true);

                Thread.Sleep(30*1000);

                // Perform an unsubscribe operation.
                mapper.Subscribe(false);
            }
        }
    }
}
```

Following example shows how to make various kinds of mappings:

#### Various mapping kinds with type-less mapping

```
// This example for OPC DA type-less mapping shows how to define mappings of various
// kinds, and perform subscribe and
// unsubscribe operations.
using System;
```

```
using System.Threading;
using OpcLabs.BaseLib.ComponentModel.Linking;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.Generic;
using OpcLabs.EasyOpc.DataAccess.LiveMapping;

namespace DocExamples
{
    namespace _DAClientMapper
    {
        partial class DefineMapping
        {
            class MyClassMappingKinds
            {
                public Double CurrentValue
                {
                    set
                    {
                        // Display the incoming value
                        Console.WriteLine("Value: {0}", value);
                    }
                }

                public DAVtq<Double> CurrentVtq
                {
                    set
                    {
                        // Display the incoming Vtq
                        Console.WriteLine("Vtq: {0}", value);
                    }
                }

                public Exception CurrentException
                {
                    set
                    {
                        // Display the incoming exception
                        Console.WriteLine("Exception: {0}", value);
                    }
                }

                public DAVtqResult<Double> CurrentResult
                {
                    set
                    {
                        // Display the incoming result
                        Console.WriteLine("Result: {0}", value);
                    }
                }
            }
        }
    }
}
```

```
public static void MappingKinds()
{
    var mapper = new DAClientMapper();
    var target = new MyClassMappingKinds();

    // Define several type-less mappings for the same source, with
    // different mapping kinds.

    Type targetType = target.GetType();
    var source = new DAClientItemSource("OPCLabs.KitServer.2",
    "Demo.Ramp", 1000, DADataSource.Cache);

    mapper.DefineMapping(
        source,
        new DAClientItemMapping(typeof(Double)),
        new ObjectMemberLinkingTarget(targetType, target,
    "CurrentValue"));

    mapper.DefineMapping(
        source,
        new DAClientItemMapping(typeof(Double), DAIItemMappingKind.Vtq),
        new ObjectMemberLinkingTarget(targetType, target,
    "CurrentVtq"));

    mapper.DefineMapping(
        source,
        new DAClientItemMapping(typeof(Double),
    DAIItemMappingKind.Exception),
        new ObjectMemberLinkingTarget(targetType, target,
    "CurrentException"));

    mapper.DefineMapping(
        source,
        new DAClientItemMapping(typeof(Double),
    DAIItemMappingKind.Result),
        new ObjectMemberLinkingTarget(targetType, target,
    "CurrentResult"));

    // Perform a subscribe operation.
    mapper.Subscribe(true);

    Thread.Sleep(30*1000);

    // Perform an unsubscribe operation.
    mapper.Subscribe(false);
}

}
```

## 7.1.11.5 Error Handling in the Mapper Object

There are several types of errors that can occur in relation to live mapping, and they are treated differently. It is important to understand what happens in various situations, so that you can write code that behaves properly.

In the list below, we have tried to sort the possible types of errors by severity, starting with the most severe errors first.

- A. Errors during object mapping. These are errors that are encountered when you are associating your objects (mapping targets) with mapping sources, usually by means of calling one of the [Map](#) method overloads on the mapper object. Typically, these errors are caused by improper usage of mapping attributes, such that the mapping cannot be created at all. They are basically coding errors, and are reported by throwing a [MappingException](#) during the mapping (the [Map](#) method call).
- B. Operation execution errors. These are logical errors that happen when you invoke an operation on the mapper, such as [Read](#), [Write](#), [Get](#) or [Subscribe](#). Operation execution errors are similar to the errors encountered during object mapping, except that they could not have been discovered earlier. For example, you may have a conflicting combination of attributes that does not allow the OPC Write operation be performed. Operation execution errors are basically coding errors as well, and are reported by throwing an [ExecutionException](#) from the operation method invoked.
- C. Update failures. The update failure means that a mapping target could not be updated with a new value. Each update failure is indicated by an [UpdateFailure](#) event raised on the mapper object. The event notification contains an [Exception](#) indicating the cause of the update failure, and a reference to the mapping source. You can hook your event handler to this event if you need some action be performed in such case.

There may be various reasons for update failures; below are the common specific causes.

1. Target access errors. They happen when the live mapping encounters a failure accessing your objects (mapping targets). For example, setting or getting a property on your object may throw an exception, and such exception will be caught. Target access error is represented by an instance of [TargetException](#) in an update failure.
2. Validation errors. These are errors caused by invalid values passed to or from the mapping target. For example, you may be attempting to write a null reference [DAVtq](#) object, a null timestamp, or a null quality. Validation errors are represented by an instance of [ValidationException](#) in an update failure. Note that these are validations on the client side, not any validations made by the OPC server.
- D. Mapping source problems. These are errors related to the actual mapping sources, such as OPC items or OPC properties. For example, the OPC server may be shut down, or the item ID might have changed name and is no longer accessible. Mapping source problems may actually be THE important errors for you to handle, but they have the least severity as far as the live mapping is concerned, because they are treated as just another data being mapped (and sent to your application). There are mapping kinds (see Mapping Kinds) such as [Exception](#) or [ErrorCode](#) that allow you to set up members on your objects that will receive error information in case of mapping source problems.

## 7.2 OPC-UA Modelling (Preliminary)

This development model automatically provides .NET objects that logically correspond to OPC-UA information model. The operative parts of this model are internally implemented using Live Mapping.

Currently, only certain standard OPC-UA variable types are supported; custom types are not yet available. Supported concrete classes include: [UAPropertyNode](#), [UABaseDataVariable](#); for UA Data Access: [UADatalItemNode](#), [UAAnalogItemNode](#), [UATwoStateDiscreteNode](#), [UAMultiStateDiscreteNode](#).

### 7.2.1 How It Works

Here are the steps that you need to get started with using OPC-UA modelling approach in your code:

1. Create an instance of [UAModelClient](#) class that will use the information model resident in the OPC server. You obtain this [UAModelClient](#) object by calling a static method [UAModelClient.CreateWithServerModel\(\)](#).
2. When you want to work with a node in the OPC server, obtain the corresponding node object by calling the [GetNode](#) method on the [UAModelClient](#) object. There are also extension methods that are useful to obtain nodes with a type that you expect. For example, to obtain a node for an analog variable of type TValue, use the [GetAnalogItemNode<TValue>](#) extension method.
3.
  - a. If you want to read data from the node, call the [ReadNode](#) method on the [UAModelClient](#) object, passing it the node object, and optionally the so-called modelling tag. Use [UAModellingTags.VariableValue](#) to operate only on the value of the variable. Use [UAModellingTags.VariableProperty](#) to operate on all variable properties. After the [ReadNode](#) call, the properties on the node object contain the values obtained. For example, the [Value](#) property contains the current value of the variable. Or, the [EURange](#) property contains the engineering units range.
  - b. If you want to write data into the node, set the properties of the node objects (such as the [Value](#) property), and then call the [WriteNode](#) method on the [UAModelClient](#) object. You can optionally use the modelling tags, as with the [ReadNode](#) method.
  - c. If you want to subscribe to (or unsubscribe from) changes in the node data, call the [SubscribeNode](#) method on the [UAModelClient](#) object, passing it the "active" flag (determines whether you want to subscribe or unsubscribe) and the node object. You can optionally use the modelling tags, as with the [ReadNode](#) method. The properties of the node object (such as the [Value](#) property) will be automatically be updated with changes from the OPC server.

## 7.2.2 Node Types

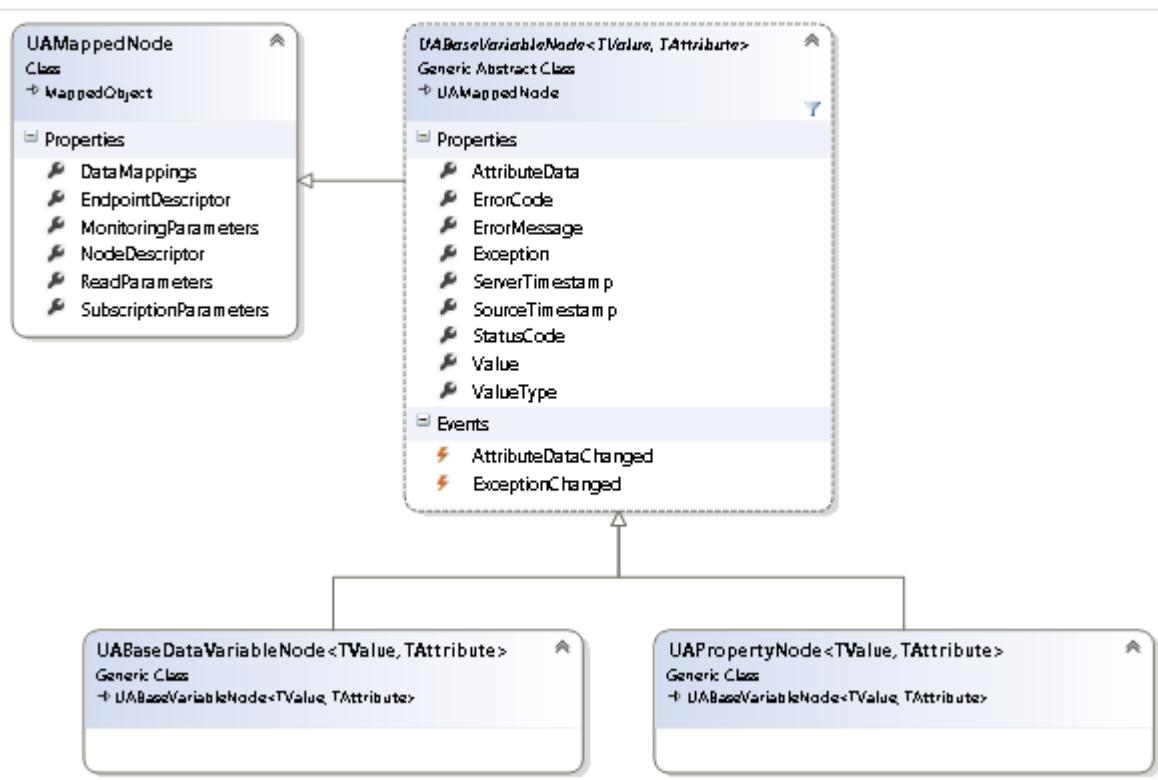
QuickOPC contains definitions of certain OPC-UA node types that can be used with the modelling. Currently, these are the only node types supported, and the list cannot be extended. Future QuickOPC versions will support the creation of use of custom node types with OPC-UA modelling.

The .NET classes for various OPC-UA node types use an inheritance scheme that follows the inheritance hierarchy in OPC-UA information model. The [UABaseVariable](#) is the root of this hierarchy; in itself, it derives from [UAMappedNode](#), which means that it contains meta-information about itself, such as the [EndpointDescriptor](#) or [NodeDescriptor](#), and also parameters that define details of operations that performed on the node – such as [ReadParameters](#), or [SubscriptionParameters](#) and [MonitoringParameters](#).

Each [UABaseVariable](#) contains properties that describe the state of the node either individually ([Value](#), [StatusCode](#), timestamps, ...), or in a summary way ([AttributeData](#)). There are also properties that give information about errors encountered during the operations ([Exception](#), [ErrorCode](#), [ErrorMessage](#)).

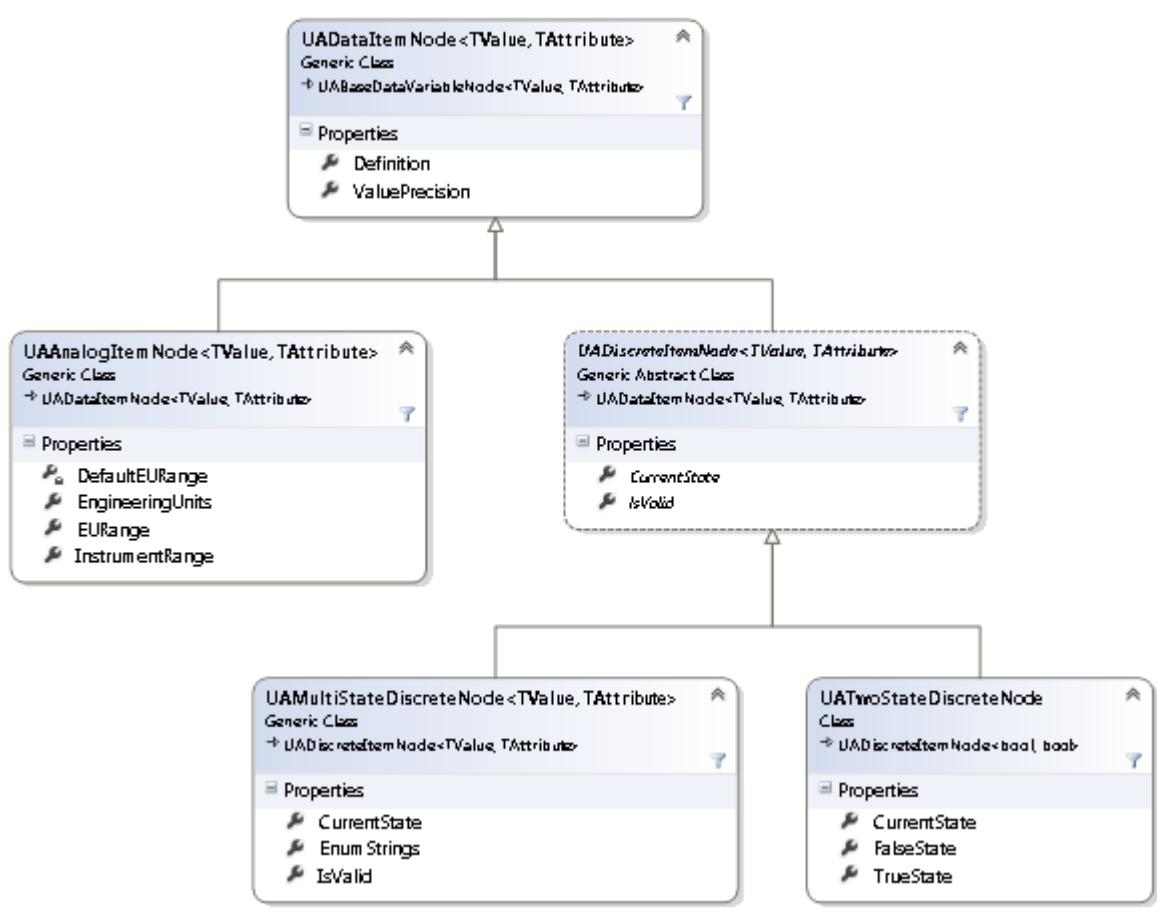
You can hook to [AttributeDataChanged](#) and [ExceptionChanged](#) events to get your code invoked whenever the relevant data of the OPC-UA variable changes.

The hierarchy of basic OPC-UA node type classes, and their most important members, are shown on the following diagram.



For use in Data Access, OPC-UA defines additional variable types, and they are reflected in QuickOPC .NET node types as well. When working with OPC-UA Data Access modelling, you will most likely use at least one of the following classes: [UAAnalogItemNode](#) (for analog variables), [UATwoStateDiscreteNode](#) (for digital – Boolean variables), or [UAMultiStateDiscreteNode](#) (for enumeration-type variables).

The hierarchy of OPC-UA Data Access node type classes, and their most important members, are shown on the following diagram.



The classes contain members that correspond to OPC-UA properties defined for these kind of variables. For example, the [Definition](#), [ValuePrecision](#), [EngineeringUnits](#), [EURange](#) etc. properties all contain information that further describes the variable. In addition, for discrete variables, the classes provide transformation between the "raw" values of the variable, and their string representation. The [CurrentState](#) property contains the current value of the variable, expressed as a string.

## 8 Live Binding Model

### 8.1 Live Binding Model for OPC Data (Classic and UA)

 With live binding development model, no manual coding is necessary to obtain OPC connectivity. You simply use the Visual Studio Designer to configure bindings between properties of visual or non-visual components, and OPC data. All functionality for OPC reads, writes, subscriptions etc. is provided by the QuickOPC components.

Note: Several binding technologies for.NET already exist, some directly from Microsoft. These technologies are mainly focused on binding database or similar data, and they are not well suitable for automation purposes like OPC. We therefore had to implement a new binding model. We use the term "live binding" so that it is clearly distinguished from other binding technologies used; it also reflects the fact that this kind of binding is well-suited for real-time, "live" data.

The Live Binding model is currently available for Windows Forms (or UI-less) containers only.

Do not confuse the live binding model with live mapping, also provided by QuickOPC. The main differences between the two are that

- live binding is for UI (User Interface), and is therefore visual, both in design and in runtime, and
- live binding works with individual object members, and not on whole types.

An important aspect of the live binding model is that it is not limited to or associated with any particular set of UI controls. It works by extending the features of existing controls, and providing functionality on top of what the controls already do. Live binding works well with standard Microsoft controls, but also work with controls from other vendors. This approach gives the live binding an enormous power, because you can combine the advanced features of free or commercial controls with the OPC connectivity.

In the examples installed with the product, we provide a demonstration of the integration capability described above. There are demos that show how Live Binding can be used with several 3<sup>rd</sup> party products, some aimed directly at industrial automation area.

Following resources will help you get started with Live Binding:

- The Getting Started section in this document: Contains instructions on how to create an application with live binding in a few steps.
- For OPC-DA, the "Live Binding Demo" application that is installed with the product; its source code is contained in the WindowsForms\LiveBindingDemo project. The application contains annotated live binding configurations for typical scenarios.
- For OPC-UA, the "UA Live Binding Demo" application that is installed with the product; its source code is contained in the WindowsForms\UALiveBindingDemo project. The application contains annotated live binding configurations for typical scenarios.

#### 8.1.1 Live Binding Overview

The live binding allows you to extend your controls and components with one or more *bindings*. The binding can perform various *operations*, such as read or write values, or continuously update (subscribe). The data reside in the *binding source* – e.g. in OPC Data Access or OPC Unified Architecture server, and are provided through a *connectivity* component for that binding source. The other end of the binding is the *binding target*. The usual binding targets can accept or provide values of various types, and are therefore referred to as *value targets*. When values are transferred between to or from the target, *value conversions* can be made – using *string formats*, or *converters*.

The live binding functionality is provided by a generic *binding extender*. The specific types of bindings are made available by *binding providers*, which are encapsulated in *binder* components. The binding extender keeps *binding*

bags, i.e. collections of bindings for each control being extended.

Operations on bindings can be tied to *event sources* and thus performed automatically with events in your application. In order to handle multiple bindings together, it is possible to define *binding groups* and assign bindings to them.

*Binding templates* can be defined to make it easier to create new bindings that share similar characteristics.

The objects and terms briefly mentioned above are described in more detail in the subsequent chapters.

## 8.1.1.1 Value Targets

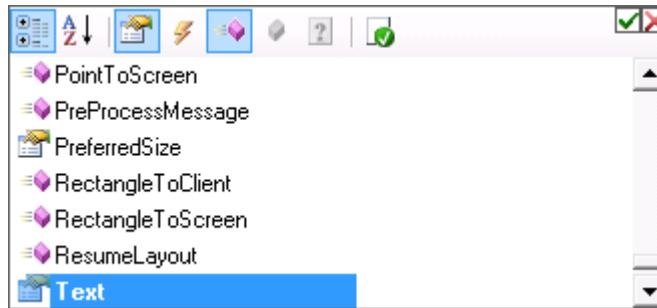
The *value target* represents a place in your application where values can be sent to or obtained from. You configure the value target by selecting the [TargetComponent](#) and [TargetMember](#). A typical value target is a property of a visual control on your form. For example, in order to send values to the [Text](#) property of the [TextBox](#) control, you would set the [TargetComponent](#) to the text box control itself, and the [TargetMember](#) to a string "Text".

It is also possible to target the so-called extender properties. In such case, you also need to select the [ExtenderProvider](#) component that actually provides the extender property. Typical and useful extender providers are the [ToolTip](#) and [ErrorProvider](#) components for Windows Forms.

When you select the [TargetComponent](#) (and the [TargetMember](#) is not set yet), the live binding system attempts to determine the [TargetMember](#) that you want to use. For example, the most commonly used property of the [TextBox](#) control is the [Text](#) property, so it makes sense to provide that as a default. This default member name is determined using several hints, including the [DefaultBindingProperty](#) and [DefaultProperty](#) attributes on the target component, and other characteristics.

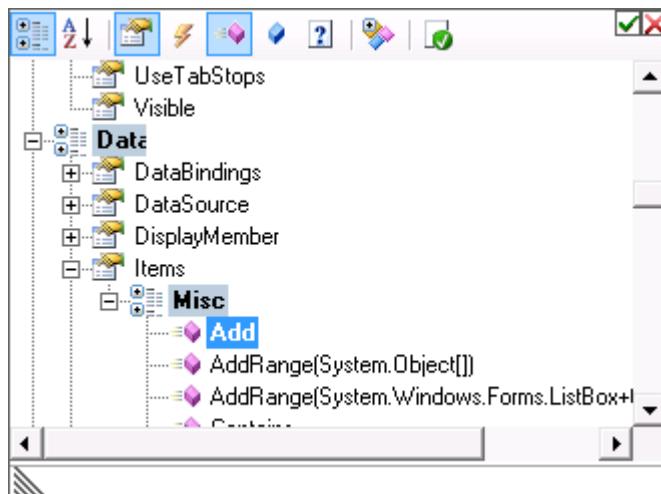
Not all controls (especially 3<sup>rd</sup> party controls) have their default members marked up (or, they may be marked up inappropriately for live binding). The live binding contains some built-in knowledge that allows proper selection of the default target member for popular controls, even in such cases. If you still get an improper target member selected for you by default, you can always change it to any control's member you like.

You can type in the name of the target member, or you can select it using the drop-down editor (picture below).



The member name editor shows different icons for different member kinds, annotates the members with number of overloads if needed, shows member information (such as type, and description) in the tooltip, allows filtering (e.g. Properties, Event, Methods, Fields), alphabetical or categorized view, and reverting to default member.

When binding .NET members, you can specify not just a member name, but a whole path to it – i.e. it is possible to bind sub-members as well. The path is a sequence of member names, separated by dots. For example, for binding on a [ListBox](#) component, it is possible to set the target path to "Items.Add", which means that we bind the [Add](#) method of the [Items](#) collection of the [ListBox](#) (in effect, this will append incoming values as new values to the [ListBox](#)). If you want to specify the path (sub-members) and not just a single member, you use the [TargetPath](#) property instead of just [TargetMember](#). By pressing a down-arrow symbol next to the path, you are offered a tree of available members and sub-members, with various filtering and sorting capabilities:



When binding .NET members (methods) that have multiple overloads, it is possible to specify the desired method signature using a type name in parenthesis, added to the element path. For example, if you wanted to bind the [Add](#) method of the [Items](#) collection of a [ListView](#), there are two overloads that take one input argument, and the Live Binding would not know which one to use. By using a target path "Items.Add(System.String)", you will instruct the Live Binding to use the overload that accepts string as an input.

## 8.1.1.2 Binding Operations

There are several things you can do with OPC Data: you can read from them, write to them, or subscribe to their changes. Not all bindings should be involved in all operations, and you therefore need to specify the operations that are of interest to you.

You can choose one of the following values, or their combination:

- [PointBindingOperations.Read](#): A Read operation.
- [PointBindingOperations.Write](#): A Write operation.
- [PointBindingOperations.Subscribe](#): A Subscribe operation.

If you want a combination, you can use a predefined symbolic name such as [ReadWrite](#), [ReadAndSubscribe](#), etc. By default, the [BindingOperations](#) argument is set to [PointBindingOperations.ReadAndSubscribe](#), meaning that the binding will be involved in Reads and Subscribes, but not in Writes.

## 8.1.1.3 Bindings

A binding is a correspondence between two entities that allows information be passed between them (either in one of the directions, or in both directions).

The main parts of binding are:

- **(Binding) Source:** The external data you want to bind to. Here you select e.g. the OPC item, update rate etc. You can also select which "part" of the data you want to bind to (e.g. just the value, or just the quality information, etc.).
- **(Binding) Target:** Defines element in your application that will be bound to the binding source. This is typically a value target (see Value Targets).
- **Binding Operations:** Which operations the binding is involved in.
- **Binding Groups:** Which binding group the binding belongs to.
- **Value Conversion:** Allows you to convert the values on their way to/from the binding target.
- **Event Links:** Defines events in your application that will automatically cause binding operation be performed.

The types of bindings that are available depend on the binder in use (see further below). For the (most common) PointBinder, there is just one type of binding available, a PointBinding. The point binding can be e.g. for an OPC-DA item, an OPC-DA property, or for OPC-UA data (node attribute).

There may be multiple bindings associated with a control. With this feature, you can, for example, bind a value (converted to string) to the actual text box's text, while binding the status information to its color.

## 8.1.1.4 Binding Bags

A binding bag is essentially a collection of bindings, with a reference to a common component to which the bindings belong. The live binding system keeps a binding for each component that is extended with bindings, and also allows you to access a common binding bag that represents all bindings defined in a container.

One binding bag can hold bindings of different types, and bindings that have different sources.

## 8.1.1.5 Binding Extender

The binding extender is a central piece of live binding model. It is represented by the [BindingExtender](#) component; this component must be present on your form in order to enable it for live binding.

The main purpose of the binding extender is to keep track of bindings defined on each control in the container (form), and provide extender properties to them.

The binding extender is generic; it does not have a specific knowledge about types of bindings that can be used. The available binding types are defined on the binding extender by a binder (or binders) that register with it. For explanation of binders, see further below.

## 8.1.1.6 Binders and Binding Providers

While the binding extender is a generic coordination component without specific knowledge about possible types of bindings, the *binding provider* is a component that is specific to certain technology and defines binding types tailored to that technology.

A binding provider has the ability to create one or more types of bindings, as necessary for the technology it covers. Binding providers exist in form of *binder* components; the binder is a binding provider (can create new bindings) with certain additional capabilities, such as event linking, and ability to perform operations on bindings.

Each binding provider needs to register itself with the binding extender; this is done by properly setting the [BindingExtender](#) property of the binder. When you drag the binder component from the Toolbox to your form's design surface, this setting is already made for you.

The most commonly used binder component is the [PointBinder](#), which is described further below in a separate chapter. The [PointBinder](#) provides a unified approach to live binding; it merges the commonalities between OPC "Classic" and OPC Unified Architecture bindings into a single component.

Note: Earlier, QuickOPC had used separate [DABinder](#) (for OPC Data Access) [UABinder](#) (for OPC Unified Architecture) component. They are now obsolete.

## 8.1.1.7 Event Sources

The *event source* represents a place in your application where events are generated. You configure the event source by selecting the [SourceComponent](#) and [SourceMember](#). A typical event source is a [Click](#) event of a [Button](#) on your form.

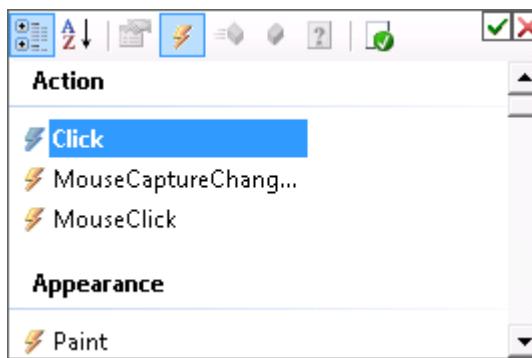
Event sources are used in live binding to automate actions that would otherwise have to be implemented in the code

by writing an event handler.

For example, if you are designing a form with HMI functionality, you probably want it to subscribe to value changes of the configured items when it is shown. This can be achieved by properly settings in the [OnlineEventSource](#) event source of the [BindingExtender](#) component (and indeed, when you drag the [BindingExtender](#) from the toolbox to a form, this event source is pre-set like this already).

Similarly, you may want to write some value into a bound OPC item when a button is clicked. Instead of writing an event handler, you can simply configure the [WriteEventSource](#) on the binding (or binding group) to the [Click](#) event of the [Button](#), and you are done.

You can type in the name of the source event, or you can select it using the drop-down editor (picture below).



The member name editor shows member information (such as type, and description) in the tooltip, and allows alphabetical or categorized view, and reverting to default member.

You can specify not just a member name, but a whole path to it – i.e. it is possible to bind sub-members as well. The path is a sequence of member names, separated by dots. If you want to specify the path (sub-members) and not just a single member, you use the [SourcePath](#) property instead of just [SourceMember](#). By pressing a down-arrow symbol next to the path, you are offered a tree of available members and sub-members, with various filtering and sorting capabilities.

## 8.1.1.8 Binding Groups

A *binding group* has the ability to handle multiple bindings together. For example, you can create a binding group, assign several bindings to it, and then have them all read or written by a single command.

The use of binding groups is optional. If you do not assign a binding to any binding group, it will belong to a *default binding group*, which always exists on the binder component.

## 8.1.2 Point Binder

The point binder (represented by a [PointBinder](#) component) provides live binding to generic data points (see further below). It supports just one type of binding – a [PointBinding](#); however, the point binding type is so generic that it can easily describe OPC-DA “Classic” items, properties, OPC-UA nodes (attributes), or just about any else point-based data source.

## 8.1.2.1 Connectivities

The *connectivity* is a component that provides access to a data source of some kind. It can (and usually does) support reading, writing, subscriptions, and browsing.

Three connectivity components are currently provided with QuickOPC: The [DACConnectivity](#) (for OPC Data Access), the [UAConnectivity](#) (for OPC Unified Architecture), and the [CompositeConnectivity](#) (can be configured to contain and

combine any number of other different connectivity components). The developer will usually start the live binding by dragging one of these connectivity components from the Visual Studio Toolbox to the design surface of the form.

Notice that after dragging any of the connectivity components to a new form, three icons will appear below the form: Besides the one for the connectivity itself, there will be one for the [PointBinder](#), and one for [BindingExtender](#).

If, after dragging a connectivity to a new form, the PointBinder and the [BindingExtender](#) components are not placed onto the form automatically, Make sure that the target framework of the project is set to ".NET Framework 4.5.2" or later.

## 8.1.2.2 Points

Each connectivity must somehow identify the pieces of data it works with. We call them *points*. A connectivity can work with one or more types of points. The composite connectivity merges together the point types of multiple constituent connectivities.

For OPC Data Access (with the [DAConnectivity](#) component), following types of points exist:

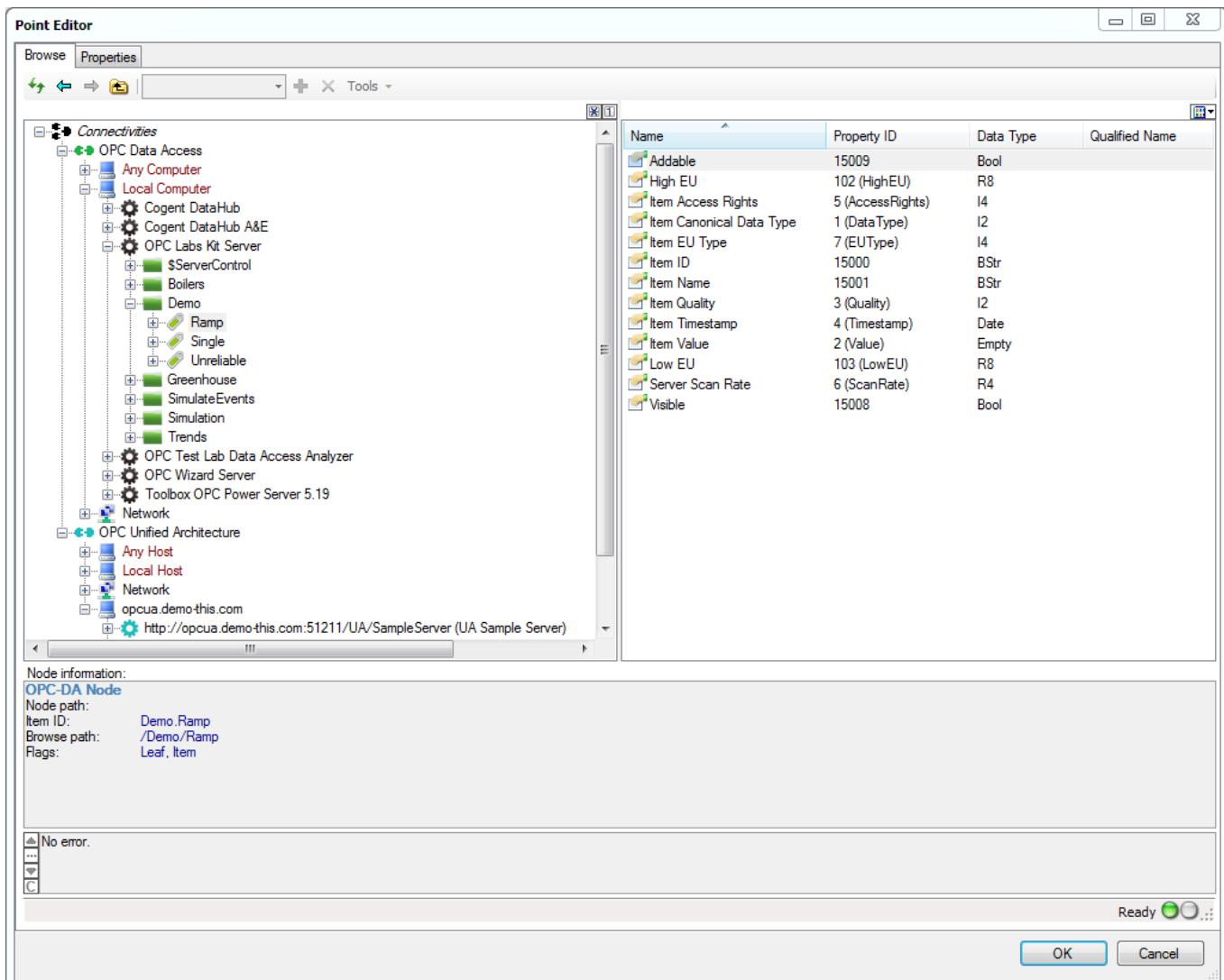
- [DALitemPoint](#): A point for OPC-DA items.
- [DAPropertyPoint](#): A point for OPC-DA properties (on OPC items).

For OPC Unified Architecture (with the [UAConnectivity](#) component), the following point type exists:

- [UAAttributePoint](#): A point for OPC-UA data (node attributes).

## 8.1.2.3 Point Editor

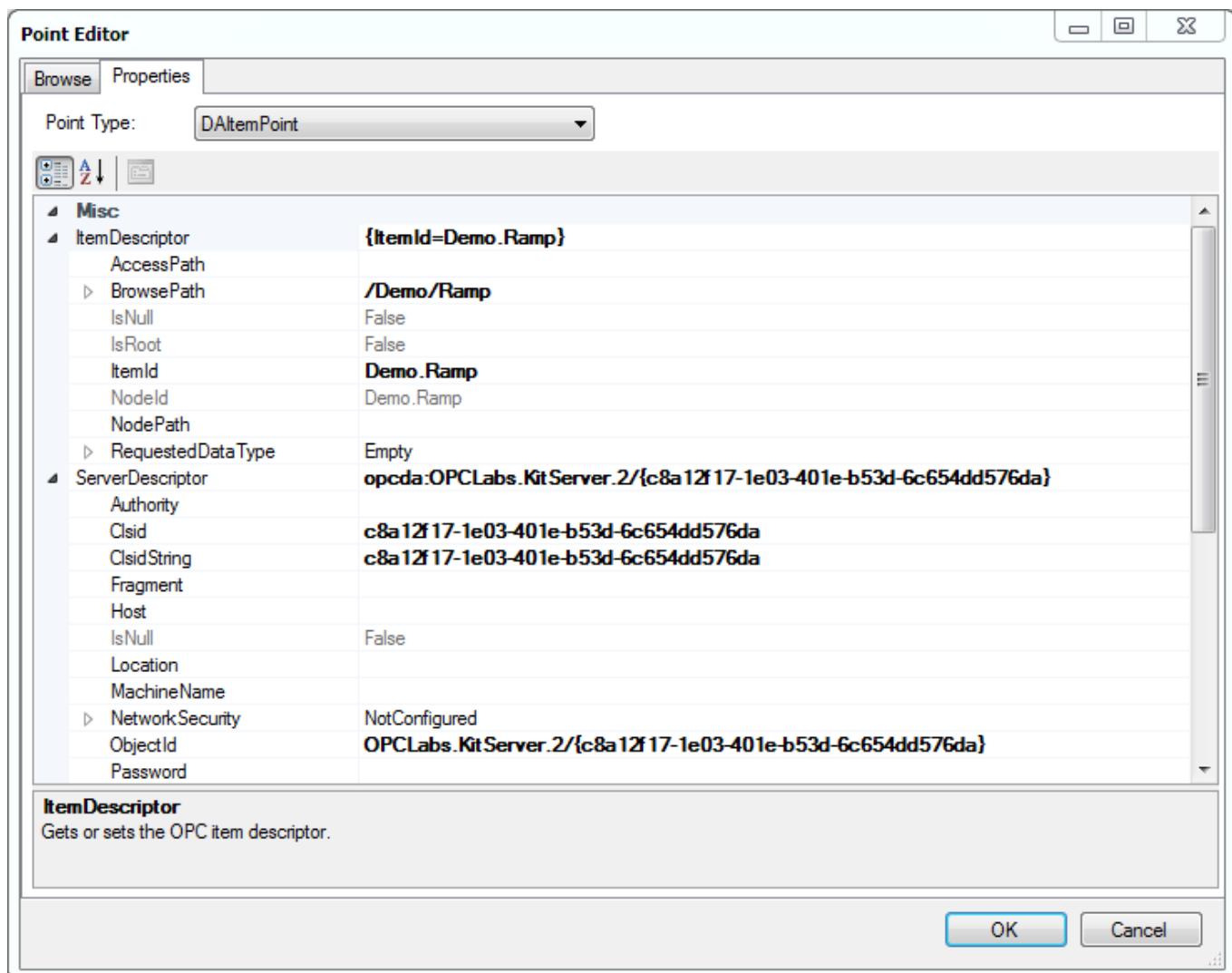
When you are working with the Live Binding in Visual Studio designer, you often need to specify the point you'd like to bind to. For this purpose, a Point Editor dialog is displayed. An example of it is on the following picture:



The actual contents of the dialog will of course differ, depending on your configuration. This particular example is with a composite connectivity configured with both OPC Data Access and OPC Unified Architecture, showing multiple binding possibilities in the same dialog.

The point editor has two tabs, labeled **Browse** and **Properties**. Initially, the **Browse** tab is selected, and you are given an opportunity to browse the network, the OPC servers present, and the nodes in them. You can simply select the desired node and press **OK**.

Sometimes the browsing is not possible, or you may need to view and possibly the details of the selected point. In such cases, you can switch to the **Properties** tab of the dialog:



In the Properties tab, the data that describe the point are displayed in a detailed property grid. In addition, in the upper part of the tab, you can select any type of point supported by the connectivity in use, and then fill in its properties as needed.

## 8.1.2.4 Parameters

While the point determines basically the location of the data, *parameters* determine how the data will be accessed. Some operations do not need parameters (in such cases they still exist, but are empty), but most do. For example, a subscription typically has some kind of update rate as its parameter.

The parameters may depend on the type of point, and the operation involved. A typical connectivity that supports reading, writing and subscription will therefore have up to three types of parameters for each type of point: Read parameters, write parameters, and subscription parameters.

The parameters are not part of the point itself, and are therefore not selected with it in the Point Editor. They are part of the binding.

## 8.1.2.5 Arguments and Results

The *arguments* are the actual data the binding operation transfers, i.e. the "what". Typically and most importantly,

arguments contain the data value being read or written or obtained through subscription.

As with the parameters, the usually depend on the type of point, and the operation involved.

Arguments are represented by an object that has properties, divided into several categories. The categories are as follows:

- **Input:** The data that is taken from the binding target and provided to the operation (for example, a value to be written).
- **Output:** The data that is obtained by the operation and stored into the binding target (for example, the value read).
- **Input/Output:** A combination of an input and an output argument.
- **Result:** The result of the operation (for example, the error code returned).

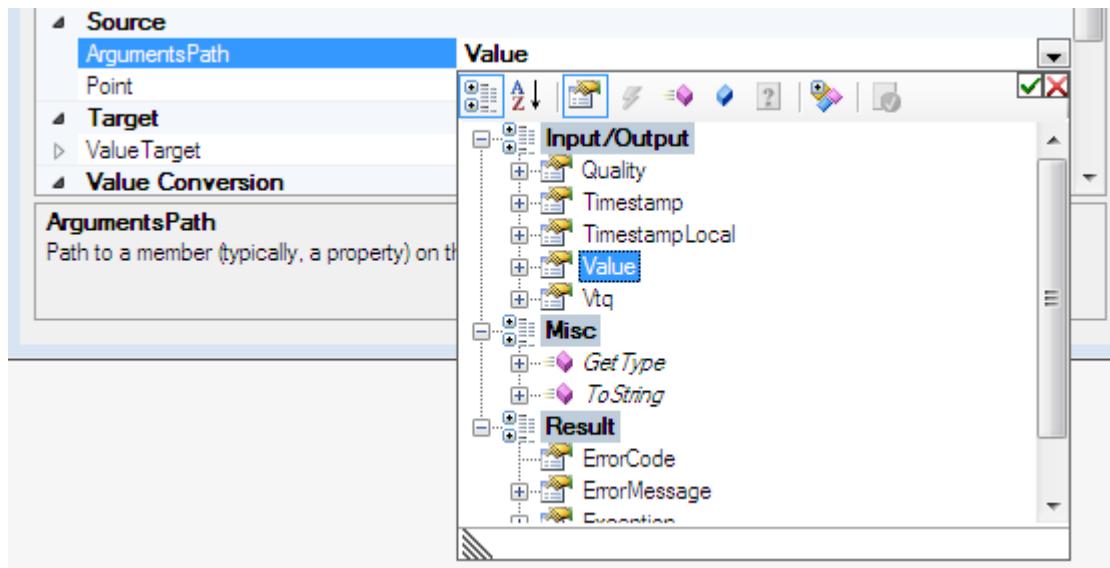
There is an important distinction between the *outputs*, and the *results*. Each operation can be either successful, or not. When the operation completes, the result arguments that are always available. The outputs, however, are only available when the operation succeeded.

The [PointBinding](#) has an [UpdateOutputOnFailure](#) property (defaults to '[false](#)'). Normally, when an operation (such as OPC read) fails, and you are binding to an operation output, nothing happens - the binding output is not updated. For example, a text bound to an OPC data value continues to display the last known value. When the [UpdateOutputOnFailure](#) property is set to '[true](#)', the binding output will always be updated. In our example, the text box will be cleared in case of operation failure.

## 8.1.2.6 Arguments Path

When making a binding, you are basically connecting a value target (typically, a property on a visual control) with an argument (described in the above chapter). The argument being used is specified using an *argument path*. An argument path can be simply a property name (name of the argument), such as [Value](#), or a sequence of member names (separated by dots) that determines the path into the argument; for example, [Quality.StatusBitField](#).

In the Visual Studio designer, the arguments path can either be typed in, or you can select using a drop-down editor, as on the following picture:



A binding can be involved in multiple binding operations (e.g. read and write), but there is just a single arguments path selection for the whole binding, and the set of arguments might be different for each operation. The arguments path should be valid for all the binding operations that are enabled on the binding. The drop-down editor offers a combined view of the possible arguments. For this reason, argument categories can change (by combining) as well: For example, in OPC DataAccess, a [Value](#) is an output in the [Read](#) operation, but it is an input in the [Write](#) operation. It

is therefore shown under the Input/Output category, when both [Read](#) and [Write](#) binding operations are enabled on the binding.

## 8.1.2.7 Parameter Templates

When you create new bindings, often they share many common characteristics. For example, you may be binding to OPC items all residing in the same OPC server, or using the same parameters, such as the update rate.

In order to make it easier to create such similar bindings, parameters of newly created bindings are taken from *parameter templates*.

For OPC Data Access (OPC-DA), you can configure following templates on the [DAConnectivity](#) component:

- [ItemReadParametersTemplate](#): OPC-DA read parameters, such as the data source (cache, device, and so on), in newly added bindings to OPC items.
- [ItemSubscribeParametersTemplate](#): OPC-DA group parameters in newly added bindings to OPC-DA items.
- [ItemWriteParametersTemplate](#): OPC-DA write parameters (such as whether just the value, or the whole VTQ triple should be written) in newly added bindings to OPC-DA items.

For OPC Unified Architecture (OPC-UA), you can configure following templates on the [UAConnectivity](#) component:

- [AttributeReadParametersTemplate](#): OPC-UA read parameters (such as the maximum value age) in newly added bindings to OPC-UA attributes.
- [AttributeSubscribeParametersTemplate](#): OPC-UA subscription and monitoring parameters (such as the sampling interval, or data change filter) in newly added bindings to OPC-UA attributes.
- [AttributeWriteParametersTemplate](#): OPC-UA write parameters (such as whether just the value should be written) in newly added bindings to OPC-UA attributes.

Templates have meaning only in the designer; they are not used in runtime.

## 8.1.2.8 Queued Execution

With the help of [QueuedExecution](#) property on the [PointBinder](#), the developer can choose that the operations (such as [Write](#)-s) are performed outside the calling thread, and therefore do not block. Typical usage involves a one-time [Read](#) or [Write](#) linked to a click on a button, which would otherwise block the user interface until completed.

The [Boolean QueuedExecution](#) property defaults to '[false](#)'. When set to '[true](#)', the operations on the binder are executed outside the calling thread, but still in the order of arrival. This allows non-blocking operation calls on the binder.

## 8.1.3 Value Conversions, String Formats and Converters

When values are transferred to or from the target, *value conversions* can be made – using *string formats*, or *data converters*.

In order to format the bound value according to a string format, simply set the [StringFormat](#) property of the binding. An empty string means no formatting. The format string is constructed according to usual .NET rules, and can contain following placeholders:

- {0}: The value.
- {1}: The type of the value (or null if the value itself is null)

The string formatting is only applied in one direction – for values being transferred from the binding source to the binding target.

In addition to formats defined by the .NET Framework, QuickOPC defines format strings for various its types, and especially the types used in Live Binding, such as [DAQuality](#), [DAVtq](#), [UAAttributeData](#), [UAStatusCode](#), etc. This makes it easy to format compound values for display purposes, without having to write any code. For details, please refer to Format String in the Reference.

When string formatting is not sufficient, any kind of conversion can be made using data converters, and possibly in both directions. Data converters are objects that implement the [IDataConverter](#) interface. You can place converters on the design surface of the form, and then select from them for the [Converter](#) property of any binding.

QuickOPC ships with several data converters, described in the subsequent chapters.

Besides using the data converter(s) that ship with the product, you can create your own data converter objects or components, and assign them to your bindings.

## 8.1.3.1 The LinearConverter Component

The [LinearConverter](#) component provide a linear conversion and its inverse, and also (optionally) allows output value clamping. The linear conversion is useful for proper scaling of values when the underlying OPC source does not directly have the desired value.

You configure the parameters of the linear conversion so that the [X1](#) value converts to [Y1](#), and the [X2](#) value converts to [Y2](#). Other values are interpolated or extrapolated according to these settings.

For values that fall outside the given limits, you can choose one of the following behaviors:

- [Normal](#). The off-limit condition is ignored, the value is converted as usual (i.e. extrapolated).
- [Error](#). An exception is thrown.
- [Clamp](#). The output value is clamped at the limit.

## 8.1.3.2 The StatusToColorConverter Component

This type of data converter is useful if you want to change some of the control's colors (e.g. background or foreground) according to the status of the bound source. For example, you may want the background of the text box turn to red color if there is an error related to the source.

The [StatusToColorConverter](#) can convert several types of values on its input. On its output, it provides a [Color](#) value that it chooses from one of the four colors that you can configure: [UnknownColor](#), [NormalColor](#), [WarningColor](#) and [ErrorColor](#). Each color corresponds to one state, and the possible state choices are given by the [StatusInfo](#) enumeration: [Unknown](#), [Normal](#), [Warning](#), and [Error](#).

The possible input values to the [StatusToColorConverter](#) are converted as follows:

- A null reference converts to [NormalColor](#).
- A [StatusInfo.Normal](#) converts to [NormalColor](#), a [StatusInfo.Warning](#) converts to [WarningColor](#), a [StatusInfo.Error](#) converts to [ErrorColor](#), and a [StatusInfo.Unknown](#) converts to [UnknownColor](#).
- An [Exception](#)-typed value converts to [ErrorColor](#).
- Empty [String](#)-s convert to [NormalColor](#), non-empty [String](#)-s convert to [ErrorColor](#).
- Zero integers convert to [NormalColor](#), positive integers convert to [WarningColor](#), negative integers convert to [ErrorColor](#).
- A true [Boolean](#) converts to [NormalColor](#), a false [Boolean](#) converts to [ErrorColor](#).

Several types in QuickOPC provide built-in conversion to [StatusInfo](#) enumeration, and can therefore be used as inputs to [StatusColorConverter](#) as well.

Following types support conversion to [StatusInfo](#) in OPC Data Access:

- [DAQuality](#)
- [DAVtqResult](#)

- [EasyDItemChangedEventArgs](#)

Following types support conversion to StatusInfo in OPC Unified Architecture:

- [EasyUADataChangeNotificationEventArgs](#)
- [UAAttributeDataResult](#)
- [UAStatusCode](#)

The rules and built-in conversion are design in such way that you can easily use various kinds of bindings with the [StatusToColorConverter](#) component. For example, if you set the [ArgumentsPath](#) property of the binding to [Exception](#), it will convert to one of two colors. Even better, if you set the [ArgumentsPath](#) of the binding to [StatusInfo](#), it will properly convert to one of all four possible colors.

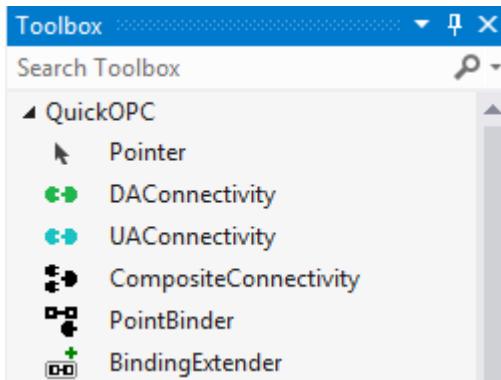
## 8.1.4 Live Binding in the Designer

The beauty of the live binding is that it can be set up without any manual coding. This means that there has to be a tight integration with the development environment (IDE), i.e. Visual Studio.

This chapter describes the various places where the live binding is integrated with the Visual Studio and its designer features.

### 8.1.4.1 Toolbox Items

The live binding components appear in Visual Studio's toolbox under the "QuickOPC" tab:



Note: For the components to be visible in the toolbox, your project needs to be set to target the .NET Framework 4.5.2 or later.

In order to work with live binding and enable it in your project, you need to drag the desired component from the toolbox to the design surface of your form (container).

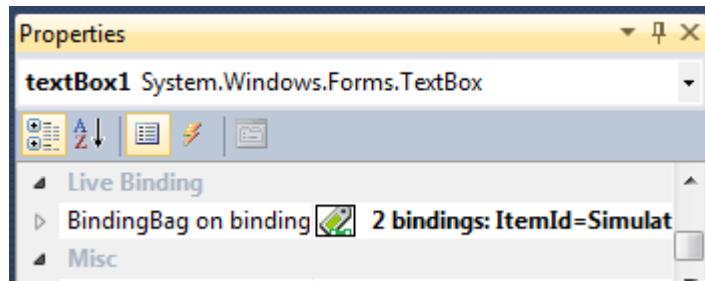
Because the [PointBinder](#) must always be used with [BindingExtender](#) in order to be usable, the toolbox item for the [PointBinder](#) uses a special logic, and assures that there is an associated [BindingExtender](#) in the container (usually, a form) as well. When you drag a [PointBinder](#) to the designer' surface, we first check whether some [BindingExtender](#) already exists in the container, and if so, we configure the newly created binder component to use it (by setting its [BindingExtender](#) property). If there is no [BindingExtender](#) component on the form, we create a new one, and configure the new binder to use it as well.

Similarly, any connectivity component is normally used with the [PointBinder](#). Dragging [DAConnectivity](#), [UAConnectivity](#) or [CompositeConnectivity](#) to the designer' surface takes with (or configures) a [PointBinder](#) as well.

As a result, in most cases you can simply "forget" about the need to place the [BindingExtender](#) or the [PointBinder](#) on the form – simply drag the connectivity ([DAConnectivity](#), [UAConnectivity](#) or [CompositeConnectivity](#)) from the toolbox to the designer's surface, and the rest will be taken care of. Just remember that you should not delete the [BindingExtender](#) or the [PointBinder](#) from your form, even though you have not explicitly placed it there.

## 8.1.4.2 Extender Properties

The binding extender ([BindingExtender](#) component) has the ability to extend the properties of existing controls. When a binding extender is in the container (is placed onto the form's design surface), each eligible control receives new properties. Some of these properties are hidden, the remaining appear under the "Live Binding" category in the Properties window for each extended control:



The [BindingBag](#) extender property contains the binding bag, i.e. the collection of live bindings on the control. The designer shows the number of bindings contained in the bag. If they have the same type, it also shows a corresponding icon for that binding type, and if they have the same source, it also shows what the source is.

## 8.1.4.3 Designer Commands

Placing a [BindingExtender](#) onto a designer's surface automatically extends the set of commands (so-called Designer Verbs) available for other components (controls) in the same container (usually, a Windows Form).

The new commands are available in the context menu for each component (when you right-click on the component or control), and also in the property grid for that component (in the Properties window), when the component is selected:

[Bind to Point...](#), [Edit Live Bindings...](#),  
[Remove Live Bindings](#)

If you do not see the commands in the Properties window, right-click in the property grid area, and make sure that the "Commands" menu item is checked.

Following designer commands are available:

### **Bind to Point:**

Allows you to select a point (OPC-DA item, property, OPC UA node/attribute) first, and then binds its value to the selected control, for the [Read](#) and [Subscribe](#) operations. It binds to the default target member of the control.

You can subsequently use the "Edit Live Bindings..." command if you want to modify the target member selection, or other characteristics of the binding.

### **Edit Live Bindings:**

Invokes the Binding Collection Editor, allowing you to add, remove, and modify bindings on the selected control.

### **Remove Live Bindings:**

Removes all live bindings currently defined on the selected control.

When repeatedly using any of the "Bind to..." commands, the browsing continues at the recently selected node. This allows for easier binding of multiple components to a set of related nodes.

When repeatedly using any of the "Bind to..." commands, the browsing dialog retains its location and size between invocations, making it easy for the user to position it conveniently during the design session.

### 8.1.4.4 Binding Collection Editor

The binding collection editor allows adding, removing, and modifying bindings. It is usually invoked to work on bindings for a specified control, but it can also be used to work on bindings of all controls in a container (a Windows form, typically). It is basically an editor for Binding Bags.

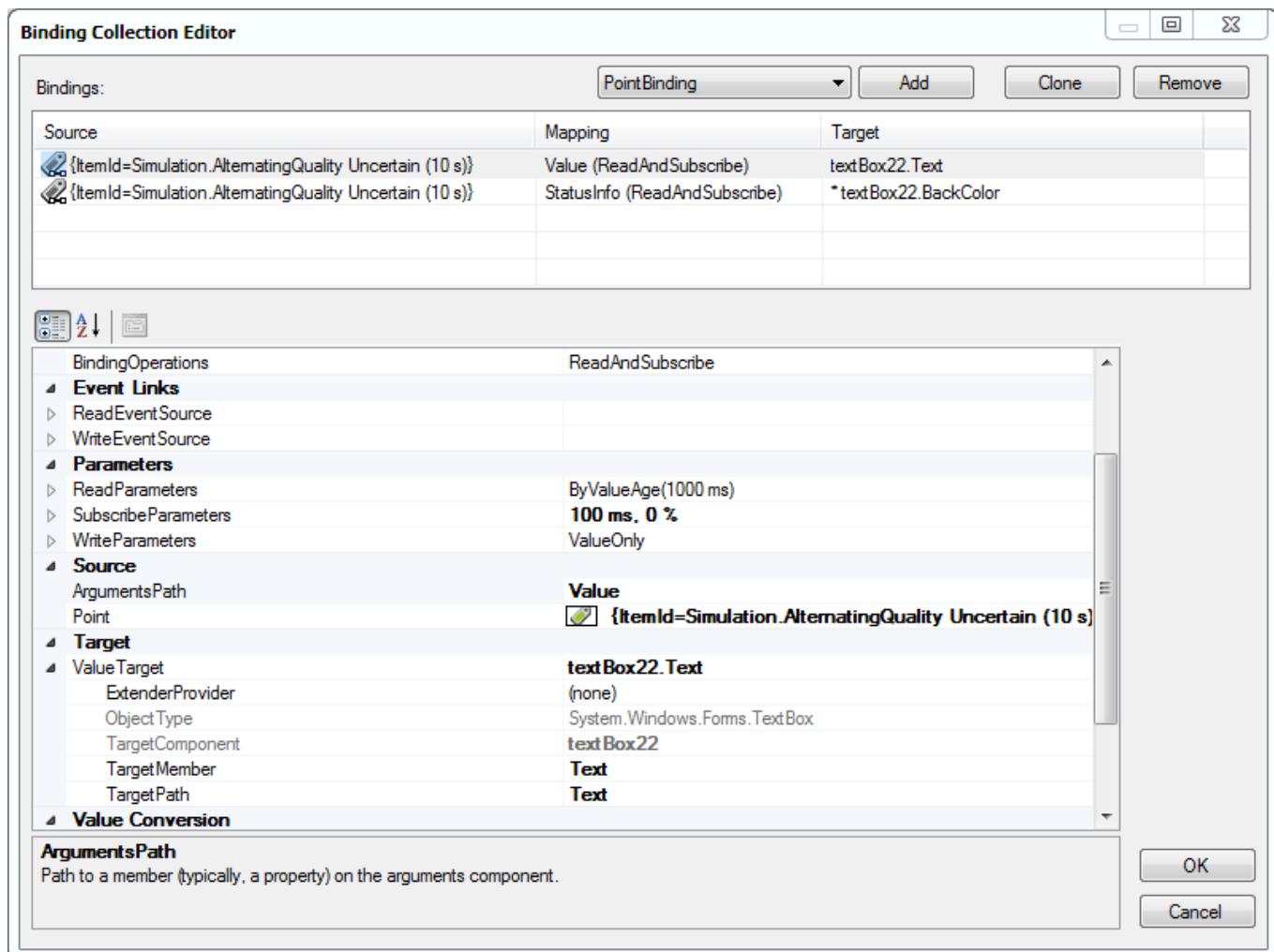
To show the binding collection editor for a specific control, you can either:

- Right-click on the control, and select "Edit Live Bindings..." from the context menu.
- Select the control, and then click on "Edit Live Bindings..." command in the Properties window (if commands are not visible in the property grid, right-click on it and check the Commands menu item).
- Select the control, and in the Properties window, find the BindingBag property (under the Live Binding category). Select this property, and then click the small "..." button on the right side.

Note: If you do not see any of the above commands, the likely reason is that you have not placed the [BindingExtender](#) component on the form.

To show the binding collection editor for all controls on a Windows form, select the [BindingExtender](#) component, and in the Properties window, find the [ComposedBindingBag](#) property. Select this property, and then click the small "..." button on the right side.

When invoked, the binding collection editor looks similar to this:



The "Bindings" list displays all bindings in the collection, and the most important information about them (the source, the binding operations, and the target of the binding). Selecting any row in the "Bindings" list displays the properties of the selected binding in the "Binding properties" property grid. You can then view and modify the detailed

properties of the binding in the property grid.

There are also following commands available in the binding collection editor:

**Add:**

Creates a new binding of the selected type, and adds it to the binding collection. Before pressing Add, you need to select the desired type of binding in the drop-down list left to the Add button.

The newly created binding gets some of its parameters from the templates defined on the binder or the connectivity component. If you create many similar bindings (such as always referring to the same OPC Server), you can save yourself lots of work by pre-filling the template with parameters that you commonly use.

**Clone:**

Create a duplicate of the selected binding (or bindings). This can be useful when you have made the main binding on the control, usually with [ArgumentsPath](#) of [Value](#), and you want to extend it further, e.g. by adding a tooltip, or change the background color. In this case you need to bind to the same source and using the same parameters, and you will be changing just the arguments path and the binding target.

**Remove:**

Remove the selected binding (or bindings).

**OK:**

Confirms the changes made to the bindings in the dialog, and closes the dialog.

**Cancel:**

Dismisses any changes made to the bindings in the dialog, and closes the dialog.

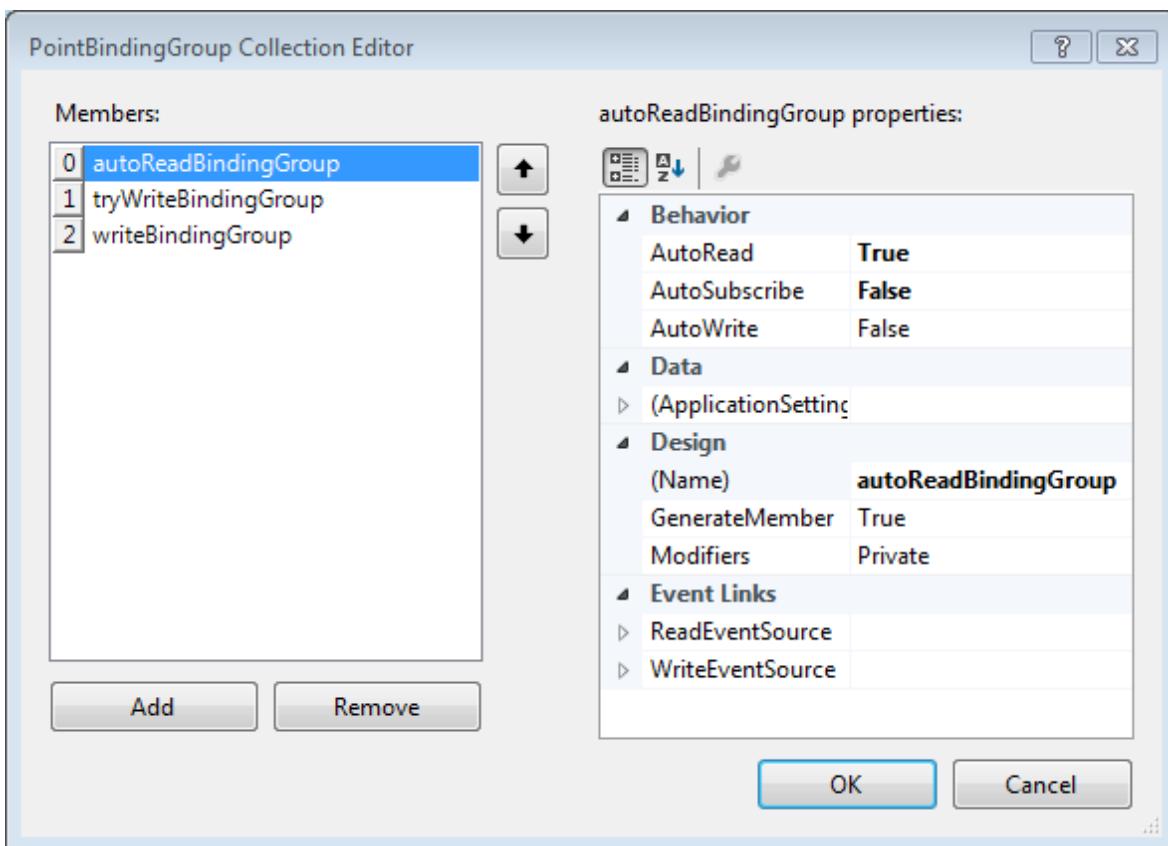
All changes that you make to the bindings in the binding collection editor are temporary, until you press OK. You can revert to the original state by pressing Cancel at any moment.

Note: Emptying the [ValueTarget.TargetComponent](#) of any binding will effectively remove the binding from the binding collection when the dialog changes are confirmed. This is because bindings must always be associated with their target.

## 8.1.4.5 Binding Group Collection Editor

The binding group collection editor allows adding, removing, and modifying the Binding Groups. To show the binding group editor, select the [PointBinder](#) component placed on your form, and in the Properties window, select its [BindingGroups](#) property. Then, click the small “...” button on the right side.

When invoked, the binding group collection editor looks similar to this:



Note that the default binding group does not appear in the [BindingGroups](#) collection. It is a separate object that you can edit directly, as the [DefaultBindingGroup](#) property of the [PointBinder](#) component.

## 8.1.4.6 Online Design

The live binding model includes an Online Design feature that allows you to explore and verify most aspects of the configured live bindings without building and running your application. You can see the binding truly live – i.e. while still designing in Visual Studio, the binding operations can be performed. This means that when you configure e.g. a [Read](#) or [Subscribe](#) operation with your binding, you can pre-view live OPC data on your form, and verify that the binding works as intended.

Initially, the Online Design mode is turned off. In order control the Online Design mode, you can either right-click on the [BindingExtender](#) component on your form, and select “Design Online” or “Design Offline” command, or you can select the [BindingExtender](#) component first, and then use the same commands in Properties windows (provided that commands are enabled in the property grid; if not, right-click on it and check the Commands item).

[Design Online](#), [Design Offline](#)

Because the live binding makes modifications to property values of components in your program, be aware that the online design has the same effect. If you do not want the values resulting from online design to persist in your project, you need to refrain from saving the changes made by online design.

When the design is made online, it can be much more demanding on computer resources, because each value update causes value changes that need to be reflected by the designer, regenerating code and making other changes in the environment. In order to maintain responsiveness of the development environment, the update rate in the online design mode is limited. Even if you set the update rate to a fast value, you may get slower update rates during the design.

## 8.1.5 Live Binding Details

This chapter describes more advanced workings of the live binding, and some internals that help in understanding it.

### 8.1.5.1 How the Binding Extender Automatically Goes Online and Offline

The BindingExtender component has two configurable event links that allow it to automatically go online or offline:

- [OnlineEventSource](#): Event triggered by this source puts the binding extender into the online mode.
- [OfflineEventSource](#): Event triggered by this source puts the binding extender into the offline mode.

When you initially place the BindingExtender component onto a Windows form by dragging it from a Toolbox, its [OnlineEventSource](#) is set to the [Form's Shown](#) event, and its [OfflineEventSource](#) is set to the [Form's FormClosing](#) event. The effect of this configuration is such that during runtime, the binding extender is automatically online while the form is open, and it also goes offline when the form closes.

This default setting makes it easy to create forms that are "live" (online) without further coding or configuration. If you, however, want to have control over the process, you can clear these event sources, and use the [BindingExtender's Online](#) property to set or reset the online state.

### 8.1.5.2 What Happens When the Binding Extender Is Set Online or Offline

When the binding extender is set to online (i.e. when its [Online](#) property is set to '`true`'), following steps are performed internally:

1. Each configured binding makes its "links", i.e. hooks the handlers for its configured event sources. For example, the [PointBinding](#)-s have [ReadEventSource](#) and [WriteEventSource](#), so if they are configured, the binding will link the events with handlers that perform [Read](#) or [Write](#) operation on the point binding.
2. All binding groups perform their configured automatic operations. For [PointBindingGroup](#)-s, this means that:
  - a. If [AutoRead](#) property is set, the [Read](#) operation is executed for all [PointBinding](#)-s that have [Read](#) included in their [BindingOperations](#) property (this includes "Get" of OPC-DA properties).
  - b. If [AutoSubscribe](#) property is set, the [Subscribe\(true\)](#) operation is executed for all [PointBinding](#)-s that have [Subscribe](#) included in their [BindingOperations](#) property.
3. All binding groups make their "links", i.e. hook the handlers for their configured event sources. For [PointBindingGroup](#)-s, this means that the events for [ReadEventSource](#) and [WriteEventSource](#) will be linked with handlers that perform [Read](#) or [Write](#) operation on the binding group.
4. The [OnlineChanged](#) event is raised on the [BindingExtender](#).

When setting the binding extender to offline, the above steps are (roughly said) reversed. There are, however, differences in Step 2: For [PointBindingGroups](#)-s, the [Read](#) operations are not executed; instead, if [AutoWrite](#) property is set, the [Write](#) operation is executed for all [PointBinding](#)-s that have [Write](#) included in their [BindingOperations](#) property. Also, if [AutoSubscribe](#) property is set, the [Subscribe\(false\)](#) operation is executed (i.e. unsubscribe, instead of subscribe).

### 8.1.5.3 What Happens When a Binding Operation Method Is Called

When any operation is invoked either automatically (with help of configures event sources), or programmatically, the live binding system first determines the set of bindings involved in the operation, and then executes the requested operation.

For operations invoked directly on the binding, the operation is performed on this single binding.

For operations invoked on some binding group, the group first determines the bindings that belong to it. It then performs the operation together on all bindings in this group.

## 8.1.5.4 Programmatic Access to Live Binding

It is expected that in most cases, you will use the live binding by configuring it from the designer, without manually writing code. There are situations, however, where some additional custom programming may be needed. The live binding exposes object model that allows it be controlled programmatically; the most important parts of that model are described in the Reference documentation.

Below are the most commonly needed programmatic actions:

- To set the binding extender (all bindings on it) to online or offline from your code, set its [Online](#) property to true or false. The binding extender is a component placed on your form, so you can refer to it by its name.
- To execute the [Read](#) or [Write](#) operation on concrete binding, call the [PointBinding](#)'s [ExecuteRead](#) or [ExecuteWrite](#) method. Note that although the individual binding do not appear on the design surface (of the form), they are still components of your form, and you can refer to individual bindings by their name (as of now, you cannot change the name from the binding collection editor, so you need to look it up in the designer-generated code, and either use it as is, or rename manually).
- To execute any of the operations on a concrete binding group (for all binding in that group that have that operation allowed), call a corresponding method on the [PointBindingGroup](#) object. The binding groups do not appear on the design surface (of the form), but they are still components of your form, and you can refer to them by their name. The name of the binding group can be viewed or modified in the Binding Group Collection Editor. The method names are [PointBindingGroup.ExecuteRead](#), [ExecuteSubscribe](#) and [ExecuteWrite](#). The [ExecuteSubscribe](#) has a [bool](#) argument that determines whether you want to subscribe or unsubscribe.

You can also override some of the error handling in live binding (see Error Handling in Live Binding).

## 8.1.5.5 Error Handling in Live Binding

There are several types of errors that can occur in relation to live binding, and they are treated differently.

- I. When you configure the live binding in the designer, there may be one or more *binding errors*, caused by improper configuration. When this happens, the designer displays a form that lists all the binding errors detected.

You can press the "Details..." button to view the details of the binding errors listed on the form.

The binding errors are caused by [BindingException](#)-s during the design, and are generally "benign", meaning that you can continue your design work, but you'd better fix the errors, otherwise they can cause more serious errors during runtime.

Examples of such binding errors are:

- A binding where the value target has no [TargetComponent](#) specified.
- Member linking error: The specified target member does not exist.
- Member linking error: Cannot determine the data type of the target member.

In the runtime, binding errors are reported by the [BindingExtender.BindingErrors](#) event. Unless you hook your own handler to this event, they are silently ignored.

- II. When the binding extender is set online or offline, there can also be *event linking errors*. They are similar to binding errors in that they are caused by improper configuration – this time, an improper configuration of some of the event sources used to automatically trigger the operations.

Unless you decide to handle them, event linking errors are silently ignored both during design-time and in runtime. For event sources defined on the binding extender, any event linking error is signaled by the [BindingExtender.EventLinkingFailure](#) event. For event sources defined on the binder component ([PointBinder](#), usually), its binding groups or its bindings, any event linking error is signaled by the [BinderBase.EventLinkingFailure](#) event.

- III. When a target could not be updated with a new value, an *update failure* occurs. Unless you decide to handle them, update failures are silently ignored both during design-time and in runtime. Each failure is signaled by the [PointBinder.UpdateFailure](#) event. The event notification contains an [Exception](#) indicating the cause of the update failure, and a reference to the related binding. You can hook your event handler to this event if you need some action be performed in such case.

There may be various reasons for update failures; the most common are *target access errors*. They happen when the live binding encounters a failure accessing your objects (binding targets). For example, setting or getting a property on your object may throw an exception, and such exception will be caught. Target access error is represented by an instance of [TargetAccessException](#) in an update failure.

Errors that originate in the binding source, such as OPC item or OPC property, are considered a completely separate issue. For example, the OPC server may be shut down, or the item ID might have changed name and is no longer accessible, or its value just has a bad “quality”. Such errors are treated as just another data to be transferred by the binding. There are arguments (see Arguments and Results) such as [Exception](#) or [ErrorCode](#) that allow you to set up members on your objects that will receive error information in case of errors that originate in the binding source.

## 8.1.5.6 Usage Guidelines

Below are some pieces of advice that may help you configure and use the live binding properly:

- If operations need to be performed together, they must be assigned to the same binding group and invoked on the group. Let's describe the wrong approach first: When the user presses a button, you want to grab values from several controls on the form, and write them to various OPC items. You configure the [WriteEventSource](#) on each binding to the [Click](#) event of the [Button](#). Well, while this “kind of” works, it is very ineffective approach, because each binding will be handled separately, and the OPC writes will be executed one by one, sequentially. The proper approach is to create a binding group for this purpose, set all bindings involved in the “write” to be members of this binding group, and then set the [WriteEventSource](#) just once, on the binding group itself. This way, the binding group will attempt to write all values into their corresponding OPC items using a single call with multiple items in it.
- Remember to always set the BindingExtender back to offline mode. This guideline only applies if you are controlling the [BindingExtender](#)-s [Online](#) property in a way that differs from the default described in How the Binding Extender Automatically Goes Online and Offline. In such case, you need to be aware that while the [BindingExtender](#) is online mode, there are event handlers set up that may prevent the normal application shutdown. Basically, by setting the [Online](#) property of [BindingExtender](#) to 'true', you indicate that for whatever reason, you are interested in being connected to the binding targets – and you have to explicitly set it back to false in order to indicate that this is no longer needed.

## 8.1.6 Typical Binding Scenarios

The live binding model is very powerful in terms of what can be achieved without any manual coding. In order to configure it right, however, you need to know how, and understand a little about the internal workings of live binding.

The scenarios described in this chapter are designed just for that – to give instructions that will guide you through configuring the live binding for typical usages.

Each scenario contains a description of its purpose, and step by step instructions explaining how to configure it.

You should be able to use these scenarios both as a “catalog”, a reference to choose from and follow the steps, and as a learning tool: If you try several scenarios, you will get a feeling of how to configure the live binding for proper behavior.

Many of the scenarios listed here are also demonstrated live in the “Live Binding Demo” applications that are installed with the product. For OPC Data Access (OPC-DA), its source code is contained in the WindowsForms\LiveBindingDemo project. For OPC Unified Architecture (OPC-UA), its source code is contained in the WindowsForms\UALiveBindingDemo project.

## 8.1.6.1 Automatic Subscription (With the Form)

This is the default behavior. The binding is online while the form is open. You can bind to all types of components and controls. You can bind to any property or field. You can also bind to some methods and events.

*How to configure this feature for OPC-DA:*

If you have not done so yet, drag the DAConnectivity component from the Toolbox to the form (this is a common step for all binding tasks).

Use the “Bind to Point” command on the target control, and select the source OPC server and OPC item.

That's it. You can verify the result immediately, without recompiling your application, by using the “Design Online” command on the [BindingExtender](#) component.

*How to configure this feature for OPC-UA:*

If you have not done so yet, drag the UAConnectivity component from the Toolbox to the form (this is a common step for all binding tasks).

Use the “Bind to Point” command on the target control, and select the source OPC-UA server and OPC node.

That's it. You can verify the result immediately, without recompiling your application, by using the “Design Online” command on the [BindingExtender](#) component.

## 8.1.6.2 Automatic Read (On Form Load)

OPC-DA item values (or OPC-UA node attributes) can be also read once, when the form loads.

*How to configure this feature for OPC-DA:*

On a [PointBinder](#), edit the [BindingGroups](#) collection, “Add” a new binding group, set its [AutoRead](#) property to True, and make sure that its [AutoSubscribe](#) property is set to False.

On a [TextBox](#) (or other control), use “Bind to Point” command and select the OPC item you want to read. Then, use the “Edit Live Bindings” command, and set the [BindingGroup](#) property of the item binding to the binding group you have created earlier. The [Read](#) operation is automatically executed for all item bindings in this group when the form loads.

*How to configure this feature for OPC-UA:*

On a [PointBinder](#), edit the [BindingGroups](#) collection, “Add” a new binding group, set its [AutoRead](#) property to True, and make sure that its [AutoSubscribe](#) property is set to False.

On a [TextBox](#), use “Bind to Point” command and select the OPC-UA node you want to read. Then, use the “Edit Live Bindings” command, and set the [BindingGroup](#) property of the data binding to the binding group you have created earlier. The [Read](#) operation is automatically executed for all data bindings in this group when the form loads.

## 8.1.6.3 Read On Custom Event

For example, clicking the Read button will execute the operation and read the OPC item value into the text box.

*How to configure this feature for OPC-DA:*

On a [TextBox](#) (or other control), use "Bind to Point" command and select the OPC item you want to read (alternatively, use "Edit Live Bindings" command, "Add" a [PointBinding](#), and configure its point's [ServerDescriptor](#) and [ItemDescriptor](#)).

Then, use the "Edit Live Bindings" command, and for [BindingOperations](#) property, leave just [Read](#) checked (and uncheck the [Subscribe](#)). In order to have the [Read](#) invoked when the button is pressed, set the [ReadEventSource.SourceComponent](#) to the [Button](#) control; the [SourceMember](#) of the [ReadEventSource](#) will be automatically set to the [Click](#) event. This causes the [Click](#) on the [Button](#) execute the [Read](#) operation, i.e. obtain the OPC item value and store it into the [TextBox](#).

Note: In order to read multiple items at once on a custom event, it is suggested that you set up a binding group under the [PointBinder](#), assign the bindings to the group, and link the [ReadEventSource](#) just once, on the binding group.

*How to configure this feature for OPC-UA:*

On a [TextBox](#), use "Bind to Point" command and select the OPC-UA node you want to read (alternatively, use "Edit Live Bindings" command, "Add" a [PointBinding](#), and configure its [EndpointDescriptor](#) and [NodeDescriptor](#)).

Then, use the "Edit Live Bindings" command, and for [BindingOperations](#) property, leave just [Read](#) checked (and uncheck the [Subscribe](#)). In order to have the [Read](#) invoked when the button is pressed, set the [ReadEventSource.SourceComponent](#) to the [Button](#) control; the [SourceMember](#) of the [ReadEventSource](#) will be automatically set to the [Click](#) event. This causes the [Click](#) on the [Button](#) execute the [Read](#) operation, i.e. obtain the OPC-UA node value and store it into the [TextBox](#).

Note: In order to read multiple nodes at once on a custom event, it is suggested that you set up a binding group under the [PointBinder](#), assign the bindings to the group, and link the [ReadEventSource](#) just once, on the binding group.

## 8.1.6.4 String Formatting

All standard .NET formatting features can be used to specify how the OPC values should be displayed.

*How to configure this feature:*

Bind the OPC-DA item (or OPC-UA node attribute) to the [TextBox](#) or similar control as usually.

Then, use the "Edit Live Bindings" command on the control, and set the [StringFormat](#) property on the item binding as needed, using all standard .NET formatting capabilities. {0} in the formatting string represents the value to be formatted. For example, {0:F3} means a floating point value formatted with 3 decimal places.

## 8.1.6.5 Change Color According to Status

Values can be converted in both directions by [IDataConverter](#) objects. For example, [StatusToColorConverter](#) provides a color value based on operation status, such as a yellow background when quality is Uncertain.

*How to configure this feature:*

Drag the [StatusToColorConverter](#) component from the toolbox to the form. Bind the OPC-DA item (or OPC-UA node attribute) to your control as usually.

Then, use the "Edit Live Bindings" command, "Clone" the existing binding, and change the cloned binding as follows: Set the [ArgumentsPath](#) to [StatusInfo](#). Set the [ValueTarget.TargetMember](#) to the [BackColor](#) (or other suitable) property of your control. For the [Converter](#) property, select the [StatusToColorConverter](#) you have placed to the form earlier.

This will cause the status information from the OPC item be converted to one of configurable colors, and change the background color of the control appropriately.

## 8.1.6.6 ToolTip and Other Extenders

You can bind to properties made available by extender providers, such as [ToolTip](#). Hovering over the control associated with the [ToolTip](#) then allows the user to see more details about the value.

*How to configure this feature for OPC-DA:*

Drag the [ToolTip](#) extender (standard Microsoft component), or other extender provider you need, from the toolbox to the form. Configure the binding in a usual way.

Then, use the "Edit Live Bindings" command, "Clone" the existing binding, and change the cloned binding as follows: Set the [ValueTarget.ExtenderProvider](#) to the [ToolTip](#) (extender provider) component on your form, and set the [ValueTarget.TargetMember](#) to the [Tooltip](#) member. Set the [ArgumentsPath](#) to [Vtq](#) (or [Timestamp](#), [Quality](#), or whatever you'll want to be displayed in the tooltip). This will cause the additional information from the OPC item be displayed by the [ToolTip](#) extender provider for the control.

*How to configure this feature for OPC-UA:*

Drag the [ToolTip](#) extender (standard Microsoft component), or other extender provider you need, from the toolbox to the form. Configure the binding in a usual way.

Then, use the "Edit Live Bindings" command, "Clone" the existing binding, and change the cloned binding as follows: Set the [ValueTarget.ExtenderProvider](#) to the [ToolTip](#) (extender provider) component on your form, and set the [ValueTarget.TargetMember](#) to the [Tooltip](#) member. Set the [ArgumentsPath](#) to [AttributeData](#) (or [SourceTimestamp](#), [ServerTimestamp](#), [StatusCode](#), or whatever you'll want to be displayed in the tooltip). This will cause the additional information from the OPC-UA be displayed by the [ToolTip](#) extender provider for the control.

## 8.1.6.7 Display Errors with ErrorProvider

The standard [ErrorProvider](#) extender is useful for displaying OPC operation error icons, with tooltips.

*How to configure this feature:*

Drag the [ErrorProvider](#) extender (standard Microsoft component) from the toolbox to the form. Configure the binding in a usual way.

Then, use the "Edit Live Bindings" command, "Clone" the existing binding, and change the cloned binding as follows: Set the [ArgumentsPath](#) to [ErrorMessage](#). Set the [ValueTarget.ExtenderProvider](#) to the [ErrorProvider](#) component on your form, and set the [ValueTarget.TargetMember](#) to the [Error](#) member. All in all, this will cause the error message from the OPC-DA item (or OPC-UA node attribute) be stored into the error text displayed by the [ErrorProvider](#) for the control.

Note: If you configure this with explicit [Read](#) or [Write](#) operations (not with [Subscribe](#)), you need to set up a binding group, assign related bindings to the same group, and have the operation invoked on a binding group.

## 8.1.6.8 Various Kinds of Binding

You normally bind to the OPC value itself, but you can select from many other binding kinds, such as [ErrorCode](#), [ErrorMessage](#), [StatusInfo](#), (for OPC-DA) [Timestamp](#), [Quality](#), [Vtq](#), or (for OPC-UA) [ServerTimestamp](#), [SourceTimestamp](#), [AttributeData](#), [StatusCode](#), etc.

*How to configure this feature:*

Bind the OPC item to your control as usually.

Then, use the "Edit Live Bindings" command on the control, and in the [ArgumentsPath](#) property, select the kind of

binding you want.

## 8.1.6.9 Write Single Value on Custom Event

A value of control property is written to the associated OPC item when a specified event is triggered. For example, you can link the [Write](#) to the button click.

*How to configure this feature:*

On a [TextBox](#) (or similar control), use "Bind to Point" command and select the OPC-DA item (or OPC-UA node) you want to write to.

Then, use the "Edit Live Bindings" command, and for the [BindingOperations](#) options, select only "[Write](#)". In order to have the [Write](#) invoked when the button is pressed, set the [WriteEventSource.SourceComponent](#) to the [Button](#) control; the [SourceMember](#) of the [WriteEventSource](#) will be automatically set to the [Click](#) event. This causes the [Click](#) on the [Button](#) execute the [Write](#) operation, i.e. obtain the value from the control and write it into the OPC item.

## 8.1.6.10 Write Group of Values on Custom Event

In this scenario, multiple property values are gathered and written to their associated OPC items in a single operation, linked to a specified event – for example, a button click.

*How to configure this feature for OPC-DA:*

On a [PointBinder](#), edit the [BindingGroups](#) collection, "Add" a new binding group, set its [AutoSubscribe](#) property to False, and set its [WriteEventSource.SourceComponent](#) to the button that should invoke the [Write](#).

On each of the value controls involved, use "Bind to Point" command and select the OPC item you want to write. Then, use the "Edit Live Bindings" command, and for the [BindingOperations](#) options, select only "[Write](#)". Also, set the [BindingGroup](#) property of the item binding to the binding group you have created earlier. The [Write](#) operation will be executed for all item bindings in this group when the button is clicked.

*How to configure this feature for OPC-UA:*

On a [PointBinder](#), edit the [BindingGroups](#) collection, "Add" a new binding group, set its [AutoSubscribe](#) property to False, and set its [WriteEventSource.SourceComponent](#) to the button that should invoke the [Write](#).

On each of the value controls involved, use "Bind to Point" command and select the OPC-UA node you want to write. Then, use the "Edit Live Bindings" command, and for the [BindingOperations](#) options, select only "[Write](#)". Also, set the [BindingGroup](#) property of the data binding to the binding group you have created earlier. The [Write](#) operation will be executed for all data bindings in this group when the button is clicked.

## 8.1.6.11 Subscribe & Write

A control can reflect OPC value changes, and also write the updated values to the OPC server when modified by the user. For example, you can bind a [TrackBar](#) and a [NumericUpDown](#) controls to the same OPC item, and they will influence each other through OPC.

*How to configure this feature:*

On the target control, use "Bind to Point" command and select the OPC-DA item (or OPC-UA node) you want to bind to.

Then, use the "Edit Live Bindings" command, and for [BindingOperations](#) property, add a check next to the [Write](#) operation. In order to have the [Write](#) invoked when the control changes, set the [WriteEventSource.SourceComponent](#) to your control; the [SourceMember](#) of the [WriteEventSource](#) will be automatically determined, but you can change it, if you want to link to a different event.

Note: In OPC-DA, some controls will require you to set the [ItemDescriptor.RequestedDataType](#) so that the OPC value

type matches the type of the target property.

Note: In OPC-UA, some controls or OPC-UA servers will require you to set the [ValueTypeCode](#) (under the [WriteParameters](#) property) so that the OPC-UA value type matches the type of the target property. It is also recommended to set the [ValueTypeCode](#) for performance reasons.

## 8.1.6.12 Display Write Errors

When the [Write](#) fails, you want the user be notified. One way to do this is with the standard [ErrorProvider](#) control. For example, if you attempt to write a value that is too large, an error icon appears, and the error message can be seen in the associated tooltip.

*How to configure this feature:*

On a [PointBinder](#), edit the [BindingGroups](#) collection, "Add" a new binding group, set its [AutoSubscribe](#) property to False, and set its [WriteEventSource.SourceComponent](#) to the button that should invoke the [Write](#).

Drag the [ErrorProvider](#) extender (standard Microsoft component) from the toolbox to the form.

Configure the binding in a usual way. Use the "Edit Live Bindings" command, and for the [BindingOperations](#) options, select only "Write", and set the [BindingGroup](#) property of the item binding to the binding group you have created earlier. Then, "Clone" the existing binding, and change the cloned binding as follows: Set the [ArgumentsPath](#) to [ErrorMessage](#). Set the [ValueTarget.ExtenderProvider](#) to the [ErrorProvider](#) component on your form, and set the [ValueTarget.TargetMember](#) to the [Error](#) member. This will cause the error message from the OPC item be stored into the error text displayed by the [ErrorProvider](#) for the control.

## 8.1.6.13 Get an OPC Property on Custom Event

You can bind to OPC-DA properties as well. For example, clicking on the Get button can obtain the value of the [HighEU](#) property.

Note: In OPC-UA Unified Architecture, there is no need for this – just use [Read](#) of any attribute of a node.

*How to configure this feature:*

On a [TextBox](#), use "Bind to Point" command and select the OPC item and property you want to get (alternatively, use "Edit Live Bindings" command, "Add" a [PointBinding](#), and configure its [ServerDescriptor](#) and [PropertyDescriptor](#)).

In order to have the [Get](#) invoked when the button is pressed, set the [ReadEventSource.SourceComponent](#) to the [Button](#) control; the [SourceMember](#) of the [ReadEventSource](#) will be automatically set to the [Click](#) event. This causes the [Click](#) on the [Button](#) execute the [Get](#) operation, i.e. obtain the OPC property value and store it into the [TextBox](#).

## 8.1.6.14 Automatically Get an OPC Property (On Form Load)

The OPC-DA property value can also be obtained automatically when the form loads.

Note: In OPC-UA Unified Architecture, there is no need for this – just use [Read](#).

*How to configure this feature:*

On a [PointBinder](#), edit the [BindingGroups](#) collection, "Add" a new binding group, and set its [AutoRead](#) property to True.

On a [TextBox](#), use "Bind to Point" command and select the OPC item and property you want to get. Then, use the "Edit Live Bindings" command, and set the [BindingGroup](#) property of the property binding to the binding group you have created earlier. The [Get](#) operation is automatically executed for all property bindings in this group when the form loads.

## 8.1.7 Obsolete Live Binding Features

QuickOPC versions before version 5.35 used two separate types of binders ([DABinder](#), [UABinder](#)), and have not used the connectivity components ([DACConnectivity](#), [UAConnectivity](#), [CompositeConnectivity](#)) or a [PointBinder](#). The old binder types had used parts of the Live Mapping functionality. The [DABinder](#) and [UABinder](#) binder types are now obsolete.

This QuickOPC version still supports both the old ([DABinder](#), [UABinder](#)) and new ([PointBinder](#) with [DACConnectivity](#) or [UAConnectivity](#)) Live Binding approaches. The [DABinder](#), [UABinder](#) and their corresponding binding types are considered obsoleted, and may become unsupported or be removed in future versions of QuickOPC.

### 8.1.7.1 Documentation for Obsoleted Features

The obsoleted live binding features are no longer documented in the conceptual documentation (the reference documentation, however, is still provided for them). For a conceptual documentation to the obsoleted features, please used the documentation that came with the QuickOPC version you used originally.

### 8.1.7.2 Old Project Conversion

It is strongly recommended that you convert your projects to the new Live Binding approach. We have provided tools and instructions to facilitate the conversion.

In order to convert your project that uses the Live Binding to the new approach (assuming that you have QuickOPC 5.35 or later installed):

1. Make a backup of your project.
2. In Visual Studio, perform the steps below for each of your forms (or custom controls or other components) that use Live Binding.
3. Open the form.
4. If the form uses a [DABinder](#), drag [DACConnectivity](#) from the Toolbox to the designer surface. If the form uses [UABinder](#), drag [UAConnectivity](#) from the toolbox to the designer surface. Note that a [PointBinder](#) component will be added automatically as well.
5. Select (click on) the existing [DABinder](#) or [UABinder](#) component in the component tray (below the form). In the Properties window (for the [DABinder](#) or [UABinder](#) component you have selected), invoke the "Convert to point bindings" command. The commands are near the bottom of the property grid. If no commands are displayed, right-click in the left column of the property grid, and enable them from the context menu first.
6. A message box will be displayed, describing the outcome of the conversion. Confirm it by pressing OK. The bindings are binding groups are now converted. Note: Binding groups get new member names.
7. Right-click on the existing [DABinder](#) or [UABinder](#) component in the component tray (below the form), and delete it (the Delete command).
8. Save the form, build the project, and test it.

## 9 Reactive Programming Model

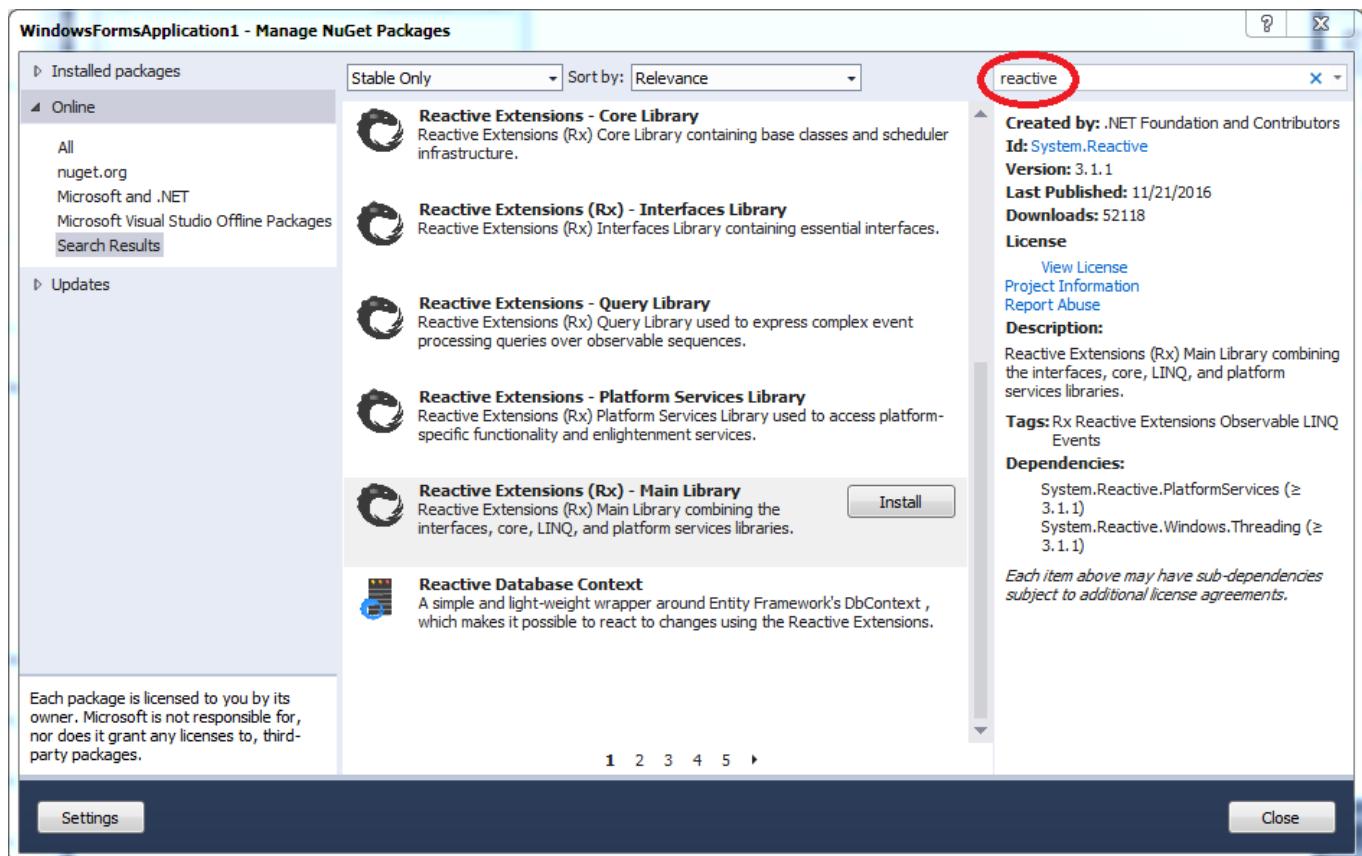


### 9.1 Reactive Programming Model for OPC Data (Classic and UA)

Microsoft Reactive Extensions (Rx) and reactive programming in general is a big subject and cannot be covered in any detail here. We will only describe the specifics that QuickOPC brings to Rx, by bridging it with the world of OPC.

The power of reactive extensions is in their high level of abstraction, asynchronous nature, and ability of composition.

Microsoft provides Reactive Extensions (Rx) as a NuGet package. Reference the corresponding package in your project, in order to get access to the Rx types. In Visual Studio, use the “Tools -> Library Package Manager” command, and search for “Reactive Extensions – Main Library”, as on the following picture.



#### 9.1.1 OPC Reactive Extensions (Rx/OPC)

OPC Reactive Extensions (Rx/OPC) are OPC-specific classes that implement the Rx interfaces. With Rx/OPC, you can e.g.:

- Create observable collections (data streams) that provide OPC data.
- Consume (observe) data streams, and process them using OPC.
- Combine data flows to create more complex operations.

For OPC Data Access and OPC-UA data, Rx/OPC gives you the ability to work with flows representing OPC (monitored) item changes, or flows that represent values to be written into an OPC item (node).

For example, your application may be required to continuously monitor the value of an OPC item, and when it is available (there is no failure obtaining the value), make some computations with it (we will multiply the value by 1000), and write the results into a different OPC item. This logic can be expressed by following code:

## Reactive transfer of OPC DA item values

```
DAItemChangedObservable.Create<int>(
    "", "OPCLabs.KitServer.2", "Simulation.Incrementing (1 s)", 100
    .Where(e => e.Exception == null)
    .Select(e => e.Vtq.Value * 1000)
    .Subscribe(DAWriteItemValueObserver.Create<int>(
        "", "OPCLabs.KitServer.2", "Simulation.Register_I4"));
```

Let's dissect what this example does:

1. It creates an observable sequence for significant changes in OPC-DA item "Simulation.Incrementing (1 s)".
2. The "Where" clause filters (from the observable sequence) only the changes that have a null Exception, i.e. those that carry a valid `DAVtq` object (value/timestamp/quality).
3. The "Select" clause takes the actual value from the `Vtq` property (it is of type `DAVtq<int>`), and returns the value multiplied by 1000.
4. An observer that writes incoming values into the "Simulation.Register\_I4" OPC-DA item is created.
5. The observer is subscribed to the transformed (processed) observable sequence.

As you can see, the code that does this kind of OPC data processing is very concise – all the extra “plumbing” needed in procedural coding model is gone, and only the meaningful pieces remain. Programs written in this way clearly express their intent, and the logic that handles certain functionality is concentrated in one place and not spread around various classes and methods.

## 9.1.2 OPC Data Observables

The `DAItemChangedObservable<TValue>` class is an observable for changes in OPC Data Access item or multiple items of type `TValue`. The `UADataChangeNotificationObservable<TValue>` class is an observable for changes in OPC-UA monitored item or multiple items of type `TValue`. It represents a data stream with information about significant data changes in the subscribed items.

Each significant change is represented by an `OnNext` message of type `EasyDAItemChangedEventArgs<T>` (for OPC Classic) or `EasyUADataChangeNotificationEventArgs<T>` (for OPC-UA). This message is used both for successes, when the `Exception` property is a null reference and the `Vtq` property (for OPC Classic) or the `AttributeData` property (for OPC-UA) contains valid data, and for failures, when the `Exception` property is non-null and contains the failure information.

The `OnCompleted` and `OnError` messages (methods of the `IObserver`) are never sent (not even in case of error related to the OPC item), thus the data stream is not terminated. If your application requires, you can process the data stream further, and filter it or split it by success/failure as needed.

For OPC Classic, you can create instances of `DAItemChangedObservable<TValue>` either by using its constructor, or with use of a static `DAItemChangedObservable` class with several overloads of the `Create` method. The static `DAItemChangedObservable.Create` methods use the default underlying `EasyDAClient` object for OPC reactive extensions. If you need to set some parameters in the client object, you can use the `ClientSelector` property to specify them.

For OPC-UA, you can create instances of `UADataChangeNotificationObservable<TValue>` either by using its constructor, or with use of a static `UADataChangeNotificationObservable` class with several overloads of the `Create` method. The static `UADataChangeNotificationObservable.Create` methods use the default underlying `EasyUAClient`

object for OPC reactive extensions. If you need to set some parameters in the client object, you can use the [ClientSelector](#) property to specify them.

This approach allows the code be expressed only in terms of pure OPC logic, and be not tied to the actual way it is implemented.

The following code fragment creates the observable using one of the [Create](#) methods:

#### Creating OPC DA observable

```
DAItemChangedObservable<double> ramp =  
    DAItemChangedObservable.Create<double>(  
        "", "OPCLabs.KitServer.2", "Demo.Ramp", 1000);
```

It is recommended that you create the instances using the [DAItemChangedObservable.Create](#) (or [UADataChangeNotificationObservable.Create](#)) methods unless you have special needs.

## 9.1.3 OPC Data Observers

The [DAWritelItemValueObserver](#)<TValue> is an observer that writes incoming values into an OPC Data Access item. The [UAWriteValueObserver](#)<TValue> is an observer that writes incoming values into an OPC-UA node (more precisely, into an attribute of a node).

Each [OnNext](#) method call on its [IObserver](#)<TValue> interface writes the value in the argument into the OPC item or node attribute associated with the observer. [OnCompleted](#) and [OnError](#) methods do nothing.

For OPC Classic, you can create instances of [DAWritelItemValueObserver](#)<TValue> either by using its constructor, or with use of a static [DAWritelItemValueObserver](#) class with several overloads of the [Create](#) method. The static [DAWritelItemValueObserver.Create](#) methods use the default underlying [EasyDAClient](#) object for OPC reactive extensions. If you need to set some parameters in the client object, you can use the [ClientSelector](#) property to specify them.

For OPC-UA, you can create instances of [UAWriteValueObserver](#)<TValue> either by using its constructor, or with use of a static [UAWriteValueObserver](#) class with several overloads of the [Create](#) method. The static [UAWriteValueObserver.Create](#) methods use the default underlying [EasyUAClient](#) object for OPC reactive extensions. If you need to set some parameters in the client object, you can use the [ClientSelector](#) property to specify them.

This approach allows the code be expressed only in terms of pure OPC logic, and be not tied to the actual way it is implemented.

The following code fragment creates the observer using one of the [Create](#) methods:

#### Creating OPC DA write observer

```
DAWriteItemValueObserver<int> observer =  
    DAWriteItemValueObserver.Create<int>(  
        "", "OPCLabs.KitServer.2", "Simulation.Register_I4");
```

It is recommended that you create the instances using the [DAWritelItemValueObserver.Create](#) (or [UAWriteValueObserver.Create](#)) methods unless you have special needs.

## 9.2 Reactive Programming Model for OPC Alarms and Events

Microsoft Reactive Extensions (Rx) and reactive programming in general is a big subject and cannot be covered in any

detail here. We will only describe the specifics that QuickOPC brings to Rx, by bridging it with the world of OPC. The power of reactive extensions is in their high level of abstraction, asynchronous nature, and ability of composition.

## 9.2.1 OPC Reactive Extensions (Rx/OPC)

OPC Reactive Extensions (Rx/OPC) are OPC-specific classes that implement the Rx interfaces. With Rx/OPC, you can e.g.:

- Create observable collections (data streams) that provide OPC events.
- Consume (observe) data streams, and process them using OPC.
- Combine data flows to create more complex operations.

For OPC Alarms & Events, Rx/OPC gives you the ability to work with flows representing OPC event notifications, or flows that represent OPC event conditions to be acknowledged.

## 9.2.2 OPC-A&E Observables

The [AENotificationObservable](#) class is an observable for notifications about OPC events. It represents a data stream with information about subscribed events.

Each significant change is represented by an [OnNext](#) message of type [EasyAENotificationEventArgs](#). This message is used both for successes, when the [Exception](#) property is a null reference, and for failures, when the [Exception](#) property is non-null and contains the failure information.

The [OnCompleted](#) and [OnError](#) messages (methods of the [IObserver](#)) are never sent (not even in case of error related to the OPC server communication), thus the data stream is not terminated. If your application requires, you can process the data stream further, and filter it or split it by success/failure as needed.

You can create instances of [AENotificationObservable](#) either by using its constructor, or with use of several overloads of the static [Create](#) method. The static [AENotificationObservable.Create](#) methods use the default underlying [EasyAEClient](#) object for OPC reactive extensions. If you need to set some parameters in the client object, you can use the [ClientSelector](#) property to specify them. This allows the code be expressed only in terms of pure OPC logic, and be not tied to the actual way it is implemented.

It is recommended that you create the instances using the [AENotificationObservable.Create](#) methods unless you have special needs.

## 9.2.3 OPC-A&E Observers

The [AEAcknowledgeConditionObserver](#) is an observer that acknowledges OPC event conditions according to incoming values.

Each [OnNext](#) method call on its [IObserver<AEAcknowledgeConditionArguments>](#) interface acknowledges an OPC event condition according to the arguments passed to it. [OnCompleted](#) and [OnError](#) methods do nothing.

You can create instances of [AEAcknowledgeConditionObserver](#) either by using its constructor, or with use of several overloads of the static [Create](#) method. The static [AEAcknowledgeConditionObserver.Create](#) methods use the default underlying [EasyAEClient](#) object for OPC reactive extensions. If you need to set some parameters in the client object, you can use the [ClientSelector](#) property to specify them. This allows the code be expressed only in terms of pure OPC logic, and be not tied to the actual way it is implemented.

It is recommended that you create the instances using the [AEAcknowledgeConditionObserver.Create](#) methods unless you have special needs.

## 10 User Interface

The following table shows which features are available for different target platforms:

	.NET	COM
<b>OPC Common Dialogs</b>	yes	yes
<b>OPC Controls</b>	yes	no

### 10.1 OPC Common Dialogs

QuickOPC contains a set of Windows Forms dialog boxes for performing common OPC-related tasks such as selecting an OPC server or OPC item in OPC "Classic", or selecting an OPC server endpoint, or an OPC data node within the server, in OPC Unified Architecture.

The dialog objects are all derived from [System.Windows.Forms.CommonDialog](#), providing consistent and well-known programming interface to use.

You can drag any of the dialog components from the Visual Studio toolbox to the designer surface, and then configure its properties. It is also possible to use the dialog components without the designer support – simply instantiate the dialog class, and work with its properties and methods from your code.

The properties of each dialog that have significance for its functionality are categorized as follows:

Category	Description
Mode	Determines the overall behavior of the dialog. For example, some dialogs allow you to choose whether a single selection or multi-selection will be performed.
Inputs	Information necessary for running the dialog that the user cannot change. For example, with an OPC-DA Item Dialog, the dialog inputs consists of the machine name and the OPC server to be browsed.
Inputs/Outputs	Information that can be provided as initial data, but the dialog user can change it. For example, with an OPC-DA Item Dialog, the information about the selected item can be passed into the dialog so that it is pre-selected at the beginning. Corresponding information may also exist in the Outputs category; the difference is that the Outputs contain more details (typically in the form of some of the <a href="#">XXXXElement</a> classes), while the Inputs/Outputs only contains the necessary identifying data (could be in form of <a href="#">XXXXDescriptor</a> classes).
Outputs	The selection(s) that the user has made in the dialog.

All OPC browsing dialogs have an additional [SizeFactor](#) property. The property allows the developer to influence the (initial) size of the dialog on screen. It can be set to one of the pre-defined values ([SizeFactors.Small/Normal/Large/ExtraLarge](#)), or to any [SizeF](#) structure, defining a width and height relative to the normal size. An additional [SizeFactorName](#) is provided primarily to allow easy access to this feature to COM callers.

All OPC browsing dialogs also have an additional [RetainAppearance](#) property. When set (which is the default behavior), the dialog retains its appearance (mainly, the location and size), between invocations. It is also possible to call the [RevertAppearance\(\)](#) method on the dialog, to restore the original values.

When any of the common dialog components is placed onto the designer surface and its properties configured, it can be tested without a need to build the project. To perform the test, use the "Test Dialog" command, available from the context menu on the dialog component itself, or the same command in the Properties window, when the dialog

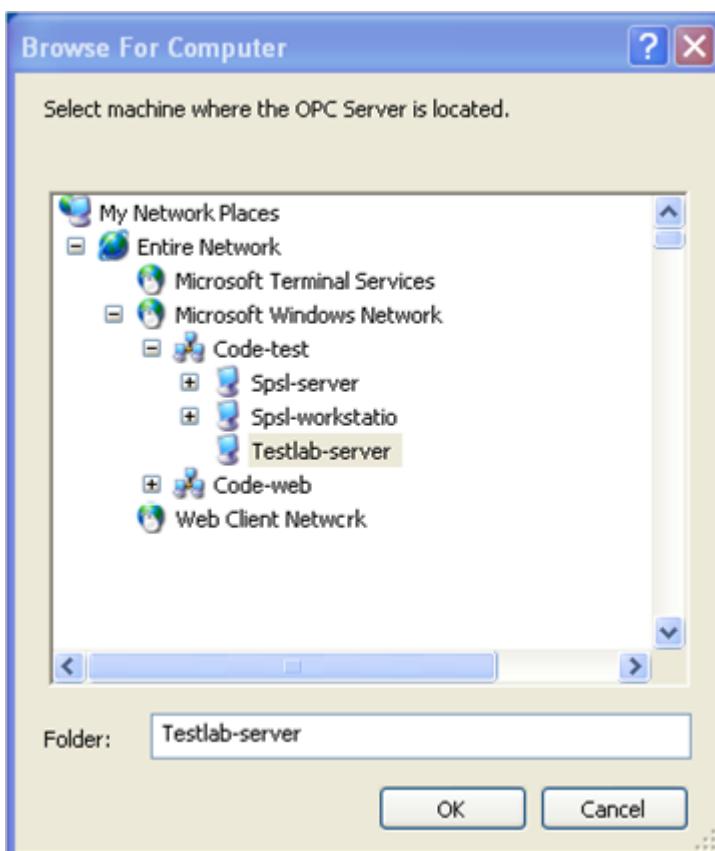
component is selected (you may have to right-click in the property grid and check "Commands" first, to display the commands under the property grid).

## 10.1.1 OPC-DA Common Dialogs

### 10.1.1.1 Computer Browser Dialog

Icon: 

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 [ComputerBrowserDialog](#) 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.

#### C++

```
// This example shows how to let the user browse for computers on the network.

#include "stdafx.h"
#include "_ComputerBrowserDialog.ShowDialog.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"        // mscorelbi
using namespace mscorelbi;

#import "System.Drawing.tlb"
using namespace System_Drawing;
```

```
#import "System.Windows.Forms.tlb"
using namespace System_Windows_Forms;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

#import "libid:A0D7CA1E-7D8C-4D31-8ECB-84929E77E331"      // OpcLabs.BaseLibForms
using namespace OpcLabs_BaseLibForms;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _ComputerBrowserDialog
{
    void ShowDialog::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            //
            _ComputerBrowserDialogPtr DialogPtr(__uuidof(ComputerBrowserDialog));
            //

            DialogResult dialogResult = DialogPtr->ShowDialog(NULL);

            // Display results
            _tprintf(_T("%d\n"), dialogResult);
            _tprintf(_T("%s\n"), CW2T(DialogPtr->SelectedName));
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}
```

## Free Pascal

```
// This example shows how to let the user browse for computers on the network.

class procedure ShowDialog.Main;
var
    Dialog: ComputerBrowserDialog;
begin
    // Instantiate the dialog object
    Dialog := CoComputerBrowserDialog.Create;

    Dialog.ShowDialog(nil);
```

```
  WriteLn(Dialog.SelectedName);  
end;
```

## Object Pascal

```
// This example shows how to let the user browse for computers on the network.  
  
class procedure ShowDialog.Main;  
var  
  ComputerBrowserDialog: TComputerBrowserDialog;  
begin  
  // Instantiate the dialog object  
  ComputerBrowserDialog := TComputerBrowserDialog.Create(nil);  
  
  ComputerBrowserDialog.ShowDialog(nil);  
  WriteLn(ComputerBrowserDialog.SelectedName);  
end;
```

## PHP

```
// This example shows how to let the user browse for computers on the network.  
  
$Dialog = new COM("OpcLabs.BaseLib.Forms.Browsing.ComputerBrowserDialog");  
printf("%d\n", $Dialog->ShowDialog);  
  
// Display results  
printf("%s\n", $Dialog->SelectedName);
```

## Python

```
# This example shows how to let the user browse for computers on the network.  
  
import win32com.client  
  
dialog =  
win32com.client.Dispatch('OpcLabs.BaseLib.Forms.Browsing.ComputerBrowserDialog')  
print(dialog.ShowDialog())  
  
# Display results  
print(dialog.SelectedName)
```

## Visual Basic (VB 6.)

```
Rem This example shows how to let the user browse for computers on the network.  
  
Private Sub ShowDialog_Main_Command_Click()  
  OutputText = ""  
  
  Dim Dialog As New ComputerBrowserDialog  
  Dim DialogResult  
  DialogResult = Dialog.ShowDialog  
  
  ' Display results  
  OutputText = OutputText & DialogResult & vbCrLf  
  OutputText = OutputText & Dialog.SelectedName & vbCrLf  
End Sub
```

## VBScript

```
Rem This example shows how to let the user browse for computers on the network.
```

## Option Explicit

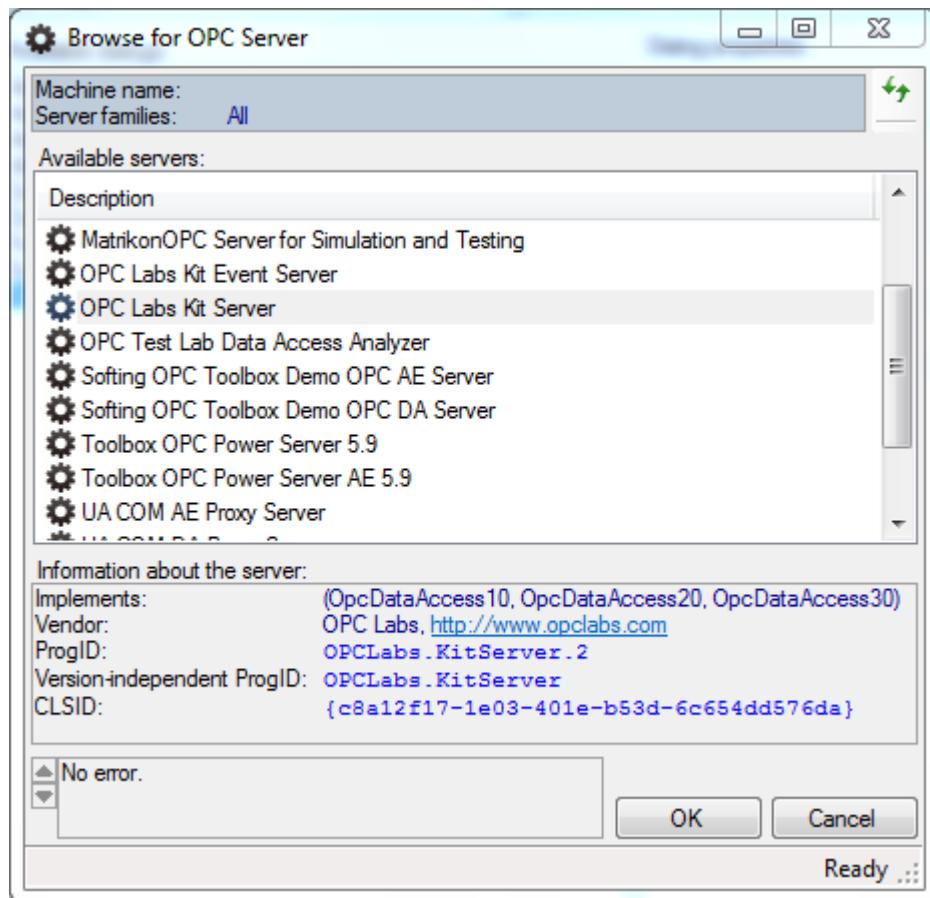
```
Dim Dialog: Set Dialog =  
CreateObject("OpcLabs.BaseLib.Forms.Browsing.ComputerBrowserDialog")  
WScript.Echo Dialog.ShowDialog  
  
' Display results  
WScript.Echo Dialog.SelectedName
```

## 10.1.1.2 OPC Server Dialog

Icon : 

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 ([OpcServerDialog class](#)) allows the user to select the OPC server interactively from the list of OPC Data Access servers installed on a particular machine.

Here is an example of OPC Server dialog in action:



The [ServerFamilies](#) property of the dialog determines whether OPC Data Access servers, OPC Alarms&Events servers, other servers, or a combination of them will be browsed. By default, all server families are browsed. In order to limit the browsing to OPC Data Access servers only, set the [ServerFamilies](#) property to [OpcDataAccess](#) enumeration member.

To run the dialog, set the [Location](#) 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 [ServerElement](#) property.

## VBScript

Rem This example shows how to let the user browse for an OPC "Classic" server.

```
Option Explicit

Dim ServerDialog: Set ServerDialog =
CreateObject("OpcLabs.EasyOpc.Forms.Browsing.OpcServerDialog")
'ServerDialog.Location = ""
WScript.Echo ServerDialog.ShowDialog

' Display results
WScript.Echo ServerDialog.ServerElement
```

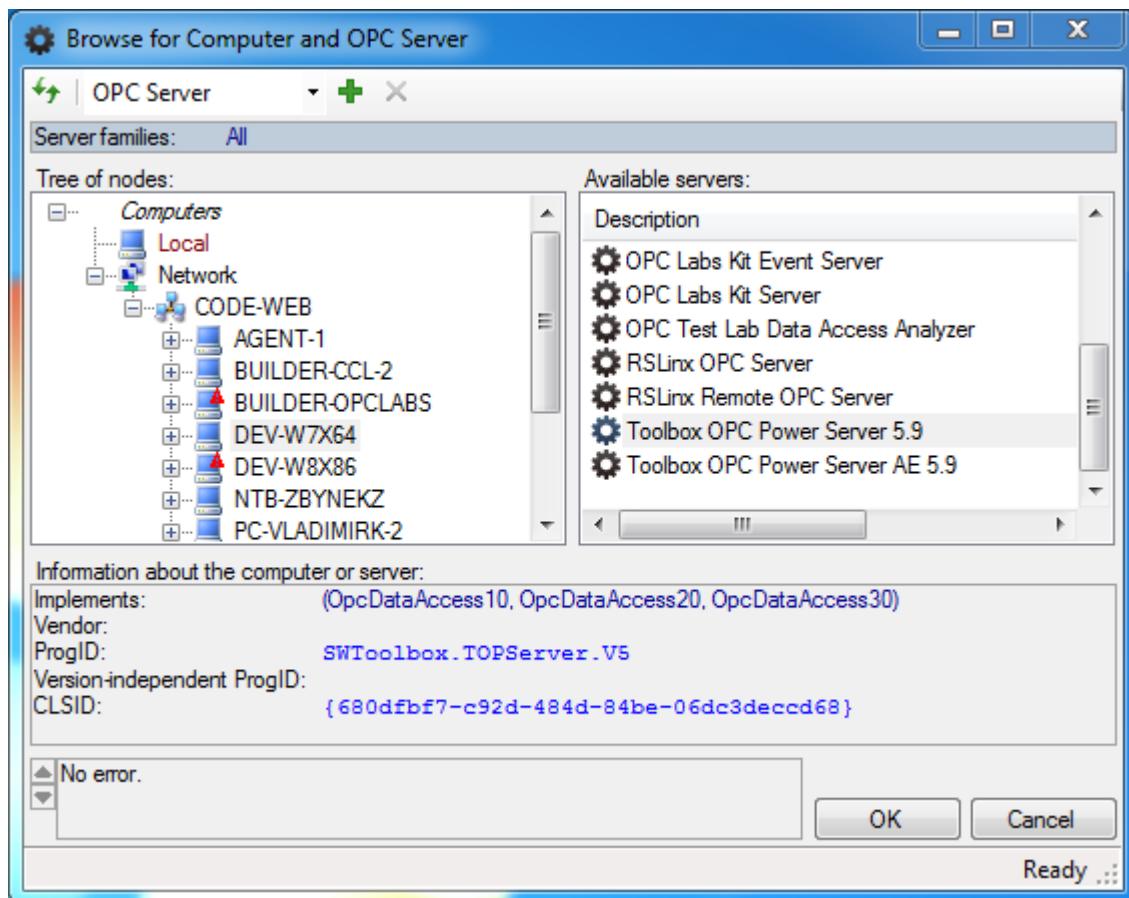
### 10.1.1.3 OPC Computer and Server Dialog

Icon: 

With [OpcComputerAndServerDialog](#), your application can integrate a dialog box from which the user can select a computer and an OPC server residing on it. This dialog box combines functions of the Computer Browser Dialog -  and OPC Server Dialog into a single dialog.

In addition, the user can add Computer nodes in case the particular computer is not visible on the network, or add OPC Server nodes in case the OPC Server is known to exist but is not returned by OPC Server enumeration feature.

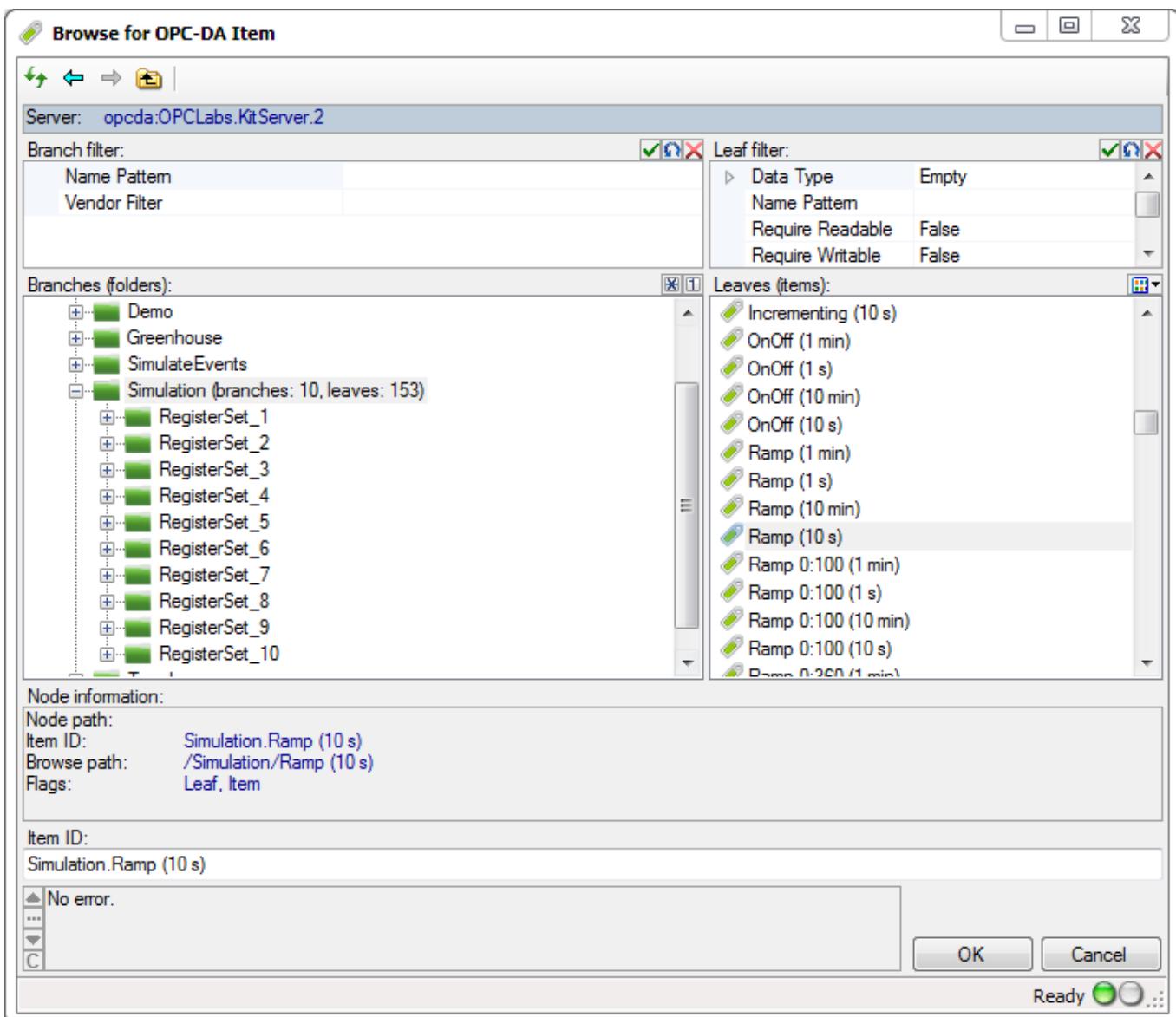
Here is an example of OPC Computer and Server dialog in action:



## 10.1.1.4 OPC-DA Item Dialog

Icon: 

The OPC-DA Item Dialog ([TDAItemDialog](#) class) allows the user to interactively select the OPC item residing in a specific OPC server.



Use the [ServerDescriptor](#) 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 [NodeElement](#) property.

### Object Pascal

```
// This example shows how to let the user browse for an OPC Data Access item.

class procedure ShowDialog.Main;
var
  ItemDialog: TDAItemDialog;
begin
  // Instantiate the dialog object
```

```
ItemDialog := TDAItemDialog.Create(nil);  
ItemDialog.ServerDescriptor.ServerClass := 'OPCLabs.KitServer.2';  
ItemDialog.ShowDialog(nil);  
  
// Display results  
WriteLn(ItemDialog.NodeElement.ToString);  
end;
```

## VBScript

---

Rem This example shows how to let the user browse for an OPC Data Access item.

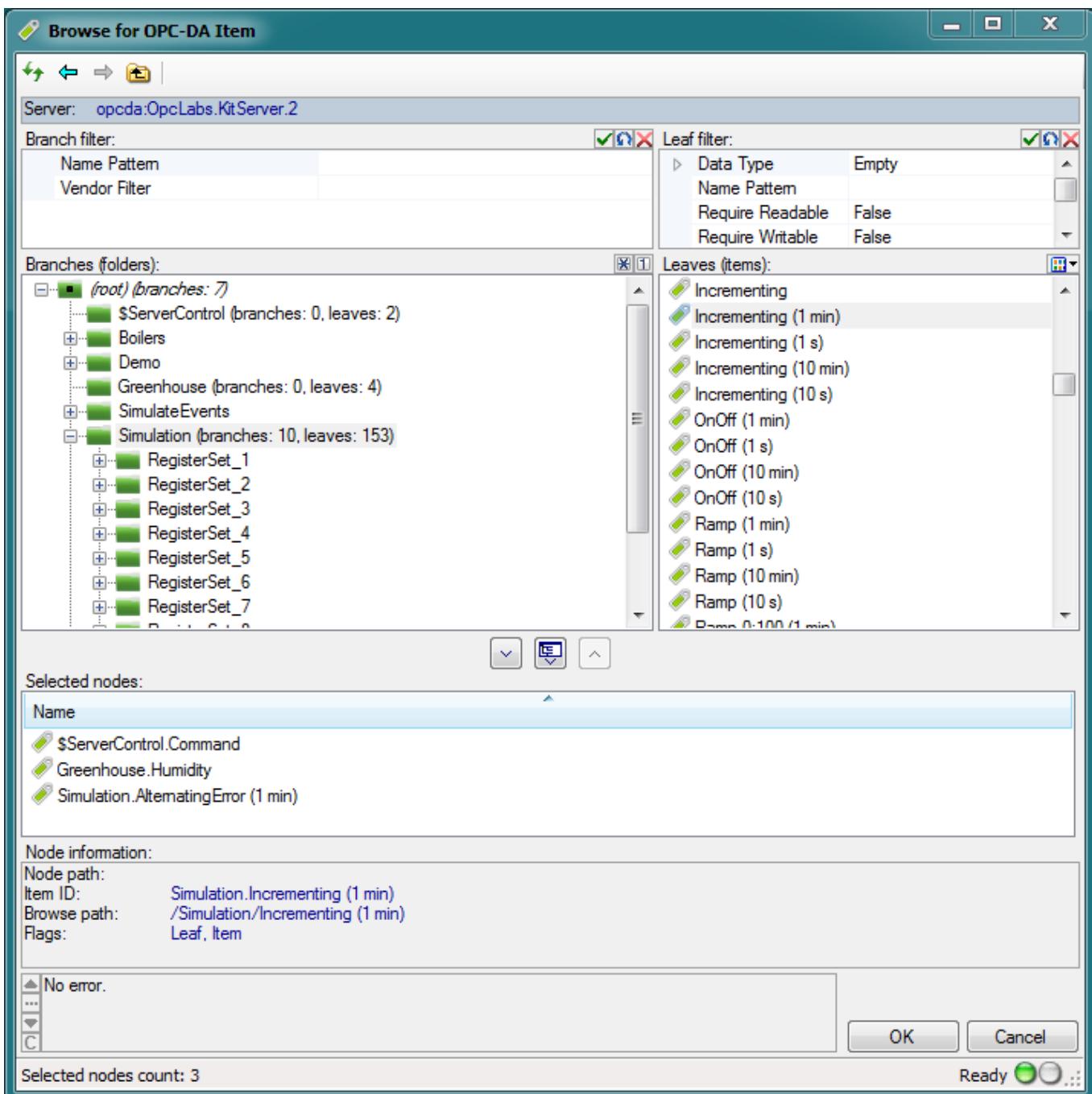
Option Explicit

```
Dim ItemDialog: Set ItemDialog =  
CreateObject("OpcLabs.EasyOpc.DataAccess.Forms.Browsing.DAItemDialog")  
ItemDialog.ServerDescriptor.ServerClass = "OPCLabs.KitServer.2"  
WScript.Echo ItemDialog.ShowDialog  
  
' Display results  
WScript.Echo "NodeElement: " & ItemDialog.NodeElement
```

The [DALItemDialog](#) component retains the filter setting for each node between the invocations of the dialog, making it faster for the user to navigate during the subsequent invocations.

## Multi-selection

When you set the [MultiSelect](#) property of the DALItemDialog to true, the dialog will allow the user to select any number of OPC-DA items. In the multi-select mode, the dialog looks similar to this:



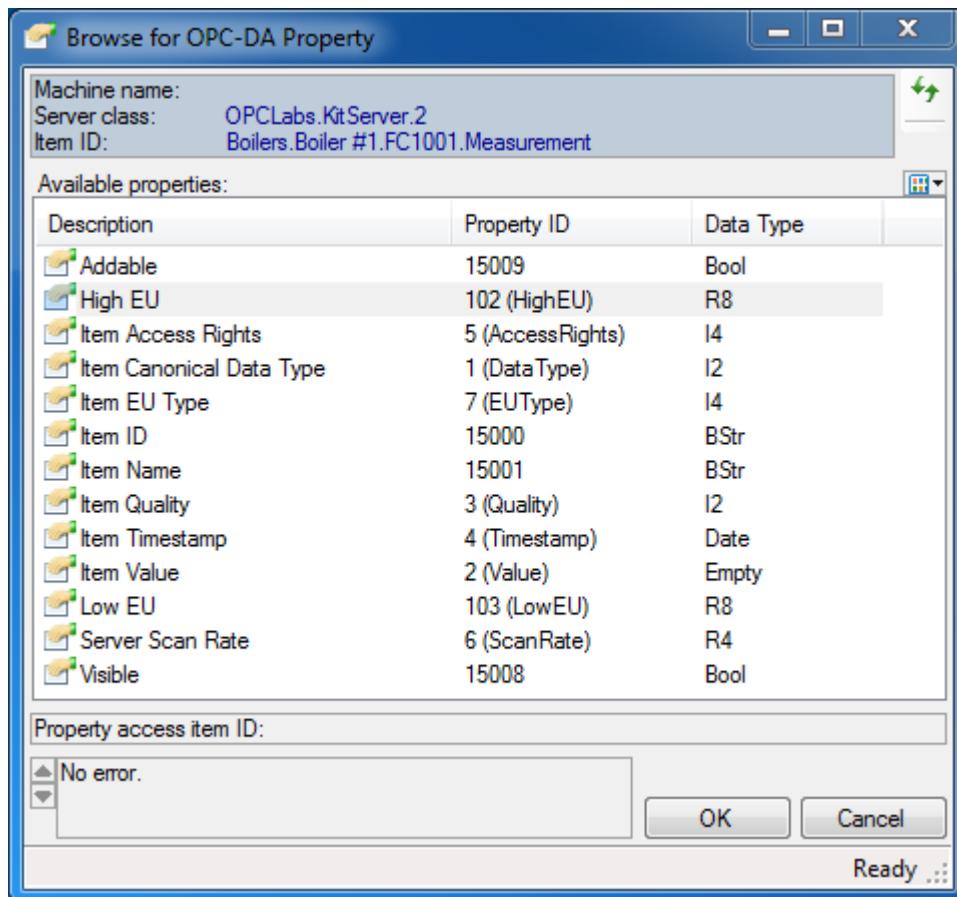
The list below the branches and leaves (labeled "Selected nodes") contains the set of nodes that the user has selected in the dialog. The user can freely add nodes to this list, or remove them. The selected set is carried over to next invocation of the dialog, unless you change it from the code.

In the multi-selection mode, the set of nodes selected on input (if any) is in the [NodeDescriptors](#) property. On the output, the dialog fills the information about selected nodes into the [NodeElements](#) property (and updated the [NodeDescriptors](#) property as well).

## 10.1.1.5 OPC-DA Property Dialog

Icon:

The OPC-DA Property Dialog ([DAPropertyDialog](#) class) allows the user to interactively select the OPC property on a specific OPC item.



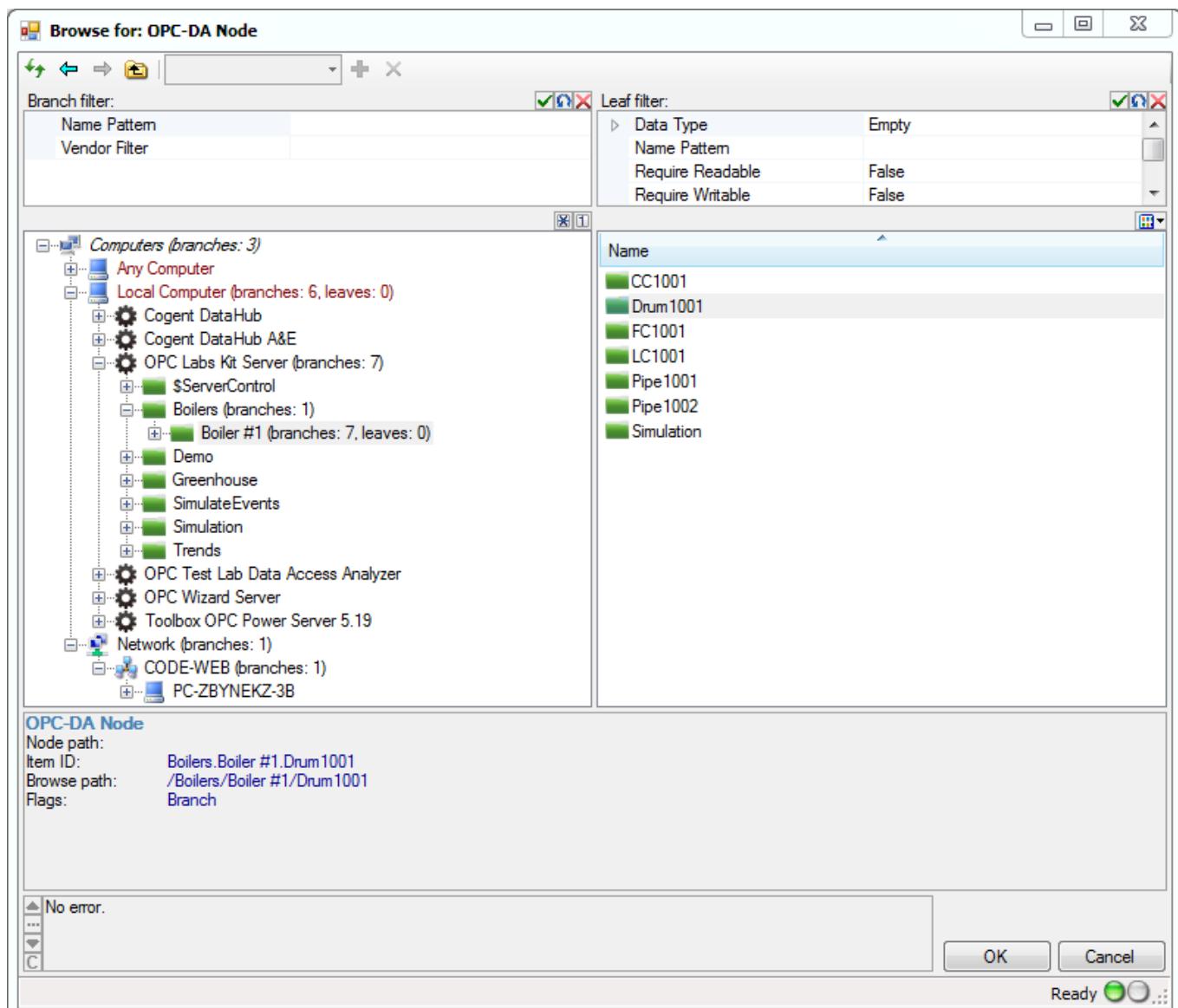
Use the [ServerDescriptor](#) property to specify the OPC Data Access server whose items are to be browsed, set the [NodeDescriptor](#) property to the OPC Item Id needed, and then call the [ShowDialog](#) method. If the result is equal to [DialogResult.OK](#), the user has selected the OPC property, and information about it can be retrieved from the [PropertyElement](#) property.

## 10.1.1.6 Generic OPC Browsing Dialog

Icon:

With [OpcBrowseDialog](#), your application can integrate a dialog with various OPC nodes from which the user can select. This dialog can be configured to serve many different purposes.

Here is an example of the generic OPC browsing dialog in action:

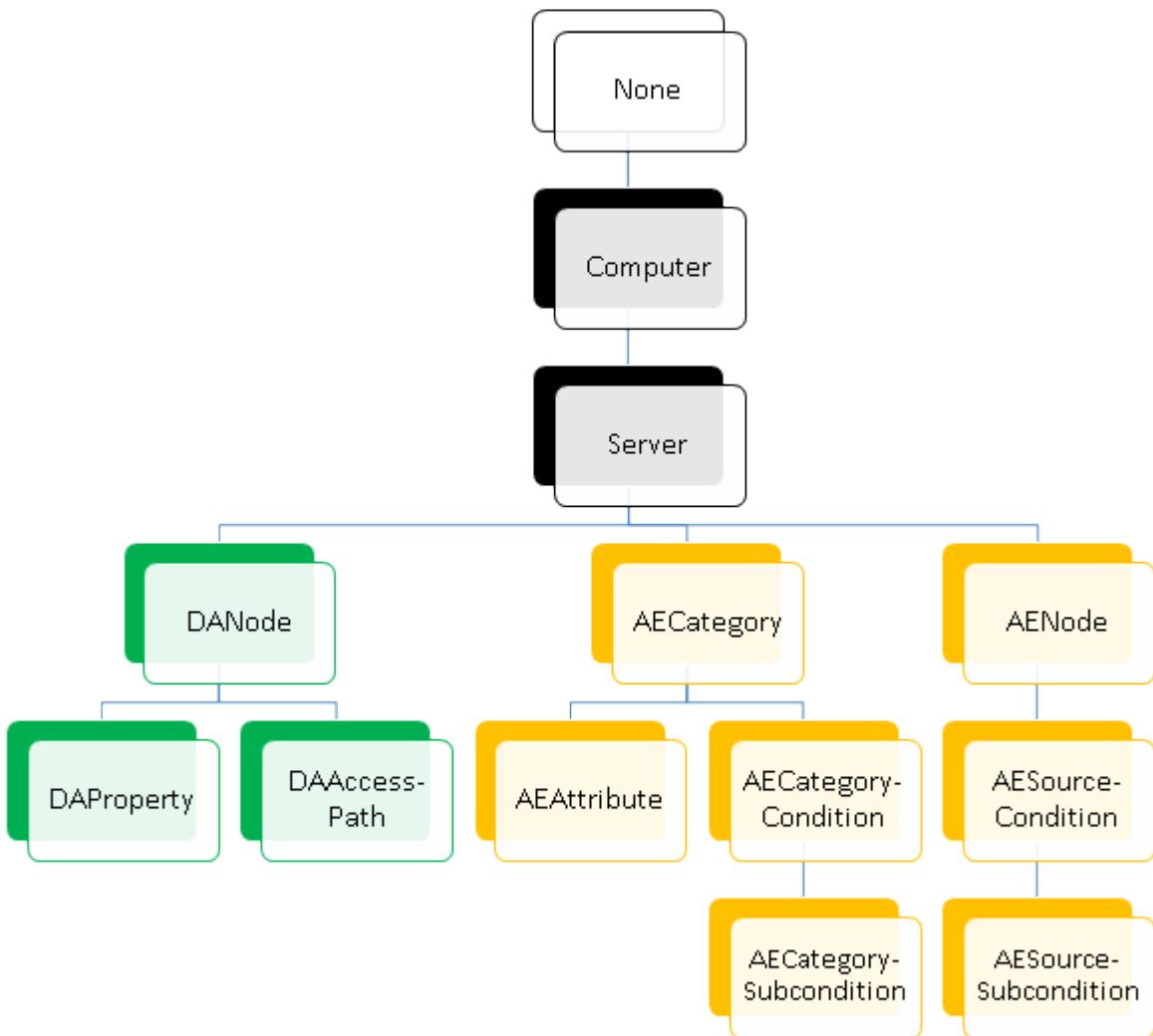


The way the dialog operates is controlled by two main properties:

- `Mode.AnchorElementType` determines the part of the selection that is given on input and cannot be changed by the user.
- `Mode.SelectElementType` determines the type of the element that the user needs to select in order to finalize the dialog.

Values of these properties can be selected from the `OpcElementType` enumeration, which has members for various types of elements that you encounter when working with OPC.

The following chart shows a hierarchy of element types that you can choose from:



For example, let's say that you set [Mode.AnchorElementType](#) to [Server](#), and [Mode.SelectElementType](#) to [DAProperty](#). This will cause the dialog to allow the user to browse for an OPC Item (node) on the server you specify, and then for an OPC property on that item.

In this case, before you run the dialog, you need to provide it with values for the [InputOutputs.CurrentNodeDescriptor.ServerDescriptor.Location](#) and [InputOutputs.CurrentNodeDescriptor.ServerDescriptor.ServerClass](#) properties, because those define your "anchor" element ([Server](#)) that the user cannot change. The dialog will only allow the user to finalize it (besides cancelling) after an OPC property is fully selected, because that is your [Mode.SelectElementType](#). After the dialog is successfully finalized, the information about the user's choice will be available in the [Outputs.CurrentNodeElement.DANodeElement](#) and [Outputs.CurrentNodeElement.DAPropertyElement](#) properties.

Note that in addition to the "minimal" scenario described above, you can also pre-set the initial node or property, using the [InputOutputs.CurrentNodeDescriptor.DANodeDescriptor](#) or [InputOutputs.CurrentNodeDescriptor.DAPropertyDescriptor.PropertyId](#) properties, and after the selection is made, these properties will be updated to the new selection as well. This way, if you run the dialog again with the same

value, the initial selection will be where the user has left it the last time the dialog was run.

Obviously, the chosen [Mode.SelectElementType](#) must be a child or indirect ancestor of chosen [Mode.AnchorElementType](#) in the hierarchy. For example, it would be an error to set [Mode.AnchorElementType](#) to [AECCategory](#) and [Mode.SelectElementType](#) to [DAProperty](#).

## Object Pascal

```
// This example shows how to let the user browse for an OPC Data Access node.

class procedure ShowDialog.Main;
var
  BrowseDialog: TOpcBrowseDialog;
begin
  // Instantiate the dialog object
  BrowseDialog := TOpcBrowseDialog.Create(nil);

  BrowseDialog.ShowDialog(nil);

  // Display results
  WriteLn(BrowseDialog.Outputs.CurrentNodeElement.DANodeElement.ToString());
end;
```

## VBScript

```
Rem This example shows how to let the user browse for an OPC Data Access node.

Option Explicit

Dim BrowseDialog: Set BrowseDialog =
CreateObject("OpcLabs.EasyOpc.Forms.Browsing.OpcBrowseDialog")
WScript.Echo BrowseDialog.ShowDialog

' Display results
WScript.Echo BrowseDialog.Outputs.CurrentNodeElement.DANodeElement
```

## Multi-selection

It is also possible to configure the dialog for a multi-selection. In this mode, the user can select zero, one, or more nodes. In order to enable the multi-select mode, set the [Mode.MultiSelect](#) property to true. In the multi-select mode, the initial set of the selected nodes (when the dialog is first displayed to the user) is given by the contents of the [InputOutputs.SelectionDescriptors](#) collection. When the user makes the selection and accepts it by closing the dialog, this collection is updated, and also, all information about the selected nodes is placed to the [Outputs.SelectionElements](#) collection.

## Other settings

There are also ways to control some finer aspects of the dialog. For example, the [Mode.ShowListBranches](#) property (defaults to true) controls whether the branches of the tree are also displayed in the list view.

### 10.1.2 OPC-A&E Common Dialogs

#### 10.1.2.1 Computer Browser Dialog

Icon: 

With [ComputerBrowserDialog](#) object, you can present your user with a dialog for selecting the remote computer where the OPC server resides.

For more information, see the Computer Browser dialog under "User Interface for OPC-DA".

## 10.1.2.2 OPC Server Dialog

Icon: 

The OPC Server Dialog ([OpcServerDialog](#) class) allows the user to select the OPC server interactively from the list of OPC "Classic" servers installed on a particular machine.

To select from OPC Alarms&Events servers, you use the same dialog as with OPC Data Access servers – for details on that dialog, see OPC Server Dialog under "User Interface for OPC-DA". The only difference is that before running the dialog, you need to make sure that its [ServerFamilies](#) property is set so that it includes (or only contains) the [OpcAlarmsAndEvents](#) enumeration member.

## 10.1.2.3 OPC Computer and Server Dialog

Icon: 

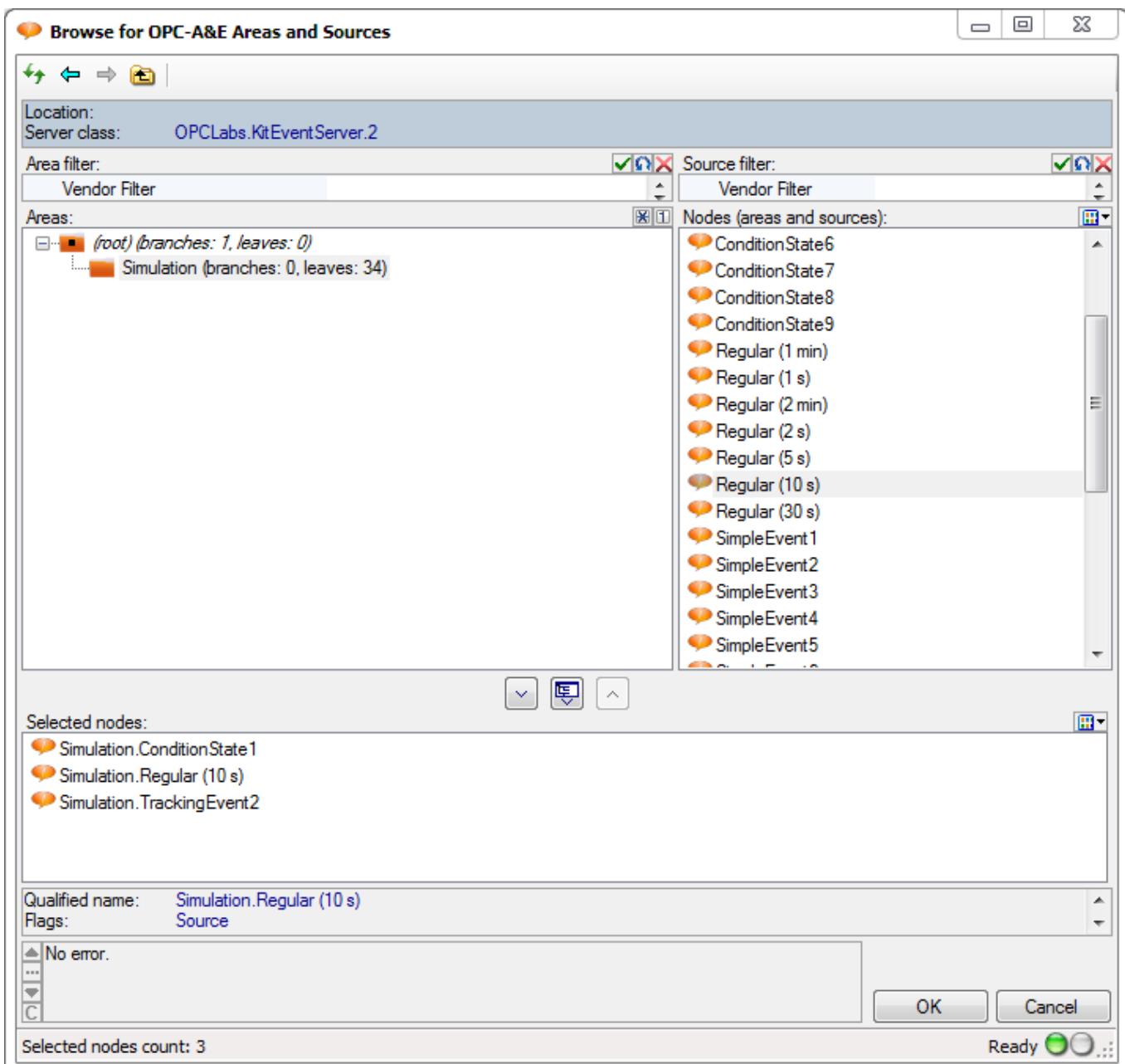
With [OpcComputerAndServerDialog](#), your application can integrate a dialog box from which the user can select a computer and an OPC server residing on it.

This dialog is identical to OPC Computer and Server Dialog described in the OPC-DA Common Dialogs chapter. The only difference is that before running the dialog, you need to make sure that its [ServerFamilies](#) property is set so that it includes (or only contains) the [OpcAlarmsAndEvents](#) enumeration member.

## 10.1.2.4 OPC-A&E Area or Source Dialog

Icon: 

With [AEAreaOrSourceDialog](#), your applicate can integrate a dialog box from which the user can select OPC-A&E event areas or sources:



The [MultiSelect](#) property determines whether the dialog allows the user to select multiple nodes as output.

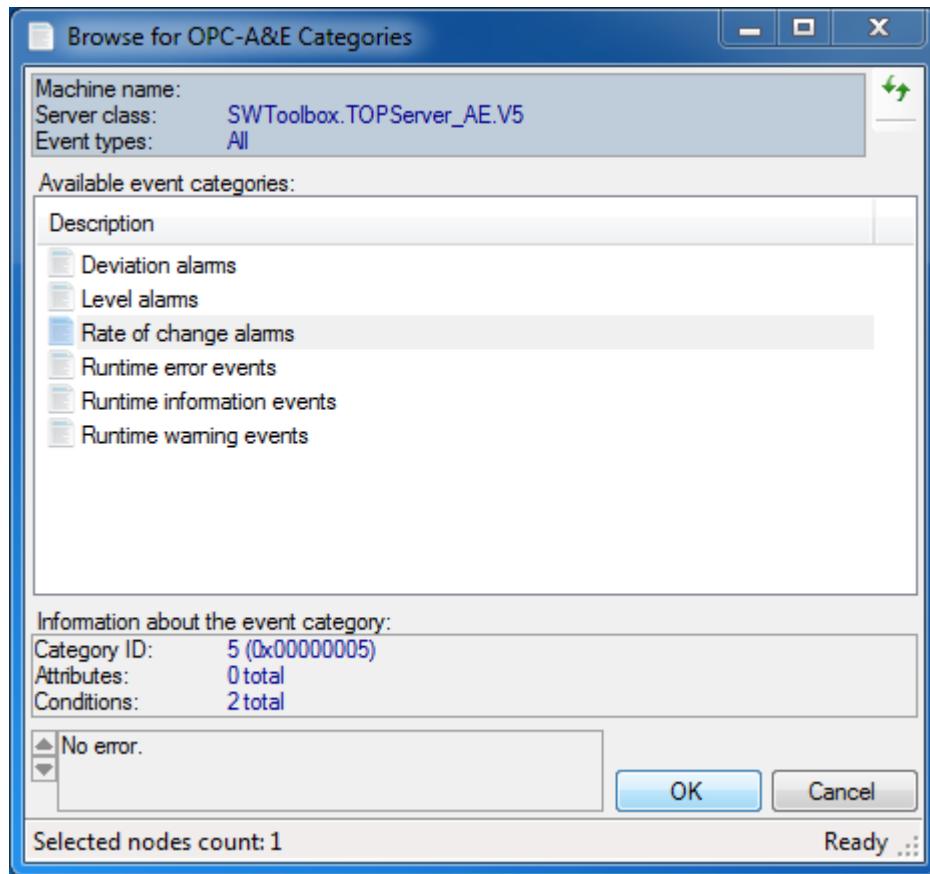
Use the [ServerDescriptor](#) property to specify the OPC Alarms&Events server whose items are to be browsed, and call the [ShowDialog](#) method. If the result is equal to [DialogResult.OK](#), the user has finished the selection, and information about it can be retrieved.

When the dialog is in single-select mode, you can obtain the information about the selected event area or source from the [NodeDescriptor](#) property. When the dialog is in multi-select mode, the information about the selected event areas is in the [AreaElements](#) property, and the information about the selected event sources in the [SourceElements](#) property; all event areas and nodes are available together in the [NodeElements](#) property.

## 10.1.2.5 OPC-A&E Category Dialog

Icon:

With [AECategoryDialog](#), your application can integrate a dialog box from which the user can select from OPC-A&E event categories provided by the OPC server:



The [MultiSelect](#) property determines whether the dialog allows the user to select multiple categories as output. The [EventTypes](#) property determines which event type(s) will be offered for selection.

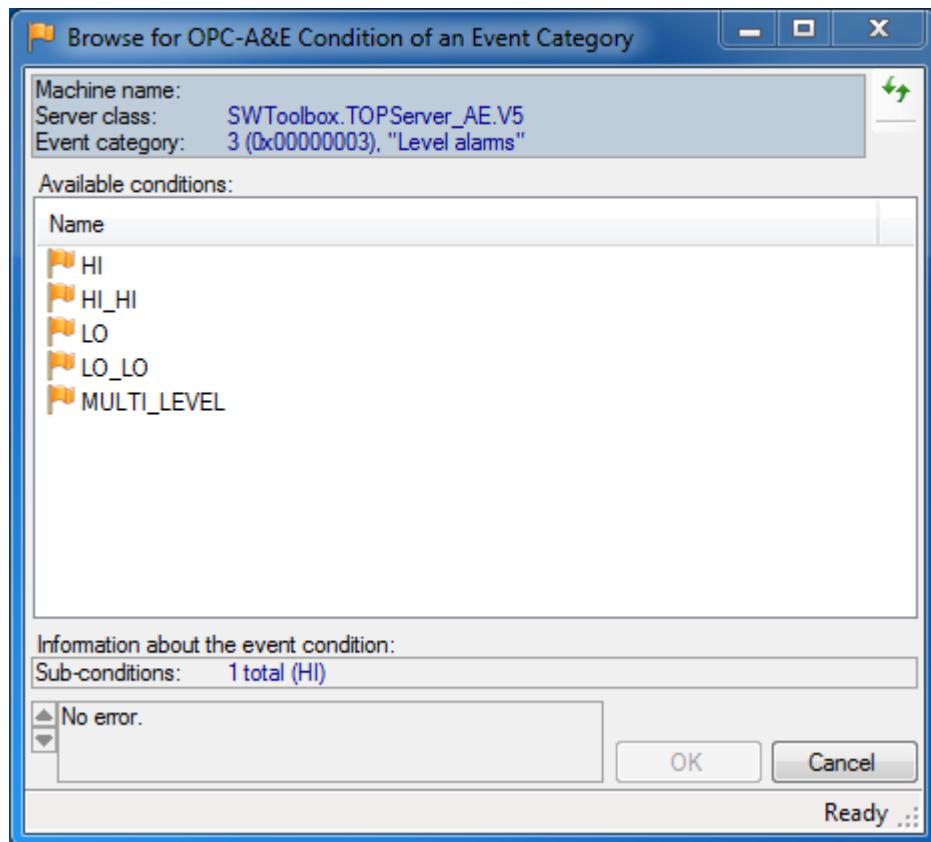
Use the [ServerDescriptor](#) property to specify the OPC Alarms&Events server whose categories are to be browsed, and call the [ShowDialog](#) method. If the result is equal to [DialogResult.OK](#), the user has finished the selection, and information about it can be retrieved.

When the dialog is in single-select mode, you can obtain the information about the selected category from the [CategoryElement](#) or [CategoryId](#) property. When the dialog is in multi-select mode, the information about the selected categories is in the [CategoryElements](#) and [CategoryIds](#) properties.

## 10.1.2.6 OPC-A&E Category Condition Dialog

Icon: 

With [AECategoryConditionDialog](#), your application can integrate a dialog box from which the user can select OPC-A&E category available on a specified event condition:

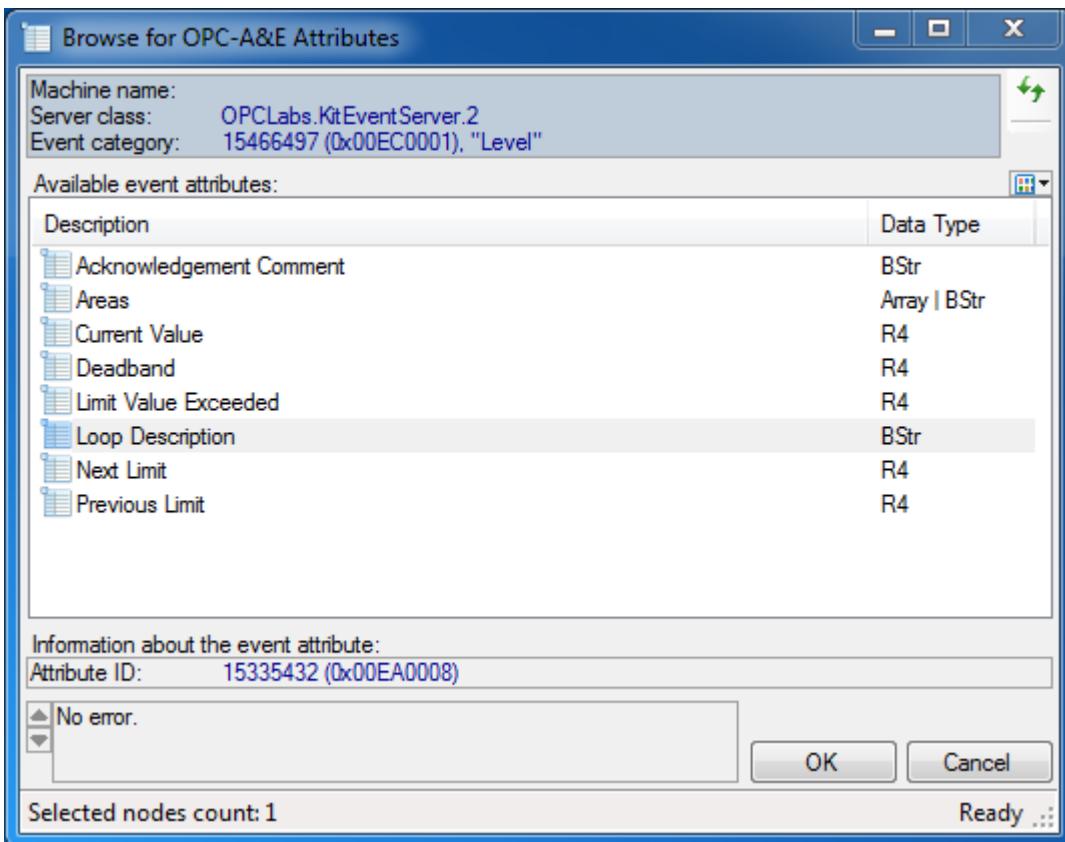


Use the [ServerDescriptor](#) property to specify the OPC Alarms&Events server, and the [CategoryId](#) property to specify the event category to be browsed. Then, call the [ShowDialog](#) method. If the result is equal to [DialogResult.OK](#), the user has selected the event condition, and information about it can be retrieved from the [ConditionElement](#) or [ConditionName](#) property.

## 10.1.2.7 OPC-A&E Attribute Dialog

Icon: 

With [AEAttributeDialog](#), your application can integrate a dialog box from which the user can select OPC-A&E event attributes:



The [MultiSelect](#) property determines whether the dialog allows the user to select multiple attributes as output.

Use the [ServerDescriptor](#) property to specify the OPC Alarms&Events server and the [CategoryId](#) property to specify the event category to be browsed. Then, call the [ShowDialog](#) method. If the result is equal to [DialogResult.OK](#), the user has finished the selection, and information about it can be retrieved.

When the dialog is in single-select mode, you can obtain the information about the selected attribute from the [AttributeElement](#) or [Attributeld](#) property. When the dialog is in multi-select mode, the information about the selected categories is in the [AttributeElements](#) and [Attributelds](#) properties.

## 10.1.2.8 Generic OPC Browsing Dialog

Icon:

With [OpcBrowseDialog](#), your application can integrate a dialog with various OPC nodes from which the user can select. This dialog can be configured to serve many different purposes.

As this dialog is shared with other OPC specifications, please refer to [Generic OPC Browsing Dialog](#) above for more information.

## 10.1.3 OPC-UA Common Dialogs

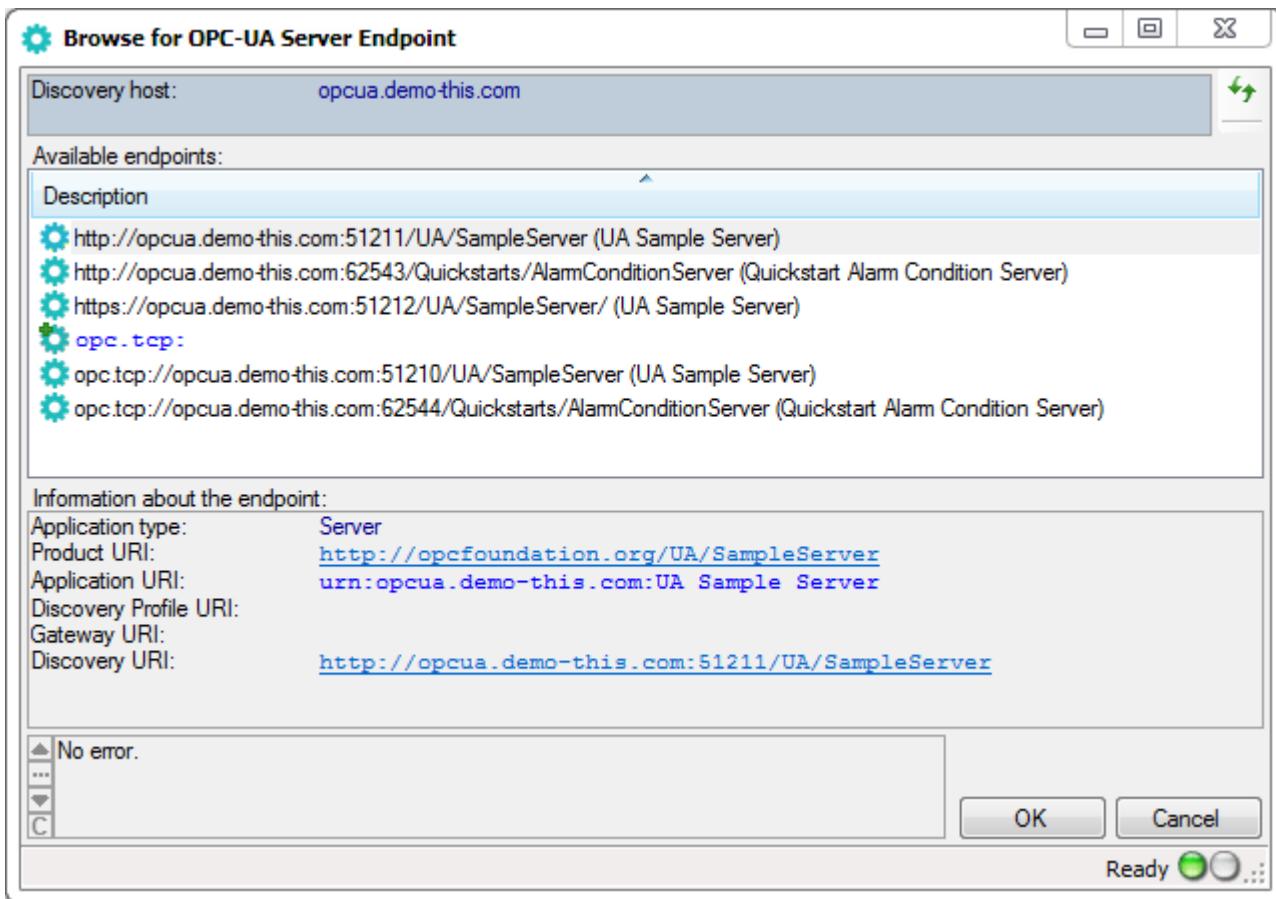
### 10.1.3.1 OPC-UA Endpoint Dialog

Icon:

If you do not know upfront which OPC-UA server and its endpoint 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-UA

Endpoint Dialog ([UAEndpointDialog](#) class) allows the user to select the OPC server and its endpoint interactively from the list of OPC Unified Architecture servers provided by a LDS (Local Discovery Server) installed on a machine.

Here is an example of OPC-UA Endpoint dialog in action:



To run the dialog, call the [ShowDialog](#) method. If the result is equal to [DialogResult.OK](#), the user has selected the OPC UA server, and information about it can be retrieved from the [ApplicationElement](#) property.

Currently, the dialog shows endpoint provided by a LDS (Local Discovery Server) that you set by a static property on the [EasyUAClient](#) object. This approach may change in future.

## C++

```
// This example shows how to let the user browse for an OPC-UA server endpoint.

#include "stdafx.h"
#include "_UAEndpointDialog.ShowDialog.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

#import "System.Drawing.tlb"
using namespace System_Drawing;

#import "System.Windows.Forms.tlb"
using namespace System_Windows_Forms;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
```

```
rename("value", "Value")
using namespace OpcLabs_BaseLib;

#import "libid:A0D7CA1E-7D8C-4D31-8ECB-84929E77E331"      // OpcLabs.BaseLibForms
using namespace OpcLabs_BaseLibForms;

// OpcLabs.EasyOpcClassic
#import "libid:1F165598-2F77-41C8-A9F9-EAF00C943F9F" \
    rename("machineName", "MachineName") \
    rename("serverClass", "ServerClass")
using namespace OpcLabs_EasyOpcClassic;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

#import "libid:2C654FA0-6CD6-496D-A64E-CE2D2925F388"      // OpcLabs.EasyOpcForms
using namespace OpcLabs_EasyOpcForms;

namespace _UAEndpointDialog
{
    void ShowDialog::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            //
            _UAEndpointDialogPtr EndpointDialogPtr(__uuidof(UAEndpointDialog));
            //

            EndpointDialogPtr->DiscoveryHost = L"opcua.demo-this.com";

            //
            DialogResult dialogResult = EndpointDialogPtr->>ShowDialog(NULL);

            // Display results
            _tprintf(_T("%d\n"), dialogResult);
            _tprintf(_T("%s\n"), CW2T(EndpointDialogPtr->ApplicationElement-
>ToString));
        }
        // Release all interface pointers BEFORE calling CoUninitialize()
        CoUninitialize();
    }
}
```

## Free Pascal

```
// This example shows how to let the user browse for an OPC-UA server endpoint.

class procedure ShowDialog.Main;
```

```
var
  EndpointDialog: UAEndpointDialog;
begin
  // Instantiate the dialog object
  EndpointDialog := CoUAEndpointDialog.Create;

  EndpointDialog.DiscoveryHost := 'opcua.demo-this.com';

  EndpointDialog.ShowDialog(nil);

  // Display results
  WriteLn(EndpointDialog.ApplicationElement.ToString);
end;
```

## Object Pascal

```
// This example shows how to let the user browse for an OPC-UA server endpoint.

class procedure ShowDialog.Main;
var
  EndpointDialog: TUAEndpointDialog;
begin
  // Instantiate the dialog object
  EndpointDialog := TUAEndpointDialog.Create(nil);

  EndpointDialog.DiscoveryHost := 'opcua.demo-this.com';

  EndpointDialog.ShowDialog(nil);

  // Display results
  WriteLn(EndpointDialog.ApplicationElement.ToString);
end;
```

## Visual Basic (VB 6.)

```
Rem This example shows how to let the user browse for an OPC-UA server endpoint.

Private Sub ShowDialog_Main_Command_Click()
  OutputText = ""

  Dim EndpointDialog As New UAEndpointDialog
  EndpointDialog.DiscoveryHost = "opcua.demo-this.com"

  Dim DialogResult
  DialogResult = EndpointDialog.ShowDialog

  ' Display results
  OutputText = OutputText & DialogResult & vbCrLf
  OutputText = OutputText & EndpointDialog.ApplicationElement & vbCrLf
End Sub
```

## VBScript

```
Rem This example shows how to let the user browse for an OPC-UA server endpoint.

Option Explicit

Dim EndpointDialog: Set EndpointDialog =
CreateObject("OpcLabs.EasyOpc.UA.Forms.Browsing.UAEndpointDialog")
EndpointDialog.DiscoveryHost = "opcua.demo-this.com"
```

```
WScript.Echo EndpointDialog.ShowDialog  
  
' Display results  
WScript.Echo EndpointDialog.ApplicationElement
```

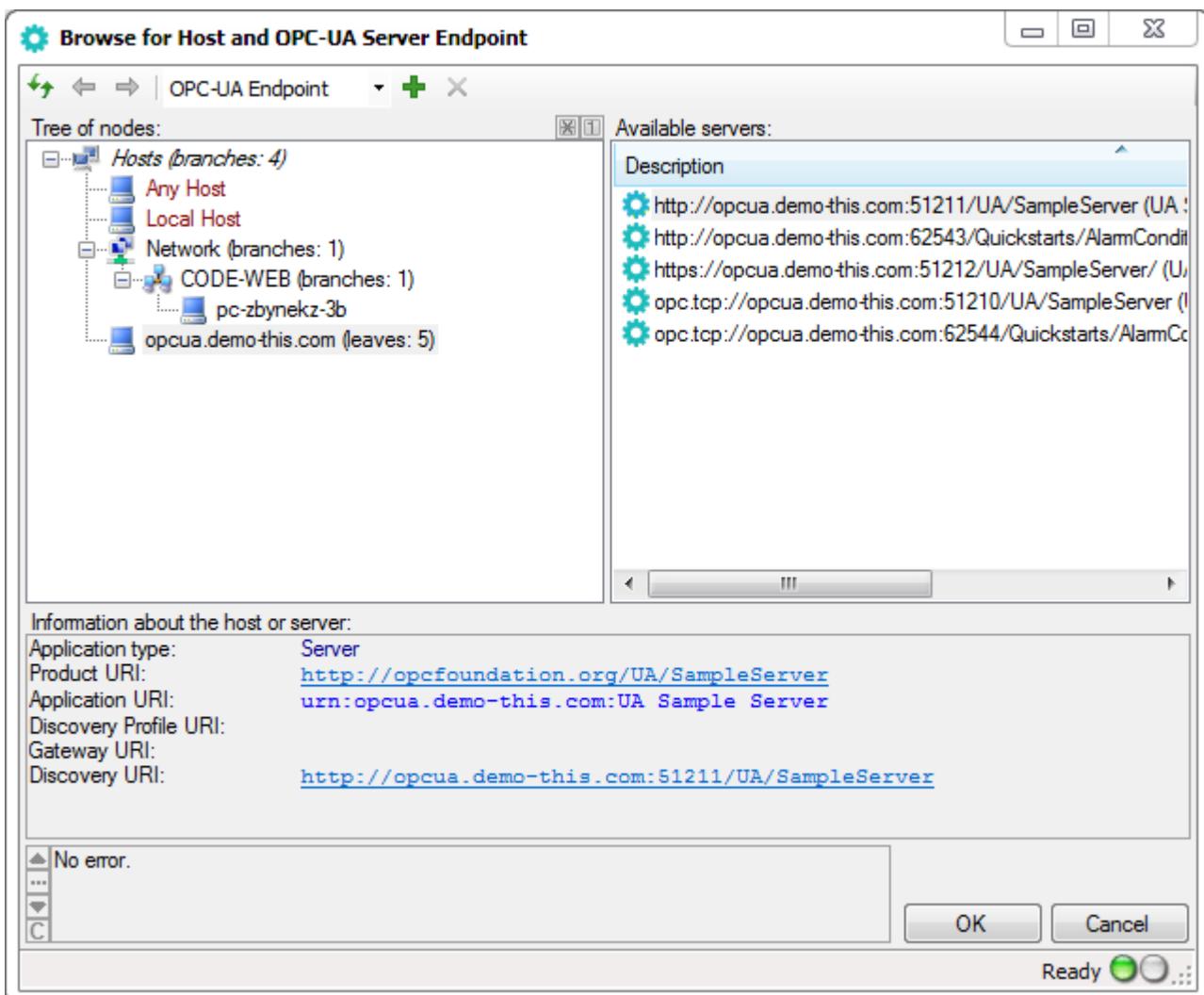
## 10.1.3.2 OPC-UA Host and Endpoint Dialog

Icon: 

With [UAHostAndEndpointDialog](#), your application can integrate a dialog box from which the user can select a host (computer) and an endpoint of OPC-UA server residing on it. This dialog box combines functions of the Computer Browser Dialog and OPC-UA Endpoint Dialog -  into a single dialog.

In addition, the user can add Host nodes in case the particular computer is not visible on the network, or add OPC-UA Endpoint nodes in case the OPC\_UA Server is known to exist but is not returned by OPC-UA discovery feature.

Here is an example of OPC-UA Host and Endpoint dialog in action:



### C++

```
// This example shows how to let the user browse for a host (computer) and an  
// endpoint of an OPC-UA server residing on it.  
  
#include "stdafx.h"
```

```
#include "_UAHostAndEndpointDialog.ShowDialog.h"

// mscorel
using namespace mscorel;

// System.Drawing.tlb
using namespace System_Drawing;

// System.Windows.Forms.tlb
using namespace System_Windows_Forms;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

#import "libid:A0D7CA1E-7D8C-4D31-8ECB-84929E77E331"      // OpcLabs.BaseLibForms
using namespace OpcLabs_BaseLibForms;

// OpcLabs.EasyOpcClassic
#import "libid:1F165598-2F77-41C8-A9F9-EAF00C943F9F" \
    rename("machineName", "MachineName") \
    rename("serverClass", "ServerClass")
using namespace OpcLabs_EasyOpcClassic;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

#import "libid:2C654FA0-6CD6-496D-A64E-CE2D2925F388"      // OpcLabs.EasyOpcForms
using namespace OpcLabs_EasyOpcForms;

namespace _UAHostAndEndpointDialog
{
    void ShowDialog::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            //
            UAHostAndEndpointDialogPtr
            HostAndEndpointDialogPtr(__uuidof(UAHostAndEndpointDialog));
            //
            HostAndEndpointDialogPtr->EndpointDescriptor->Host = L"opcua.demo-
this.com";
            //
            DialogResult dialogResult = HostAndEndpointDialogPtr->>ShowDialog(NULL);
        }
    }
}
```

```
// Display results
    _tprintf(_T("%d\n"), dialogResult);
    _tprintf(_T("HostElement: %s\n"), CW2T(HostAndEndpointDialogPtr-
>HostElement->ToString));
    _tprintf(_T("ApplicationElement: %s\n"), CW2T(HostAndEndpointDialogPtr-
>ApplicationElement->ToString));
}
// Release all interface pointers BEFORE calling CoUninitialize()
CoUninitialize();
}
}
```

## Free Pascal

```
// This example shows how to let the user browse for a host (computer) and
// an endpoint of an OPC-UA server residing on it.

class procedure ShowDialog.Main;
var
    HostAndEndpointDialog: UAHostAndEndpointDialog;
begin
    // Instantiate the dialog object
    HostAndEndpointDialog := CoUAHostAndEndpointDialog.Create;

    HostAndEndpointDialog.EndpointDescriptor.Host := 'opcua.demo-this.com';

    HostAndEndpointDialog.ShowDialog(nil);

    // Display results
    WriteLn('HostElement: ', HostAndEndpointDialog.HostElement.ToString);
    WriteLn('ApplicationElement: ',
    HostAndEndpointDialog.ApplicationElement.ToString);
end;
```

## Object Pascal

```
// This example shows how to let the user browse for a host (computer) and
// an endpoint of an OPC-UA server residing on it.

class procedure ShowDialog.Main;
var
    HostAndEndpointDialog: TUAHostAndEndpointDialog;
begin
    // Instantiate the dialog object
    HostAndEndpointDialog := TUAHostAndEndpointDialog.Create(nil);

    HostAndEndpointDialog.EndpointDescriptor.Host := 'opcua.demo-this.com';

    HostAndEndpointDialog.ShowDialog(nil);

    // Display results
    WriteLn('HostElement: ', HostAndEndpointDialog.HostElement.ToString);
    WriteLn('ApplicationElement: ',
    HostAndEndpointDialog.ApplicationElement.ToString);
end;
```

## Visual Basic (VB 6.)

Rem This example shows how to let the user browse for a host (computer) and an endpoint of an OPC-UA server residing on it.

```
Private Sub ShowDialog_Main_Command_Click()
    OutputText = ""

    Dim HostAndEndpointDialog As New UAHostAndEndpointDialog
    HostAndEndpointDialog.EndpointDescriptor.Host = "opcua.demo-this.com"

    Dim DialogResult
    DialogResult = HostAndEndpointDialog.ShowDialog

    ' Display results
    OutputText = OutputText & DialogResult & vbCrLf
    OutputText = OutputText & "HostElement: " & HostAndEndpointDialog.HostElement &
    vbCrLf
    OutputText = OutputText & "ApplicationElement: " &
    HostAndEndpointDialog.ApplicationElement & vbCrLf
End Sub
```

## VBScript

---

Rem This example shows how to let the user browse for a host (computer) and an endpoint of an OPC-UA server residing on it.

```
Option Explicit

Dim HostAndEndpointDialog: Set HostAndEndpointDialog =
CreateObject("OpcLabs.EasyOpc.UA.Forms.Browsing.UAHostAndEndpointDialog")
HostAndEndpointDialog.EndpointDescriptor.Host = "opcua.demo-this.com"

WScript.Echo HostAndEndpointDialog.ShowDialog

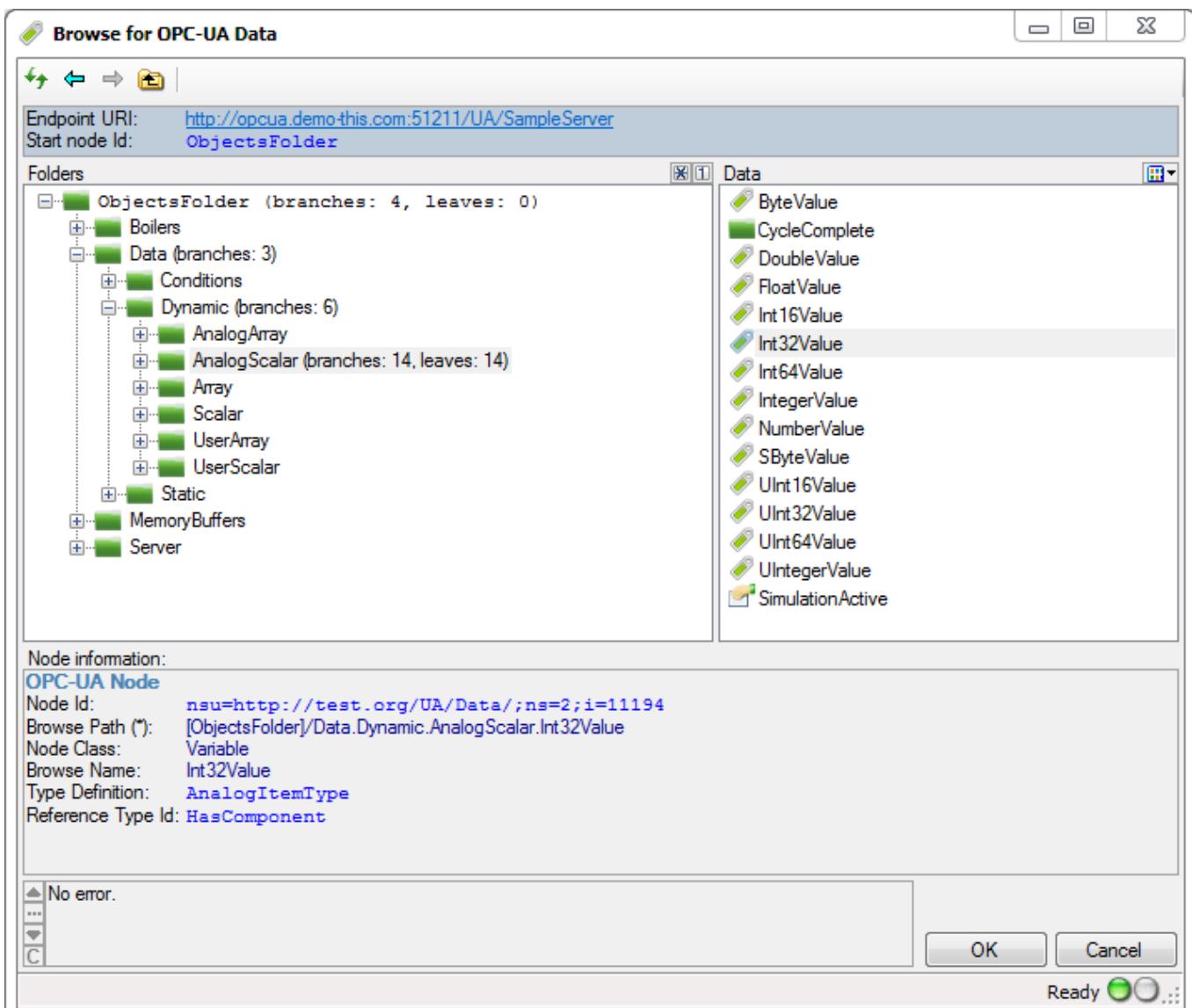
' Display results
WScript.Echo "HostElement: " & HostAndEndpointDialog.HostElement
WScript.Echo "ApplicationElement: " & HostAndEndpointDialog.ApplicationElement
```

### 10.1.3.3 OPC-UA Data Dialog

Icon: 

The OPC-UA Data Dialog ([UADialog](#) class) allows the user to interactively select the OPC data node residing in a specific OPC Unified Architecture server. It also has a different mode (controlled by the UserPickEndpoint property) which allows the user to start the node selection by choosing the host and server endpoint first.

Here is an example of OPC-UA Data dialog in action:



In the default mode (when `UserPickEndpoint` is false), your code should set the `EndpointDescriptor` property to specify the OPC Unified Architecture server whose nodes are to be browsed. If you want the user to pick the endpoint, set the `UserPickEndpoint` property to true; in this case, setting the `EndpointDescriptor` is not needed on input.

After setting the inputs as needed, your code calls the `ShowDialog` method. If the result is equal to `DialogResult.OK`, the user has selected the OPC data node, and information about it can be retrieved from the `NodeElement` (and also `NodeDescriptor`) property. If `UserPickEndpoint` is true, the chosen server endpoint can be retrieved from the `EndpointDescriptor` property.

The user can browse through the UA Objects, and select from Data Variables or Properties.

## C++

```
// This example shows how to let the user browse for an OPC-UA data node (a Data
Variable or a Property).

#include "stdafx.h"
#include "_UADialog.ShowDialog.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorelib
```

```
using namespace mscorelib;

#import "System.Drawing.tlb"
using namespace System_Drawing;

#import "System.Windows.Forms.tlb"
using namespace System_Windows_Forms;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

#import "libid:A0D7CA1E-7D8C-4D31-8ECB-84929E77E331"      // OpcLabs.BaseLibForms
using namespace OpcLabs_BaseLibForms;

// OpcLabs.EasyOpcClassic
#import "libid:1F165598-2F77-41C8-A9F9-EAF00C943F9F" \
    rename("machineName", "MachineName") \
    rename("serverClass", "ServerClass")
using namespace OpcLabs_EasyOpcClassic;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

#import "libid:2C654FA0-6CD6-496D-A64E-CE2D2925F388"      // OpcLabs.EasyOpcForms
using namespace OpcLabs_EasyOpcForms;

namespace _UADialog
{
    void ShowDialog::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {

            // 
            _UADialogPtr DataDialogPtr(__uuidof(UADialog));

            //
            DataDialogPtr->EndpointDescriptor->UrlString = L"http://opcua.demo-
this.com:51211/UA/SampleServer";
            DataDialogPtr->UserPickEndpoint = true;

            //
            DialogResult dialogResult = DataDialogPtr->>ShowDialog(NULL);

            // Display results
            _tprintf(_T("%d\n"), dialogResult);
            _tprintf(_T("EndpointDescriptor: %s\n"), CW2T(DataDialogPtr-
>EndpointDescriptor->ToString));
        }
    }
}
```

```
        _tprintf(_T("NodeElement: %s\n"), CW2T(DataDialogPtr->NodeElement->ToString));
    }
    // Release all interface pointers BEFORE calling CoUninitialize()
    CoUninitialize();
}
}
```

## Free Pascal

```
// This example shows how to let the user browse for an OPC-UA data node
// (a Data Variable or a Property).

class procedure ShowDialog.Main;
var
  DataDialog: UADataDialog;
begin
  // Instantiate the dialog object
  DataDialog := CoUADataDialog.Create;

  DataDialog.EndpointDescriptor.UrlString := 'http://opcua.demo-
this.com:51211/UA/SampleServer';
  DataDialog.UserPickEndpoint := True;

  DataDialog.ShowDialog(nil);

  // Display results
  WriteLn('EndpointDescriptor: ', DataDialog.EndpointDescriptor.ToString);
  WriteLn('NodeElement: ', DataDialog.NodeElement.ToString);
end;
```

## Object Pascal

```
// This example shows how to let the user browse for an OPC-UA data node
// (a Data Variable or a Property).

class procedure ShowDialog.Main;
var
  DataDialog: TUADialog;
begin
  // Instantiate the dialog object
  DataDialog := TUADialog.Create(nil);

  DataDialog.EndpointDescriptor.UrlString := 'http://opcua.demo-
this.com:51211/UA/SampleServer';
  DataDialog.UserPickEndpoint := True;

  DataDialog.ShowDialog(nil);

  // Display results
  WriteLn('EndpointDescriptor: ', DataDialog.EndpointDescriptor.ToString);
  WriteLn('NodeElement: ', DataDialog.NodeElement.ToString);
end;
```

## Visual Basic (VB 6.)

Rem This example shows how to let the user browse for an OPC-UA data node (a Data Variable or a Property).

```
Private Sub ShowDialog_Main_Command_Click()
  OutputText = ""
```

```
Dim DataDialog As New UADataDialog
DataDialog.EndpointDescriptor.UrlString = "opc.tcp://opcua.demo-
this.com:51210/UA/SampleServer"
DataDialog.UserPickEndpoint = True

Dim DialogResult
DialogResult = DataDialog.ShowDialog

' Display results
OutputText = OutputText & DialogResult & vbCrLf
OutputText = OutputText & "EndpointDescriptor: " & DataDialog.EndpointDescriptor
& vbCrLf
OutputText = OutputText & "NodeElement: " & DataDialog.NodeElement & vbCrLf
End Sub
```

## VBScript

Rem This example shows how to let the user browse for an OPC-UA data node (a Data Variable or a Property).

```
Option Explicit

Dim DataDialog: Set DataDialog =
CreateObject("OpcLabs.EasyOpc.UA.Forms.Browsing.UADataDialog")
DataDialog.EndpointDescriptor.UrlString = "http://opcua.demo-
this.com:51211/UA/SampleServer"
DataDialog.UserPickEndpoint = True

WScript.Echo DataDialog.ShowDialog

' Display results
WScript.Echo "EndpointDescriptor: " & DataDialog.EndpointDescriptor
WScript.Echo "NodeElement: " & DataDialog.NodeElement
```

## Multi-selection

When you set the [MultiSelect](#) property of the [UADataDialog](#) to true, the dialog will allow the user to select any number of OPC-UA nodes. The list below the folders and data nodes (labeled "Selected nodes") contains the set of nodes that the user has selected in the dialog. The user can freely add nodes to this list, or remove them. The selected set is carried over to next invocation of the dialog, unless you change it from the code.

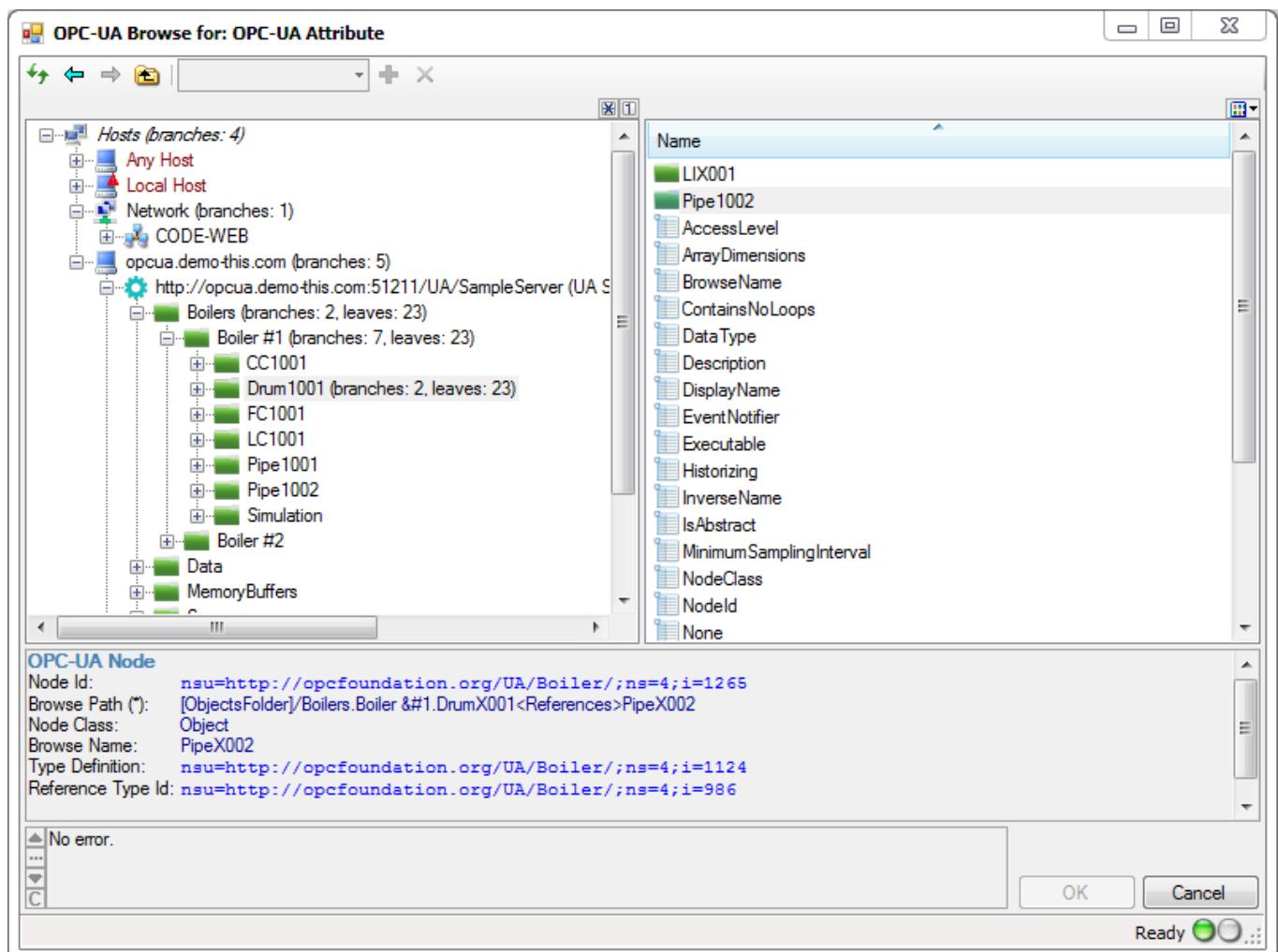
In the multi-selection mode, the set of nodes selected on input (if any) is in the [NodeDescriptors](#) property. On the output, the dialog fills the information about selected nodes into the [NodeElements](#) property (and updated the [NodeDescriptors](#) property as well). If [UserPickEndpoint](#) is true, each node may come from a different server. In this case, the [EndpointDescriptors](#) array contains the server endpoints for each node in [NodeDescriptors](#), with corresponding indices in both arrays.

### 10.1.3.4 Generic OPC-UA Browsing Dialog

Icon:

With [UABrowseDialog](#), your application can integrate a dialog with various OPC-UA elements from which the user can select. This dialog can be configured to serve many different purposes.

Here is an example of the generic OPC-UA browsing dialog in action:

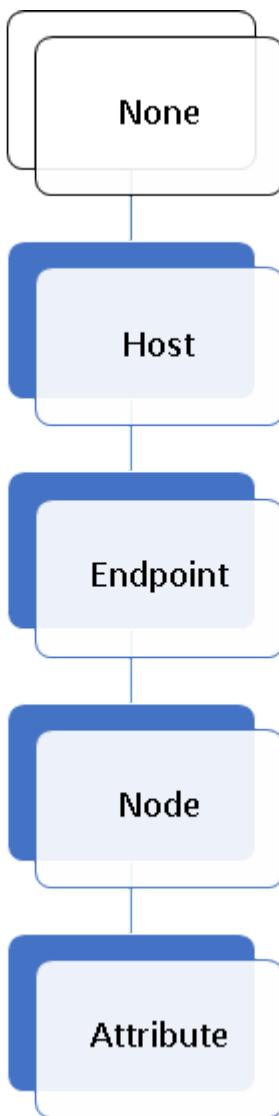


The way the dialog operates is controlled by two main properties:

- **Mode.AnchorElementType** determines the part of the selection that is given on input and cannot be changed by the user.
- **Mode.SelectElementType** determines the type of the element that the user needs to select in order to finalize the dialog.

Values of these properties can be selected from the **UAEElementType** enumeration, which has members for various types of elements that you encounter when working with OPC-UA.

The following chart shows a hierarchy of element types that you can choose from:



For example, let' say that you set [Mode.AnchorElementType](#) to [Endpoint](#), and [Mode.SelectElementType](#) to [Attribute](#). This will cause the dialog to allow the user to browse for an OPC-UA Node on a given server, and then for an OPC-UA attribute on that node.

In this case, before you run the dialog, you need to provide it with values for the [InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor](#) property, because those define your “anchor” element ([Endpoint](#)) that the user cannot change. The dialog will only allow the user to finalize it (besides cancelling) after an OPC-UA attribute is fully selected, because that is your [Mode.SelectElementType](#). After the dialog is successfully finalized, the information about the user’s choice will be available in the [Outputs.CurrentNodeElement.NodeElement](#) and [InputsOutputs.CurrentNodeDescriptor.AttributeId](#) properties.

Note that in addition to the “minimal” scenario described above, you can also pre-set the initial node or attribute, using the [InputsOutputs.CurrentNodeDescriptor.NodeDescriptor](#) or [InputsOutputs.CurrentNodeDescriptor.AttributeId](#) properties, and after the selection is made, these properties will be updated to the new selection as well. This way, if you run the dialog again with the same value, the initial selection will be where the user has left it the last time the dialog was run.

Obviously, the chosen [Mode.SelectElementType](#) must be a child or indirect ancestor of chosen [Mode.AnchorElementType](#) in the hierarchy.

## C++

---

```
// This example shows how to let the user browse for an OPC-UA node.
```

```
#include "stdafx.h"
#include "_UABrowseDialog.ShowDialog.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"      // mscorel
using namespace mscorel;

#import "System.Drawing.tlb"
using namespace System_Drawing;

#import "System.Windows.Forms.tlb"
using namespace System_Windows_Forms;

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

#import "libid:A0D7CA1E-7D8C-4D31-8ECB-84929E77E331"      // OpcLabs.BaseLibForms
using namespace OpcLabs_BaseLibForms;

// OpcLabs.EasyOpcClassic
#import "libid:1F165598-2F77-41C8-A9F9-EAF00C943F9F" \
    rename("machineName", "MachineName") \
    rename("serverClass", "ServerClass")
using namespace OpcLabs_EasyOpcClassic;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

#import "libid:2C654FA0-6CD6-496D-A64E-CE2D2925F388"      // OpcLabs.EasyOpcForms
using namespace OpcLabs_EasyOpcForms;

namespace _UABrowseDialog
{
    void ShowDialog::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {

            // 
            _UABrowseDialogPtr BrowseDialogPtr(__uuidof(UABrowseDialog));
            //

            BrowseDialogPtr->InputsOutputs->CurrentNodeDescriptor-
>EndpointDescriptor->Host = L"opcua.demo-this.com";
            BrowseDialogPtr->Mode->AnchorElementType = UAEElementType_Host;
        }
    }
}
```

```
//  
DialogResult dialogResult = BrowseDialogPtr->ShowDialog(NULL);  
  
// Display results  
_tprintf(_T("%d\n"), dialogResult);  
_tprintf(_T("%s\n"), CW2T(BrowseDialogPtr->Outputs->CurrentNodeElement->NodeElement->ToString));  
}  
// Release all interface pointers BEFORE calling CoUninitialize()  
CoUninitialize();  
}  
}  
}
```

## Free Pascal

```
// This example shows how to let the user browse for an OPC-UA node.  
  
class procedure ShowDialog.Main;  
var  
  BrowseDialog: UABrowseDialog;  
begin  
  // Instantiate the dialog object  
  BrowseDialog := CoUABrowseDialog.Create;  
  BrowseDialog.InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor.Host :=  
'opcua.demo-this.com';  
  BrowseDialog.Mode.AnchorElementType := UAEElementType_Host;  
  
  BrowseDialog.ShowDialog(nil);  
  
  // Display results  
  WriteLn(BrowseDialog.Outputs.CurrentNodeElement.NodeElement.ToString);  
end;
```

## Object Pascal

```
// This example shows how to let the user browse for an OPC-UA node.  
  
class procedure ShowDialog.Main;  
var  
  BrowseDialog: TUABrowseDialog;  
begin  
  // Instantiate the dialog object  
  BrowseDialog := TUABrowseDialog.Create(nil);  
  BrowseDialog.InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor.Host :=  
'opcua.demo-this.com';  
  BrowseDialog.Mode.AnchorElementType := UAEElementType_Host;  
  
  BrowseDialog.ShowDialog(nil);  
  
  // Display results  
  WriteLn(BrowseDialog.Outputs.CurrentNodeElement.NodeElement.ToString);  
end;
```

## PHP

```
// This example shows how to let the user browse for an OPC-UA node.  
  
$UAEElementType_Host = 1;  
  
$BrowseDialog = new COM("OpcLabs.EasyOpc.UA.Forms.Browsing.UABrowseDialog");  
$BrowseDialog->InputsOutputs->CurrentNodeDescriptor->EndpointDescriptor->Host =
```

```
"opcua.demo-this.com";
$BrowseDialog->Mode->AnchorElementType = $UAElementType_Host;

printf("%d\n", $BrowseDialog->ShowDialog);

// Display results
printf("%s\n", $BrowseDialog->Outputs->CurrentNodeElement->NodeElement);
```

## Python

---

```
# This example shows how to let the user browse for an OPC-UA node.

import win32com.client

UAElementType_Host = 1

browseDialog =
win32com.client.Dispatch('OpcLabs.EasyOpc.UA.Forms.Browsing.UABrowseDialog')
browseDialog.InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor.Host =
"opcua.demo-this.com"
browseDialog.Mode.AnchorElementType = UAElementType_Host

print(browseDialog.ShowDialog())

# Display results
print(browseDialog.Outputs.CurrentNodeElement.NodeElement)
```

## Visual Basic (VB 6.)

---

```
Rem This example shows how to let the user browse for an OPC-UA node.

Private Sub ShowDialog_Main_Command_Click()
    OutputText = ""

    Dim BrowseDialog As New UABrowseDialog
    BrowseDialog.InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor.Host =
"opcua.demo-this.com"

    Dim DialogResult
    DialogResult = BrowseDialog.ShowDialog

    ' Display results
    OutputText = OutputText & DialogResult & vbCrLf
    OutputText = OutputText & BrowseDialog.Outputs.CurrentNodeElement.NodeElement &
vbCrLf
End Sub
```

## VBScript

---

```
Rem This example shows how to let the user browse for an OPC-UA node.

Option Explicit

Const UAElementType_Host = 1

Dim BrowseDialog: Set BrowseDialog =
CreateObject("OpcLabs.EasyOpc.UA.Forms.Browsing.UABrowseDialog")
BrowseDialog.InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor.Host =
"opcua.demo-this.com"
BrowseDialog.Mode.AnchorElementType = UAElementType_Host
```

```
WScript.Echo BrowseDialog.ShowDialog  
  
' Display results  
WScript.Echo BrowseDialog.Outputs.CurrentNodeElement.NodeElement
```

## Multi-selection

It is also possible to configure the dialog for a multi-selection. In this mode, the user can select zero, one, or more nodes. In order to enable the multi-select mode, set the [Mode.MultiSelect](#) property to true. In the multi-select mode, the initial set of the selected nodes (when the dialog is first displayed to the user) is given by the contents of the [InputOutputs.SelectionDescriptors](#) collection. When the user makes the selection and accepts it by closing the dialog, this collection is updated, and also, all information about the selected nodes is placed to the [Outputs.SelectionElements](#) collection.

### VBScript

```
Rem This example shows how to let the user browse for multiple OPC-UA nodes.  
  
Option Explicit  
  
Const UAEElementType_Host = 1  
  
Dim BrowseDialog: Set BrowseDialog =  
CreateObject("OpcLabs.EasyOpc.UA.Forms.Browsing.UABrowseDialog")  
BrowseDialog.InputsOutputs.CurrentNodeDescriptor.EndpointDescriptor.Host =  
"opcua.demo-this.com"  
BrowseDialog.Mode.AnchorElementType = UAEElementType_Host  
BrowseDialog.Mode.MultiSelect = True  
  
WScript.Echo BrowseDialog.ShowDialog  
  
' Display results  
Dim SelectionElements: Set SelectionElements =  
BrowseDialog.Outputs.SelectionElements  
Dim i: For i = 0 To SelectionElements.Count - 1  
    Dim Element: Set Element = SelectionElements(i)  
    WScript.Echo "SelectionElements(" & i & ") : " & Element.NodeElement  
Next
```

## Other settings

There are also ways to control some finer aspects of the dialog. For example, the [Mode.ShowListBranches](#) property (defaults to true) controls whether the branches of the tree are also displayed in the list view.

When you set the [Simulated](#) property of the dialog to true, the dialog will provide its contents from a pre-defined, simulated view of the world, with fake networks, computers, OPC servers, and their contents. This can be useful for experimentation and testing, either by the developer during the design (right in Visual Studio), or by the end-user (if you expose this functionality in your application), when the environment is not accessible.

## 10.2 OPC Controls

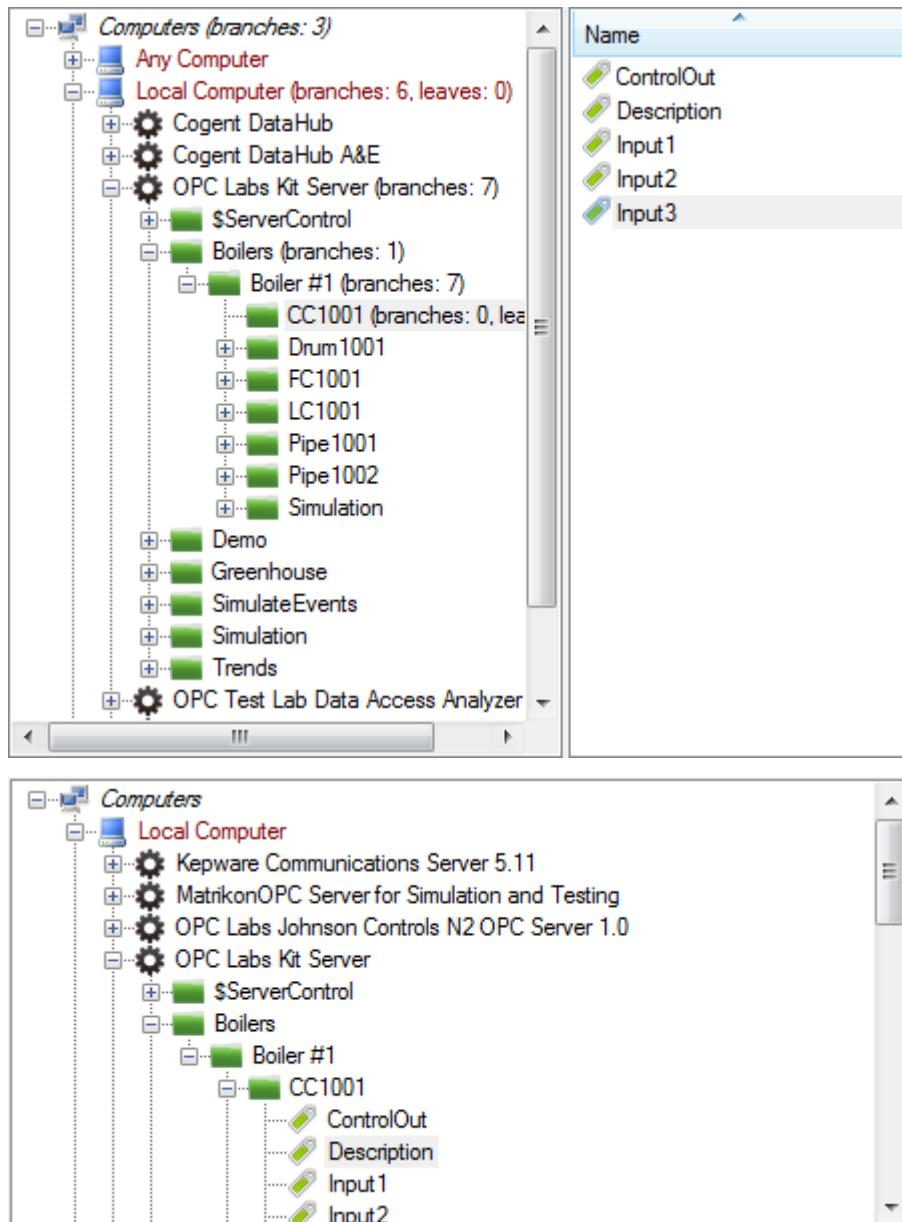
### 10.2.1 OPC Classic Controls

QuickOPC.NET contains OPC-enabled .NET controls that can be placed onto your Windows Forms and configured and programmed to cooperate with other controls.

## 10.2.1.1 Generic OPC Classic Browsing Control

With [OpcBrowseControl](#), your forms can integrate a control with various OPC nodes from which the user can select. This control can be configured to serve many different purposes.

Here are examples of the generic OPC browsing control in action:



The functionality and properties of the [OpcBrowseControl](#) are similar to that of [OpcBrowseDialog](#), described earlier in this text. Please refer to the documentation of [OpcBrowseDialog](#) for details. Here are the major differences:

- There is no toolbar, no filter controls, no node information box, no error box, and no status bar. Events are provided (described further below) that allow you to implement similar functionality yourself, if needed.
- Changes you make in run-time to relevant values of the [Inputs](#) and [InputsOutputs](#) properties are immediately reflected in the control. Analogically, actions by the user that change the current node or the selection set are immediately projected into the [InputsOutputs](#) and [Outputs](#) properties.

- When multi-selection mode is enabled, the nodes are always selected directly in the list view, and not in a separate list. For this reason, the nodes in a multi-selection must all belong under a single parent branch.
- If only a tree view is displayed, the tree will also contain the leaves.
- If only a list view is displayed, the list will also always contain the branches, regardless of the [Mode.ShowListBranches](#) property value.
- When a node is selected in the tree view, a first node in the list view is not automatically focused.
- In the design-time mode, the control always shows simulated data, even if the [Simulated](#) property is set to `false`.

Using the [Kind](#) property, the browsing control can be configured to provide a tree view only ([BrowseControlKinds.Tree](#)), a list view only ([BrowseControlKinds.List](#)), or a combined tree view and list view ([BrowseControlKinds.TreeAndList](#); this is the default).

The [View](#) property (of [System.Windows.Forms.View](#) enumeration type) controls how the list view items are displayed. Possible values are [LargeIcon](#), [Details](#) (the default), [SmallIcon](#), [List](#), or [Tile](#).

In order to achieve tight integration with other controls on your form, you can hook to events that the browsing control provides.

The [CurrentNodeChanged](#) event occurs when the current node changes. You can obtain the information about the new current node from [InputsOutputs.CurrentNodeDescriptor](#) and [Outputs.CurrentNodeElement](#) properties inside the event handler.

The [SelectionChanged](#) event is meant for multi-selection mode, and occurs when the selection set changes. You can obtain the information about the new selection set from [InputsOutputs.SelectionDescriptors](#) and [Outputs.SelectionElements](#) collections inside the event handler.

The [NodeDoubleClick](#) event occurs when a node is double-clicked. This is the current node, and therefore the information about it can be obtained in the same way as in the [CurrentNodeChanged](#) event handler, described above.

The [BrowseFailure](#) event indicates that an exception has occurred during browsing. The information about the exception is contained in the event arguments.

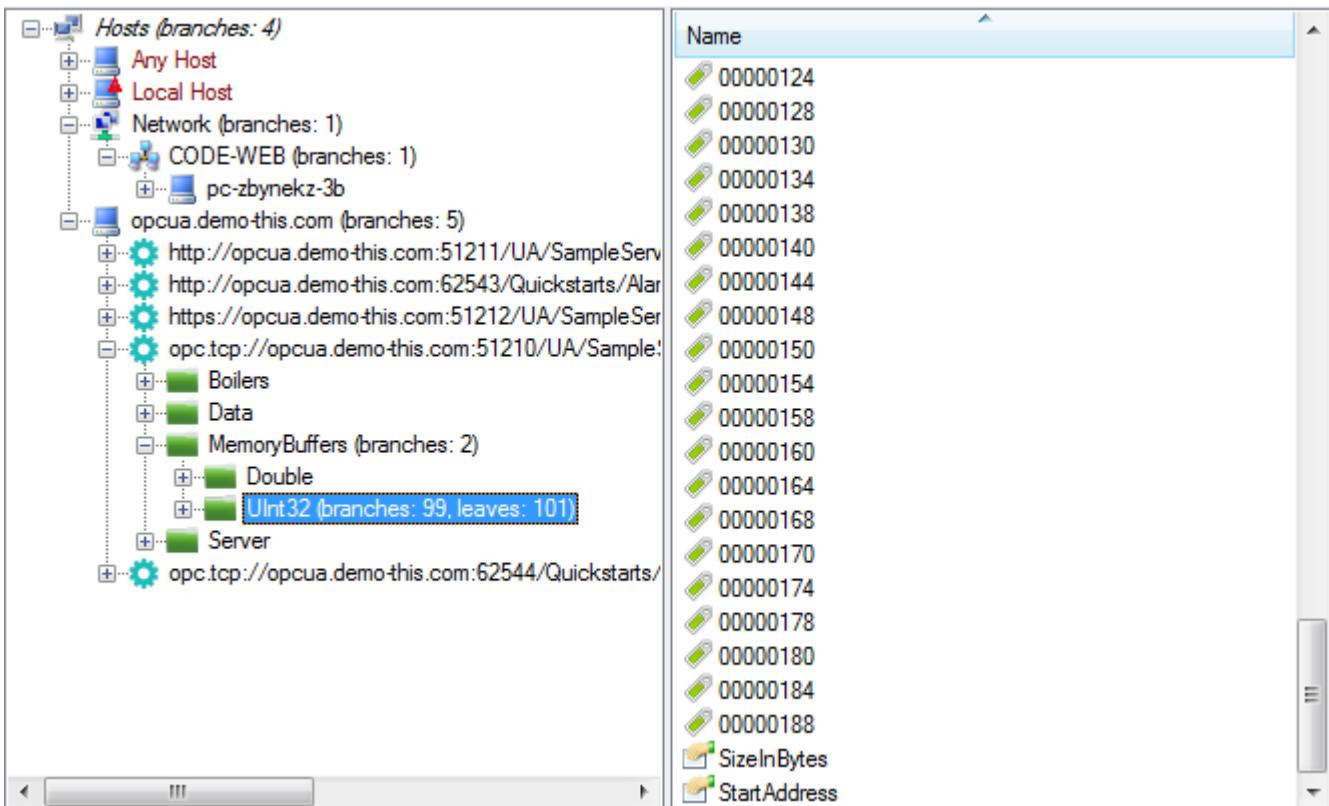
## 10.2.2 OPC-UA Controls

QuickOPC-UA contains OPC-enabled .NET controls that can be placed onto your Windows Forms and configured and programmed to cooperate with other controls.

### 10.2.2.1 Generic OPC UA Browsing Control

With [UABrowseControl](#), your forms can integrate a control with various OPC nodes from which the user can select. This control can be configured to serve many different purposes.

Here is an example of the generic OPC browsing control in action:



The functionality and properties of the [UABrowseControl](#) are similar to that of [UABrowseControl](#), described earlier in this text. Please refer to the documentation of [UABrowseControl](#) for details. Here are the major differences:

- There is no toolbar, no filter controls, no node information box, no error box, and no status bar. Events are provided (described further below) that allow you to implement similar functionality yourself, if needed.
- Changes you make in run-time to relevant values of the [Inputs](#) and [InputsOutputs](#) properties are immediately reflected in the control. Analogically, actions by the user that change the current node or the selection set are immediately projected into the [InputsOutputs](#) and [Outputs](#) properties.
- When multi-selection mode is enabled, the nodes are always selected directly in the list view, and not in a separate list. For this reason, the nodes in a multi-selection must all belong under a single parent branch.
- If only a tree view is displayed, the tree will also contain the leaves.
- If only a list view is displayed, the list will also always contain the branches, regardless of the [Mode.ShowListBranches](#) property value.
- When a node is selected in the tree view, a first node in the list view is not automatically focused.
- In the design-time mode, the control always shows simulated data, even if the [Simulated](#) property is set to false.

Using the [Kind](#) property, the browsing control can be configured to provide a tree view only ([BrowseControlKinds.Tree](#)), a list view only ([BrowseControlKinds.List](#)), or a combined tree view and list view ([BrowseControlKinds.TreeAndList](#); this is the default).

The [View](#) property (of [System.Windows.Forms.View](#) enumeration type) controls how the list view items are displayed. Possible values are [LargeIcon](#), [Details](#) (the default), [SmallIcon](#), [List](#), or [Tile](#).

In order to achieve tight integration with other controls on your form, you can hook to events that the browsing control provides.

The [CurrentNodeChanged](#) event occurs when the current node changes. You can obtain the information about the new current node from [InputsOutputs.CurrentNodeDescriptor](#) and [Outputs.CurrentNodeElement](#) properties inside the event handler.

The [SelectionChanged](#) event is meant for multi-selection mode, and occurs when the selection set changes. You can obtain the information about the new selection set from [InputsOutputs.SelectionDescriptors](#)

and `Outputs.SelectionElements` collections inside the event handler.

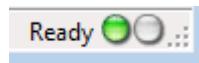
The `NodeDoubleClick` event occurs when a node is double-clicked. This is the current node, and therefore the information about it can be obtained in the same way as in the `CurrentNodeChanged` event handler, described above.

The `BrowseFailure` event indicates that an exception has occurred during browsing. The information about the exception is contained in the event arguments.

## 10.3 Common Functionality in Browsing Dialogs and Controls

The tree view in the browsing dialogs displays the number of node sub-branches and sub-leaves (when known) in the parentheses next to the node name.

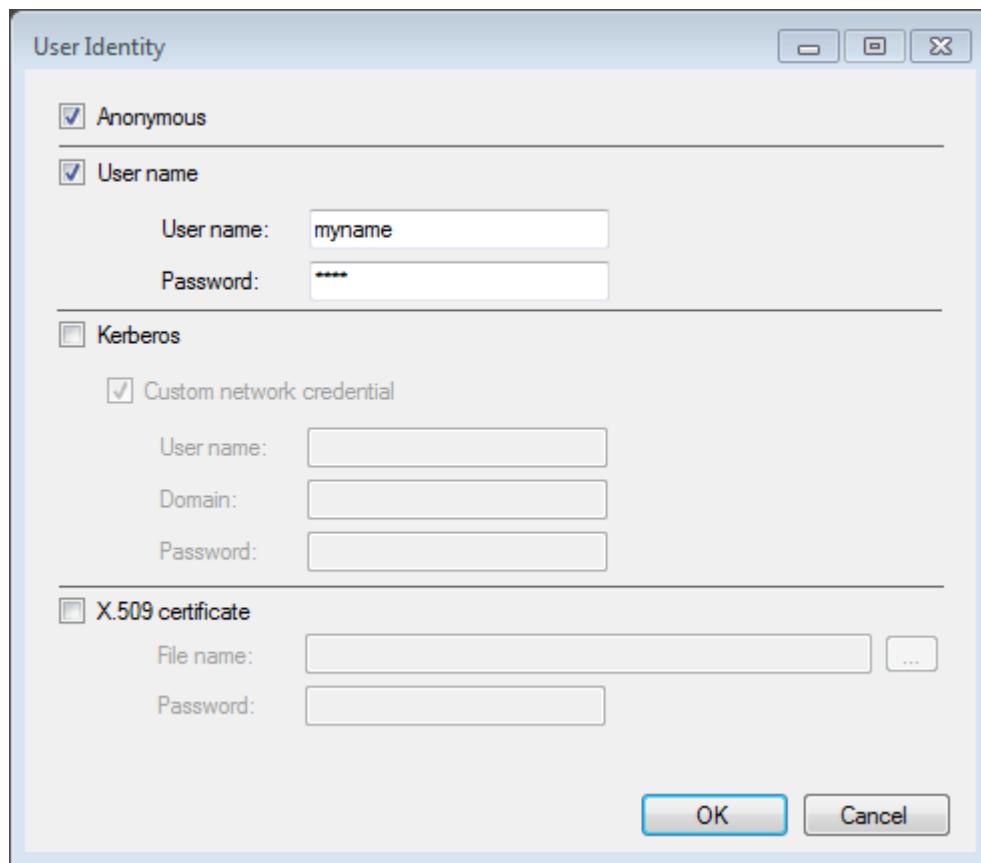
The browsing dialogs also have a "traffic light" status indicator in the lower right corner, allowing for quick recognition of the status by the user.



The hierarchical browsing dialogs have Back and Forward buttons, allowing better navigation experience in some scenarios. The buttons behave similarly to those commonly found in Internet browsers.



In OPC-UA browsing dialogs and controls, the context menu (right-click) on an OPC UA endpoint node provides a "User Identity" command, allowing the user to specify the identity (or identities) that should be used during the connection attempt to the server.



On the host and computer nodes, the user can invoke a "Ping Host Test" or a "Ping Computer Test" command from the node's context menu, in order to quickly verify the basic connectivity to the selected host or computer. These

commands are also available on OPC UA endpoint nodes.

On the OPC UA Endpoint nodes, the user can invoke a "TCP Connect Test" command from the node's context menu.. This command attempts to open a TCP socket to the selected endpoint (without any OPC UA-specific actions), in order to verify the network connectivity and the presence of the server.

## 11 EasyOPC Extensions for .NET

EasyOPC Extensions for .NET is a pack of .NET classes that adds more functionality to the EasyOPC.NET and EasyOPC-UA components. It is built upon and relies on EasyOPC.NET and EasyOPC-UA, 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.

### 11.1 Usage

A significant part of the EasyOPC Extensions for .NET 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, the [EasyDAClient](#) class appears 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 [IEasyDAClientExtension2](#) 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 [IEasyDAClientExtension2.GetDataTypePropertyValue](#) instead, and pass it the [EasyDAClient](#) object as the first argument.

#### 11.1.1 Generic Types

Some methods (described further below) that provide generic access return values (or arrays of values) of type [ValueResult<T>](#). [ValueResult<T>](#) is a type derived from [ValueResult](#) that holds returns with values of type T. Attempt to store a value of incompatible type causes an exception. This means that you specify the type of the value you expect from OPC, and safely assume that the value of given type will be returned.

Further generic types exist for various OPC specifications. See e.g. Generic Types for OPC Data Access.

### 11.2 Data Access Extensions

This chapter describes EasyOPC.NET extensions for OPC Data Access.

#### 11.2.1 Generic Types for OPC-DA

Following generic types are available for use with OPC Data Access:

- [DAVtq<T>](#): Holds typed data value (of type T), timestamp, and quality.
- [DAVtqResult<T>](#): Holds result of an operation in form of a typed [DAVtq<T>](#) (value, timestamp, quality).
- [DALitemValueArguments<T>](#): Arguments for an operation on an OPC item. Carries a value of type T.
- [DALitemVtqArguments<T>](#): Arguments for an operation on an OPC item. Carries a [DAVtq<T>](#) object.
- [EasyDALitemSubscriptionArguments<T>](#): Arguments for subscribing to an OPC item of type T.
- [EasyDALitemChangedEventArgs<T>](#): Data of an event or callback for a significant change in OPC item of type T.

These types are used with generic access extensions for OPC items (see Generic Access) and with generic access extensions for OPC properties (see Generic Access). They are also use with the Live Mapping Model.

For example, you can call the [EasyDAClient.ReadItem<T>](#) extension method with a given type T, and receive back a [DAVtq<T>](#) object with a [TypedValue](#) already properly typed as T.

Noted that if the untyped property on the base (non-generic) type has a name like **XXXX**, the corresponding typed

property on the generic type has a name like `TypedXXXX`. For example, for an untyped property `DAVtqResult.Vtq`, there is a corresponding typed property `DAVtqResult<T>.TypedVtq`.

## 11.2.2 Extensions for OPC Items

### 11.2.2.1 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.WritelItemValueInt32`. There is such a method for each primitive type, named `WritelItemValueXXXX`, 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 `WritelItemValueArrayOfXXXX`, where **XXXX** is the name of the element type. For example, `EasyDAClient.WritelItemValueArrayOfInt32` will write into an item as an array of 32-bit signed integers.

### 11.2.2.2 Generic Access

The `EasyDAClient.ReadItem<T>` extension method reads an OPC item of a given type `T`, and return the result as `DAVtq<T>` (typed value/timestamp/quality) object.

The `EasyDAClient.ReadItemValue<T>` extension method reads an OPC item of a given type `T`, and return the actual item's value typed as `T`.

The `EasyDAClient.WritelItem<T>` extension method writes a typed value/timestamp/quality into an OPC item, using `DAVtq<T>` object.

The `EasyDAClient.WritelItemValue<T>` extension method writes a typed value of type `T` into an OPC item.

There are several overloads of all the above described methods, with different structure of arguments.

## 11.2.3 Extensions for OPC Properties

### 11.2.3.1 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.

## 11.2.3.2 Generic Access

The `EasyDAClient.GetPropertyValue<T>` extension method gets a value of OPC property of a given type T, and return the actual value typed as T. There are several overloads of this method with different structure of arguments.

## 11.2.3.3 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, `GetDataPropertyTypeValue` method returns a `VarType`, `GetScanRatePropertyValue` method returns a `float`, and `GetDescriptionPropertyValue` method returns a `string`.

### C#

```
// This example shows how to obtain a data type of an OPC item.
using System;
using OpcLabs.BaseLib.ComInterop;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.Extensions;

namespace DocExamples
{
    namespace _EasyDAClientExtension
    {
        class GetDataTypePropertyValue
        {
            public static void Main()
            {
                var easyDAClient = new EasyDAClient();

                // Get the DataType property value, already converted to VarType
                VarType varType = easyDAClient.GetDataTypePropertyValue("", "OPCLabs.KitServer.2", "Simulation.Random");

                // Display the obtained data type
                Console.WriteLine("VarType: {0}", varType); // Display data type
symbolically
            }
        }
    }
}
```

### VB.NET

```
' This example shows how to obtain a data type of an OPC item.
Imports OpcLabs.BaseLib.ComInterop
Imports OpcLabs.EasyOpc.DataAccess
Imports OpcLabs.EasyOpc.DataAccess.Extensions

Namespace _EasyDAClientExtension
    Friend Class GetDataTypePropertyValue
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            ' Get the DataType property value, already converted to VarType
```

```
Dim varType As VarType = easyDAClient.GetDataTypePropertyValue("",  
"OPCLabs.KitServer.2", "Simulation.Random")  
  
    ' Display the obtained data type  
    Console.WriteLine("VarType: {0}", varType) ' Display data type  
symbolically  
End Sub  
End Class  
End Namespace
```

## 11.2.3.4 Alternate Access Methods

The [GetPropertyValueCollection](#) 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).

### C#

```
// This example shows how to obtain a dictionary of OPC property values for an OPC  
item, and extract property values.  
using System;  
using OpcLabs.EasyOpc.DataAccess;  
using OpcLabs.EasyOpc.DataAccess.Extensions;  
  
namespace DocExamples  
{  
    namespace _EasyDAClientExtension  
{  
        class GetPropertyValueDictionary  
{  
            public static void Main()  
{  
                var easyDAClient = new EasyDAClient();  
  
                // Get dictionary of property values, for all well-known properties  
                DAPROPERTYVALUEDICTIONARY propertyValueDictionary =  
                    easyDAClient.GetPropertyValueDictionary("",  
"OPCLabs.KitServer.2", "Simulation.Random");  
  
                // Display some of the obtained property values  
                // The production code should also check for the .Exception first,  
before getting .Value  
  
                Console.WriteLine("propertyValueDictionary[DAPROPERTYID.ACCESSRIGHTS].Value: {0}",  
                    propertyValueDictionary[DAPROPERTYIDS.ACCESSRIGHTS].Value);  
  
                Console.WriteLine("propertyValueDictionary[DAPROPERTYID.DATATYPE].Value: {0}",  
                    propertyValueDictionary[DAPROPERTYIDS.DATATYPE].Value);  
  
                Console.WriteLine("propertyValueDictionary[DAPROPERTYID.TIMESTAMP].Value: {0}",  
                    propertyValueDictionary[DAPROPERTYIDS.TIMESTAMP].Value);  
            }  
        }  
    }  
}
```

## VB.NET

```
' This example shows how to obtain a dictionary of OPC property values for an OPC
item, and extract property values.
Imports OpcLabs.EasyOpc.DataAccess
Imports OpcLabs.EasyOpc.DataAccess.Extensions

Namespace _EasyDAClientExtension
    Friend Class GetPropertyValueDictionary
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            ' Get dictionary of property values, for all well-known properties
            Dim PropertyValueDictionary As DAPropertyValueDictionary =
easyDAClient.GetPropertyValueDictionary("", "OPCLabs.KitServer.2",
"Simulation.Random")

            ' Display some of the obtained property values
            ' The production code should also check for the .Exception first, before
getting .Value

Console.WriteLine("PropertyValueDictionary[DAPropertyId.AccessRights].Value: {0}",
PropertyValueDictionary(DAPropertyIds.AccessRights).Value)
            Console.WriteLine("PropertyValueDictionary[DAPropertyId.DataType].Value:
{0}", PropertyValueDictionary(DAPropertyIds.DataType).Value)

Console.WriteLine("PropertyValueDictionary[DAPropertyId.Timestamp].Value: {0}",
PropertyValueDictionary(DAPropertyIds.Timestamp).Value)
        End Sub
    End Class
End Namespace
```

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.

## C#

```
// This example shows how to obtain a structure containing property values for an
OPC item, and display some property values.
using System;
using OpcLabs.EasyOpc.DataAccess;
using OpcLabs.EasyOpc.DataAccess.Extensions;

namespace DocExamples
{
    namespace _EasyDAClientExtension
    {
        class GetItemPropertyRecord
        {
            public static void Main()
            {
                var easyDAClient = new EasyDAClient();

                // Get a structure containing values of all well-known properties
                DAItemPropertyRecord itemPropertyRecord =
                    easyDAClient.GetItemPropertyRecord("", "OPCLabs.KitServer.2",
"Simulation.Random");
```

```
// Display some of the obtained property values
Console.WriteLine("itemPropertyRecord.AccessRights: {0}",
itemPropertyRecord.AccessRights);
Console.WriteLine("itemPropertyRecord.DataType: {0}",
itemPropertyRecord.DataType);
Console.WriteLine("itemPropertyRecord.Timestamp: {0}",
itemPropertyRecord.Timestamp);
}
}
}
```

## VB.NET

```
' This example shows how to obtain a structure containing property values for an OPC
item, and display some property values.
Imports OpcLabs.EasyOpc.DataAccess
Imports OpcLabs.EasyOpc.DataAccess.Extensions

Namespace _EasyDAClientExtension
    Friend Class GetItemPropertyRecord
        Public Shared Sub Main()
            Dim easyDAClient = New EasyDAClient()

            ' Get a structure containing values of all well-known properties
            Dim itemPropertyRecord As DAIItemPropertyRecord =
easyDAClient.GetItemPropertyRecord("", "OPCLabs.KitServer.2", "Simulation.Random")

            ' Display some of the obtained property values
            Console.WriteLine("itemPropertyRecord.AccessRights: {0}",
itemPropertyRecord.AccessRights)
            Console.WriteLine("itemPropertyRecord.DataType: {0}",
itemPropertyRecord.DataType)
            Console.WriteLine("itemPropertyRecord.Timestamp: {0}",
itemPropertyRecord.Timestamp)
        End Sub
    End Class
End Namespace
```

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.

## 11.3 Alarms&Events Extensions

This chapter describes EasyOPC.NET extensions for OPC Alarms&Events.

### 11.3.1 OPC-A&E Queries

The extension method [EasyAEClient.FindEventCategory](#) attempts to find an event category given by its numerical category ID in the OPC server. If successful, it provides the [AECategoryElement](#) filled in with details about the event category.

The extension method [EasyAEClient.FindEventCondition](#) attempts to find an event condition in the OPC server, given the numerical category ID, and event condition name. If successful, it provides the [AEConditionElement](#) filled in with details about the event condition.

## 11.4 Unified Architecture Extensions

This chapter describes EasyOPC extensions for OPC Unified Architecture (OPC-UA).

### 11.4.1 Extensions on Helper Types

The [UAApplicationTypes.IsServer](#) extension method allows you to determine whether a UA application (as returned e.g. by server discovery) includes OPC-UA server functionality. This is a shortcut for testing whether the application type is either [Server](#) or [ClientOrServer](#).

The extensions on the [UANodeElement](#) (as returned e.g. by browsing for OPC-UA nodes) allow finer categorization of the nodes over what is provided just by the [NodeClass](#) property. For example, you can test whether a node is a Data Variable or a Property, by extensions methods [IsDataVariable](#) and [IsProperty](#).

## 11.5 Software Toolbox Extender Replacement

### 11.5.1 Introduction

QuickOPC.NET can serve as a replacement for Software Toolbox Extender ([www.opcextender.net](http://www.opcextender.net)) component. The related types are available under the [OpcLabs.EasyOpc.SwtbExtenderReplacement](#) namespace.

### 11.5.2 Details

Only a subset of Extender's functionality is available through the replacement. It is aimed at lower-level coding, with no support for data binding. If your application with Software Toolbox Extender uses data binding, it should be rewritten using the Live Binding features of QuickOPC.NET, but there is no direct compatibility.

Following public classes, interfaces and other types are contained in the replacement:

- Common
- ISwtbOPCDABasic
- ITagVTQ
- OPCDataNetEngine (replaces [OPCEngine](#) or its derivatives)
- TagVTQ

Some limitations, described below, exist for usage of these classes:

- Methods using the [IPAddress](#) class are not available.
- The [NetworkToSearch](#) argument of [BrowseComputers](#) method of [ISwtbOPCDABasic](#) is ignored.
- The [IsConnected](#) method, and the [ConnectionChanged](#) event of [ISwtbOPCDABasic](#), is not available.
- The [FormattedValue](#) method of [ITagVTQ](#) is not available.
- The [ReadAsync](#) and [WriteAsync](#) methods of [OPCDataNetEngine](#) ([ISwtbOPCDABasic](#)) perform asynchronous OPC reads and writes. The methods work, however, synchronously with respect to the calling program: they will block the caller and perform the callback method on the thread that has invoked the [ReadAsync](#) or [WriteAsync](#) method.

### 11.5.3 Assemblies and Deployment Elements

Software Toolbox Extender Replacement is part of EasyOPC.NET Extensions, which reside in the [OpcLabs.EasyOpcClassic Assembly](#). Please refer to the base documentation for information about

the [OpcLabs.EasyOpcClassic Assembly](#), and its deployment.

## 12 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.

### 12.1 Deployment Elements

#### 12.1.1 Assemblies (.NET)

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:

- App\_Web\_OpcLabs.EasyOpcClassicRaw.amd64.dll
- App\_Web\_OpcLabs.EasyOpcClassicRaw.x86.dll
- OpcLabs.BaseLib.dll
- OpcLabs.BaseLibForms.dll
- OpcLabs.EasyOpcClassic.dll
- OpcLabs.EasyOpcForms.dll
- OpcLabs.EasyOpcUA.dll

Please refer to "Product Parts" chapter for description of purpose of individual assemblies. You will find them under the Assemblies\NET452 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.

There are other methods of how assemblies can be located, including their installation to GAC (Global Assembly Cache). Which method you use depends on the needs of your application. Unless you have a compelling need, however, we do not recommend installing the assemblies into the GAC.

#### 12.1.2 COM Components and Development Libraries

Because QuickOPC-COM and QuickOPC-UA for COM are implemented by exposing the .NET objects of QuickOPC to COM, you need to basically do two things:

1. Deploy the assemblies needed for .NET (see preceding chapter).
2. Register them using the Assembly Registration tool (Regasm.exe)

The Assembly Registration tool is a part of .NET Framework, and is therefore available on every computer that has the .NET Framework installed. You only need to apply the Assembly Registration tool to the following assemblies:

- OpcLabs.BaseLib.dll
- OpcLabs.BaseLibForms.dll
- OpcLabs.EasyOpcClassic.dll
- OpcLabs.EasyOpcForms.dll
- OpcLabs.EasyOpcUA.dll

For more information, please refer to [http://msdn.microsoft.com/en-us/library/tzat5yw6\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/tzat5yw6(v=vs.110).aspx). It is recommended that you use the /codebaseoption when registering the assemblies.

## 12.1.3 Prerequisites

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

1. Microsoft .NET Framework 4.5.2.

**Needed when:** Always.

In some special cases, you may want to install other prerequisites – please refer to “Prerequisites Boxing” under “Advanced Topics”.

## 12.1.4 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 (GUI or console-based) is needed for this. The GUI-based License Manager is contained in the LicenseManager.exe file. The console-based License Manager is contained in the LMConsole.exe file. Both files are located under the Bin subdirectory of the product.

## 12.2 Deployment Methods

### 12.2.1 Manual Deployment

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

1. Check that proper version of Microsoft .NET Framework is installed, and if it is not, install it.
2. Run the QuickOPC installation program, selecting “Custom install” choice and later “Production installation” type.
3. Perform any steps needed to install your own application.
4.
  - a. Copy the QuickOPC.NET and/or QuickOPC-UA assemblies to their target locations for your application.
  - b. For QuickOPC-COM and QuickOPC-UA for COM, perform additional actions (such as the assembly registrations) as described above (under Deployment Elements chapter) with the respective files.
5. Run the License Manager from the Start menu or the Launcher application, and install the runtime license. Alternative, use command line and the LMConsole utility for this.
6. With QuickOPC-UA only: Run your application once with the administrative privileges, giving it a chance to create its client instance certificate (this step is only needed if you do not provide the client instance certificate in some other way).

### 12.2.2 Automated deployment, roll your own

Using this method, you write an installer for your application, and include an installation of QuickOPC elements in it as well, step by step.

The installer for your application should contain following steps:

1. Check that prerequisites (in their proper versions), are installed, and if they are not, instruct the user to install them (or install them automatically).
2. Install QuickOPC.NET and/or QuickOPC-UA assemblies to their target locations for your application.
3. For QuickOPC-COM and QuickOPC-UA for COM, perform additional actions (such as the assembly registrations) as described above (under Deployment Elements chapter) with the respective files.
4. Perform any steps needed to install your own application.
5. With QuickOPC-UA only: While running with administrative privileges, execute your application with a special

command-line switch that will cause it to call static `EasyUAClient.Install` method. This will create the client instance certificate, in accordance with settings of other static properties on the object (this step is only needed if you do not provide the client instance certificate in some other way).

6. Take care of the license installation (this is described further down).

If you want the end user to install the license:

1. Install LicenseManager.exe or LMConsole.exe (from Bin or Bin\x64).
2. Create an empty registry key  
HKEY\_LOCAL\_MACHINE\SOFTWARE\Licensing\QuickOPC
3. Offer the user an option to run the License Manager.

The end user who deploys the application will then:

1. Run your installer and follow the steps.
2. Use the License Manager (GUI or console-based) 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, you can automate the deployment of the license as well. This can be done in two ways:

- a. Directly - by writing information into the license store. For more information, see **Direct License Deployment (Section 12.2.2.1)**.
- b. By using the **LMConsole Utility (License Manager Console) (Section 4.8.1)**. Your installer will (at least temporarily) place the LMConsole.exe and the actual license file on the disk, and then call the LMConsole utility with corresponding parameters on the command line.

## 12.2.2.1 Direct License Deployment

Proper license (for runtime usage) must be installed on the target computer (the computer where the product will be running).

The standard method for installing the QuickOPC license on runtime machines is manual: The user runs the License Manager utility (GUI or console-based), and installs the license by pointing the program to the license file stored on the disk.

This part of the documentation describes how to automate the above procedure so that the user does not have to interact with the License Manager utility.

If you have multiple QuickOPC versions installed on the computer, you need to deploy the license that covers them all.

### 12.2.2.1.1 License Store

The license is stored in the registry, under the following key:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Licensing\QuickOPC\Multipurpose

In order to install the license, this key and anything that is beneath it, must be properly deployed to the target machine.

Note: On 64-bit systems, the license is stored in 32-bit registry. When the registry is accessed from 64-bit application, you must use the following key instead:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Licensing\QuickOPC\Multipurpose

 On 64-bit systems, if QuickOPC is used from both 32-bit and 64-bit processes, you have to install the license into

both 32-bit and 64-bit parts of the registry.

## 12.2.2.1.2 Deployment Automation

In order to automate the installation of the license, you first need to obtain the proper registry content that corresponds to it.

Use the License Manager utility on the development computer, and install the license you intend to distribute. Then, start the REGEDIT utility, navigate to the registry key for the license, select it, and perform File -> Export command. You will obtain a registration file (with .REG extension) that contains the license information.

Your installation program can then use this information in various ways. For example, it can invoke the REGEDIT utility, and pass it the name of the generated .REG file on the command line. Alternatively, you can use other methods that your installation program provides for manipulating the registry, and simply have it replicate the information contained in the .REG file to the target system.



The license must not be modified in any way. Changing any of its fields (such as the Subject name) will render the license invalid.

Note: Take care on 64-bit systems, as both 32-bit and 64-bit REGEDIT applications exist, and each has a different view of the registry.

## 12.2.3 Automated deployment with Production installer

The QuickOPC contains a separate installation package that is for production purposes only. This installer executable has a "-Production" postfix. It is much smaller than the Full installer: At the time of writing this article, the full installer has 123 MB, while the production installer has only 10 MB. The production installer contains only the assemblies and the LMConsole utility (for licensing), and a code to optionally register the assemblies as COM components, and/or produce the type libraries (TLB) for them. The production installer does not install any Start menu icons. It is intended for embedding in other installation programs. It can be automated from the command line, and can also be run silently.

The developer will find the link to download the production installer in the Redist directory.

The production installer is created using [Inno Setup](#). This means that if you want to automate it from your own installer using command line, you can use all [Setup Command Line Parameters](#) it provides, and check [Setup Exit Codes](#) as needed. You will typically use the /SILENT or /VERYSILENT parameter, possibly combined with /TYPE, /COMPONENTS, /TASKS or /MERGETASKS.

The available setup types are:

Setup Type Name	Description
<b>productionnet</b>	Production installation (.NET)
<b>productioncom</b>	Production installation (COM)
<b>custom</b>	Custom installation

If you need more control over the installed parts, you can specify the components to be installed individually. The available components are:

Component Name	Description
<b>assemblies</b>	Assemblies
<b>assemblies\embedded</b>	Embedded assemblies

<b>comcomponents</b>	COM Components
<b>devlibs</b>	Development Libraries (COM)
<b>tools\licensemanager</b>	License Manager
<b>options\streaminsight</b>	StreamInsight Option

 The License Manager component in the Production installer contains only the console-based license manager (LMConsole.exe), and not the GUI.

You can also instruct the Setup to perform additional tasks during installation. The available tasks are:

Task Name	Description
<b>installtogac</b>	Install component assemblies into the GAC. This is normally neither necessary nor recommended.

The production installer can be passed an additional command line parameter ("~/LicenseFile=...") with a path to the license (.bin) file, causing it to automatically install the license as part of the setup process.

It is also possible to automate the uninstallation. For that, the production installer places a unins000.exe file in the Setup directory of the product folder, where your own uninstaller can find it and call it as well. More information to this you will find in [Uninstaller Command Line Parameters](#) and [Uninstaller Exit Codes](#).

## 13 Advanced Topics

### 13.1 OPC Specifications

QuickOPC.NET and QuickOPC-COM 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 XML-DA 1.01 Specification (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-UA and QuickOPC-UA for COM components directly support following OPC specifications:

- OPC Unified Architecture 1.00 Specifications (Released)
- OPC Unified Architecture 1.01 Specifications (Released)
- OPC Unified Architecture 1.02 Specifications (Released)
- OPC Unified Architecture 1.03 Specifications (Released)

Parts needed to support Data Access and Alarms & Conditions functionality are covered.

QuickOPC is officially certified by OPC Foundation for "UA Generic Client" profile (UA CTT 1.02.0.164), Serial Number 1312CS0045, <https://opcfoundation.org/products/view/52>.



OPC Foundation Certified for Compliance logo is a trademark of the OPC Foundation and may be used only by written permission of the OPC Foundation. Any unauthorized use of the Certified for Compliance logo is prohibited.

### 13.2 OPC Interoperability

The QuickOPC 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 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 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.

## 13.2.1 OPC-UA via UA Proxy

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.

The QuickOPC-UA product 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 (available e.g. from OC Foundation as part of OPC UA COM Interop Components).

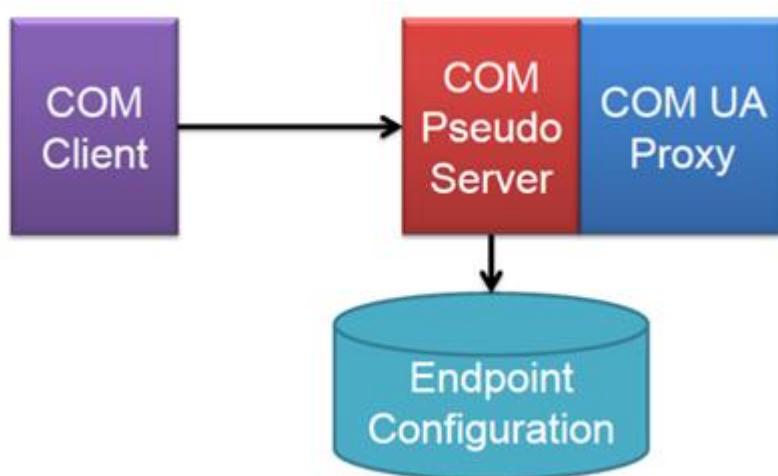
The OPC UA COM Interop Components from OPC Foundation make it possible for existing COM-based applications to communicate with UA applications. OPC Labs is using them to add UA support to existing products. Support for OPC UA COM Interop Components is not currently provided.

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.

## 13.2.2 OPC "Classic" via UA Wrapper

OPC "Classic" refers to a set of older OPC specifications that use Microsoft DCOM as their underlying communication mechanism. These are specifications such as OPC Data Access, OPC Alarms and Events, and others.

QuickOPC-Classic products (QuickOPC.NET and QuickOPC-COM) allow native connections to OPC "Classic" servers.

QuickOPC-UA is not a native OPC "Classic" client, but you can still use it to connect to OPC "Classic" servers, using so-called UA Wrapper (for OPC COM Servers) that is available from e.g. from the OPC Foundation (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.

## 13.3 Application Configuration

OPC Unified Architecture is a rich specification that allows the participating software use different approaches and parameters to achieve their goals. As a result, there is large number of variable settings that can be tweaked to influence the performance, or achieve full interoperability.

QuickOPC-UA attempts to determine many of these setting automatically, and also exposes the most important ones to the developer by means of properties on the various QuickOPC-UA objects. As long as this approach works well for your application, you do not have to do anything extra to configure the UA application.

One important piece of information is the subject name of the application instance certificate. The component uses the subject name to locate the instance certificate in the certificate store, and if it is not found, it attempts to create a new instance certificate, and store it with the subject name. The default subject name is created automatically, based on the description (title) of the application assembly, or (if the preceding yields a name that is too generic, such as 'mscorlib' in many hosted scenarios) from other sources of information, such as the executable's FileVersionInfo, or a main module name.

If you want to use a specific subject name for your application instance certificate, set the

`EasyUAClient.SharedParameters.EngineParameters.ApplicationCertificateSubject` property accordingly.

Other parameters that influence the application certificate creation are

`EasyUAClient.SharedParameters.EngineParameters.ApplicationName`, `ApplicationUriString`, and `ProductUriString`.

If, for any reason, you need to change parameters that cannot be influenced using various properties on QuickOPC-UA objects, you can decide to provide the whole set of parameters yourself. To do so, make sure that the

`EasyUAClient.SharedParameters.EngineParameters.ConfigurationSources` property contains

the `UAConfigurationSources.AppConfig` bit set (in the default value setting, this bit is set, together with `UAConfigurationSources.Internal`).

With this setting on, the whole set of parameters for OPC UA .NET Stack and SDK becomes accessible to you. The UA configuration file can be placed alongside the application's executable, and given the same, but a file extension of `".config.xml"`. Such file will automatically be found and used, with no extra settings needed. If you want to place the UA configuration file elsewhere or give it a different name, you need to make several other steps yourself then:

1. Add an entry for a new configuration section, named "UAClientEngine", into 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 beginning of this file may look like this:

UAClientEngine section declaration in application configuration file

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <section name="UAClientEngine"
type="Opc.Ua.ApplicationConfigurationSection,Opc.Ua.Core" />
    </configSections>
```

...

2. Add a "UAClientEngine" element into the application configuration file. The main purpose of this element is to refer to an external XML file that contains the UA configuration. The element may look like this:

UAClientEngine section definition in application configuration file

```
<UAClientEngine>
    <ConfigurationLocation
xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd">
        <FilePath>MyApplication.Config.xml</FilePath>
    </ConfigurationLocation>
</UAClientEngine>
```

3. Create the UA configuration file with the name specified above, and fill it with settings you need. Unless specified otherwise, the file should be placed alongside the executable of your application.

Please refer to the OPC UA .NET Stack and SDK documentation for information on sections and parameters in the `app.config` file and in the UA configuration file. Following are examples of what can be changed using the UA configuration file:

- Application name
- Application URI
- Product URI
- Store type and store path for the application instance certificate
- Subject name of the application instance certificate
- Trusted issuer certificates

- Trusted peer certificates (store type and store path)
- Transport configurations
- Transport quotas and timeouts
- Trace configuration

If you switch your application so that its UA configuration is placed in an external file like described above, you can then also use the "UA Configuration Tool" from OPC Foundation to configure certain parameters for the application (mainly, the security-related settings).

When the [UAConfigurationSources.Internal](#) bit is set as well, a failure to load the UA configuration from file causes QuickOPC-UA to determine and provide all configuration parameters automatically, which is therefore the default behavior if no UA configuration file is present.

## 13.4 Event Logging

### 13.4.1 Event Logging for OPC "Classic"

Currently, QuickOPC.NET for OPC "Classic" specifications does not have support for event logging.

### 13.4.2 Event Logging for OPC UA

The QuickOPC-UA does not log events on itself, but allows you to implement such functionality. It generates notifications with log entries that contain the necessary data. You can receive these notifications, and implement any kind of event logging with them in your code.

The [EasyUAClient](#) component has a static [LogEntry](#) event. This event is raised for loggable entries originating in the OPC-UA client engine and the [EasyUAClient](#) component. Each event notification carries a [LogEventArgs](#) object, which contains information such as the source of the event, the event message, event type, timestamp, etc.

Note that you should not programmatically use the logging event notifications for any kind of direct actions on the OPC subsystem. For example, the notifications that inform you about the connections and disconnections to/from OPC servers have no direct relation to your OPC method calls, because the component internally merges the requests from multiple objects, and also makes optimizations to prevent unnecessary disconnections etc. The purpose of event logging is different from OPC communication.

In COM, you cannot access static members of the [EasyUAClient](#) object, and therefore you cannot directly use its [LogEntry](#) event. To get around this, create an instance of the [EasyUAClientConfiguration](#) object instead, and use its (non-static) [LogEntry](#) event to achieve the same result.

#### C++

```
// This example demonstrates the loggable entries originating in the OPC-UA client
// engine and the EasyUAClient component.

#include "stdafx.h"
#include <atlcom.h>
#include "_EasyUAClientConfiguration.LogEntry.h"

#import "libid:BED7F4EA-1A96-11D2-8F08-00A0C9A6186D"           // mscorel
using namespace mscorel;
```

```

// OpcLabs.BaseLib
#import "libid:ecf2e77d-3a90-4fb8-b0e2-f529f0cae9c9" \
    rename("value", "Value")
using namespace OpcLabs_BaseLib;

// OpcLabs.EasyOpcUA
#import "libid:E15CAAE0-617E-49C6-BB42-B521F9DF3983" \
    rename("attributeData", "AttributeData") \
    rename("browsePath", "BrowsePath") \
    rename("endpointDescriptor", "EndpointDescriptor") \
    rename("expandedText", "ExpandedText") \
    rename("inputArguments", "InputArguments") \
    rename("inputTypeCodes", "InputTypeCodes") \
    rename("nodeDescriptor", "NodeDescriptor") \
    rename("nodeId", "NodeId") \
    rename("value", "Value")
using namespace OpcLabs_EasyOpcUA;

namespace _EasyUAClientConfiguration
{
    // CEeasyUAClientConfigurationEvents

    class CEeasyUAClientConfigurationEvents : public IDispEventImpl<1,
CEeasyUAClientConfigurationEvents>
    {
        public:
            BEGIN_SINK_MAP(CEeasyUAClientConfigurationEvents)
                // Event handlers must have the __stdcall calling convention
                SINK_ENTRY(1, 1 /*DISPID_EASYUAClientConfigurationEvents_LogEntry*/,
&CEeasyUAClientConfigurationEvents::LogEntry)
            END_SINK_MAP()

            public:
                // Event handler for the LogEntry event. It simply prints out the event.
                STDMETHOD(LogEntry)(VARIANT varSender, _LogEntryEventArgs* pEventArgs)
                {
                    _tprintf(_T("%s\n"), CW2T(pEventArgs->ToString()));
                    return S_OK;
                }
        };
    };

    void LogEntry::Main()
    {
        // Initialize the COM library
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        {
            // The configuration object allows access to static behavior - here, the
            shared LogEntry event.
            _EasyUAClientConfigurationPtr
ClientConfigurationPtr(__uuidof(EasyUAClientConfiguration));

            // Hook events
            CEeasyUAClientConfigurationEvents* pClientConfigurationEvents = new
CEeasyUAClientConfigurationEvents();
            AtlGetObjectSourceInterface(ClientConfigurationPtr,
&pClientConfigurationEvents->m_libid,
            &pClientConfigurationEvents->m_iid,
            &pClientConfigurationEvents->m_wMajorVerNum,

```

```

&pClientConfigurationEvents->m_wMinorVerNum);
    pClientConfigurationEvents->m_iid =
_uuidof(DEasyUAClientConfigurationEvents);
    pClientConfigurationEvents->DispEventAdvise(ClientConfigurationPtr,
&pClientConfigurationEvents->m_iid);

        // Do something - invoke an OPC read, to trigger some loggable entries.
        _EasyUAClientPtr ClientPtr(_uuidof(EasyUAClient));
        ClientPtr->ReadValue(L"http://opcua.demo-
this.com:51211/UA/SampleServer", L"nsu=http://test.org/UA/Data/;i=10853");

        _tprintf(_T("Processing log entry events for 1 minute...\n"));
        Sleep(60*1000);

        // Unhook events
        pClientConfigurationEvents->DispEventUnadvise(ClientConfigurationPtr,
&pClientConfigurationEvents->m_iid);
    }

    // Release all interface pointers BEFORE calling CoUninitialize()
    CoUninitialize();
}

}

```

## Free Pascal

```

// This example demonstrates the loggable entries originating in the OPC-UA
// client engine and the EasyUAClient component.

type
  TClientConfigurationEventHandlers = class
    procedure OnLogEntry(
      Sender: TObject;
      sender0: OleVariant;
      eventArgs: _LogEntryEventArgs);
  end;

// Event handler for the LogEntry event. It simply prints out the event.
procedure TClientConfigurationEventHandlers.OnLogEntry(
  Sender: TObject;
  sender0: OleVariant;
  eventArgs: _LogEntryEventArgs);
begin
  WriteLn(eventArgs.ToString);
end;

class procedure LogEntry.Main;
var
  Client: EasyUAClient;
  EvsClientConfiguration: TEvsEasyUAClientConfiguration;
  ClientConfiguration: EasyUAClientConfiguration;
  ClientConfigurationEventHandlers: TClientConfigurationEventHandlers;
  Value: OleVariant;
begin
  // The configuration object allows access to static behavior - here, the
  // shared LogEntry event.
  EvsClientConfiguration := TEvsEasyUAClientConfiguration.Create(nil);
  ClientConfiguration := EvsClientConfiguration.ComServer;
  ClientConfigurationEventHandlers := TClientConfigurationEventHandlers.Create;
  EvsClientConfiguration.OnLogEntry := @ClientConfigurationEventHandlers.OnLogEntry;

  // Do something - invoke an OPC read, to trigger some loggable entries.

```

```
Client := CoEasyUAClient.Create;
Value := Client.ReadValue(
  'http://opcua.demo-this.com:51211/UA/SampleServer',
  'nsu=http://test.org/UA/Data/;i=10853');

WriteLn('Processing log entry events for 1 minute...');
PumpSleep(60*1000);
end;
```

## Object Pascal

```
// This example demonstrates the loggable entries originating in the OPC-UA
// client engine and the EasyUAClient component.

type
  TClientConfigurationEventHandlers = class
    procedure OnLogEntry(
      ASender: TObject;
      sender: OleVariant;
      const eventArgs: _LogEntryEventArgs);
  end;

// Event handler for the LogEntry event. It simply prints out the event.
procedure TClientConfigurationEventHandlers.OnLogEntry(
  ASender: TObject;
  sender: OleVariant;
  const eventArgs: _LogEntryEventArgs);
begin
  WriteLn(eventArgs.ToString);
end;

class procedure LogEntry.Main;
var
  Client: TEasyUAClient;
  ClientConfiguration: TEasyUAClientConfiguration;
  ClientConfigurationEventHandlers: TClientConfigurationEventHandlers;
  Value: OleVariant;
begin
  // The configuration object allows access to static behavior - here, the
  // shared LogEntry event.
  ClientConfiguration := TEasyUAClientConfiguration.Create(nil);
  ClientConfigurationEventHandlers := TClientConfigurationEventHandlers.Create;
  ClientConfiguration.OnLogEntry := ClientConfigurationEventHandlers.OnLogEntry;
  ClientConfiguration.Connect;

  // Do something - invoke an OPC read, to trigger some loggable entries.
  Client := TEeasyUAClient.Create(nil);
  Value := Client.ReadValue(
    'http://opcua.demo-this.com:51211/UA/SampleServer',
    'nsu=http://test.org/UA/Data/;i=10853');

  WriteLn('Processing log entry events for 1 minute...');
  PumpSleep(60*1000);
end;
```

## PHP

```
// This example demonstrates the loggable entries originating in the OPC-UA client
// engine and the EasyUAClient component.

class ClientConfigurationEvents {
```

```
// Event handler for the LogEntry event. It simply prints out the event.
function LogEntry($Sender, $E)
{
printf("%s\n", $E);
}
}

// The configuration object allows access to static behavior - here, the shared
LogEntry event.
$ClientConfiguration = new COM("OpcLabs.EasyOpc.UA.EasyUAClientConfiguration");
$ClientConfigurationEvents = new ClientConfigurationEvents();
com_event_sink($ClientConfiguration, $ClientConfigurationEvents,
"DEasyUAClientConfigurationEvents");

// Do something - invoke an OPC read, to trigger some loggable entries.
$client = new COM("OpcLabs.EasyOpc.UA.EasyUAClient");
$value = $client->ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853");

printf("Processing log entry events for 1 minute...");
$startTime = time(); do { com_message_pump(1000); } while (time() < $startTime +
60);
```

## Visual Basic (VB 6.)

Rem This example demonstrates the loggable entries originating in the OPC-UA client engine and the EasyUAClient component.

```
' The configuration object allows access to static behavior - here, the shared
LogEntry event.
'Public WithEvents ClientConfiguration1 As EasyUAClientConfiguration

Private Sub LogEntry_Main_Command_Click()
    OutputText = ""

    Set ClientConfiguration1 = New EasyUAClientConfiguration

    ' Do something - invoke an OPC read, to trigger some loggable entries.
    Dim Client As New EasyUAClient
    Dim value As Variant
    value = Client.ReadValue("http://opcua.demo-this.com:51211/UA/SampleServer",
"nsu=http://test.org/UA/Data/;i=10853")

    OutputText = OutputText & "Processing log entry events for 1 minute..." & vbCrLf
    Pause 60000

    Set ClientConfiguration1 = Nothing
    OutputText = OutputText & "Finished..." & vbCrLf
End Sub

' Event handler for the LogEntry event. It simply prints out the event.
Private Sub ClientConfiguration1_LogEntry(ByVal sender As Variant, ByVal eventArgs
As OpcLabs_BaseLib.LogEntryEventArgs)
    OutputText = OutputText & eventArgs & vbCrLf
End Sub
```

## VBScript

Rem This example demonstrates the loggable entries originating in the OPC-UA client

engine and the EasyUAClient component.

## Option Explicit

```
' The configuration object allows access to static behavior - here, the shared
LogEntry event.
Dim ClientConfiguration: Set ClientConfiguration =
CreateObject("OpcLabs.EasyOpc.UA.EasyUAClientConfiguration")
WScript.ConnectObject ClientConfiguration, "ClientConfiguration_"

' Do something - invoke an OPC read, to trigger some loggable entries.
Dim Client: Set Client = CreateObject("OpcLabs.EasyOpc.UA.EasyUAClient")
Dim value: value = Client.ReadValue("http://opcua.demo-
this.com:51211/UA/SampleServer", "nsu=http://test.org/UA/Data/;i=10853")

WScript.Echo "Processing log entry events for 1 minute..."
WScript.Sleep 60*1000

' Event handler for the LogEntry event. It simply prints out the event.
Sub ClientConfiguration_LogEntry(Sender, e)
    WScript.Echo e
End Sub
```

## 13.5 Debugging

In OPC-UA, the component attempts to detect whether a debugger is attached to the application, and if so, it uses a different, much slower keep-alive interval (this keep-alive interval is controlled by the [UAClientSessionParameters.KeepAliveIntervalDebug](#) property, and defaults to 1 day). This allows the developer to break into the debugger and examine the status of the program, without causing the OPC-UA sessions be disconnected due to the fact that keep-alives are not being exchanged while the execution is suspended.

## 13.6 Object Serialization



QuickOPC.NET and QuickOPC-UA 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](#), [EasyAEClient](#) and [EasyUAClient](#) 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](#), and [UANodeElementCollection](#)).
- You can load and store results of reading (e.g. [DAVtq](#), or [UAAtributeData](#)).

- You can load and store condition states ([AEConditionState](#)) and event data ([AEEventData](#)).
- You can load and store all notification data contained in event arguments, for OPC Data Access item changes ([EasyDAItemChangedEventArgs](#)), OPC Alarms and Events notifications ([EasyAENotificationEventArgs](#)), or OPC Unified Architecture monitored item changes ([EasyUADataChangeNotificationEventArgs](#)). This functionality can be used e.g. for easy logging of significant changes and events in the underlying system.

## 13.7 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.NET and QuickOPC-COM:

- 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.

Here are some of the optimizations implemented in QuickOPC-UA:

- Wherever possible, OPC operations are performed on multiple elements at once.
- OPC UA sessions are not closed immediately, but only if not used for certain amount of time.
- Multiple uses of the same OPC server endpoint in the user application are merged into a single request.

In QuickOPC-UA, you can call a static method [EasyUAClient.CloseAll](#) to close all unused sessions that are open to OPC-UA servers.

## 13.8 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:

- With OPC Classic, the OPC server may not be registered on the target machine – permanently, or even temporarily, when a new version is being installed.
- With OPC UA, the OPC server may not be running or registered with the discovery service on the target machine – permanently, or even temporarily, when a new version is being installed.
- The (DCOM, TCP, Web service or other) 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 information (item in OPC Classic, node and attribute in OPC UA) 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 Classic 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. In OPC Classic, objects like OPC groups and OPC items are restored to their original state after a failure. In OPC UA, objects like OPC subscriptions and OPC monitored items are restored to their original state after a failure.

Even if you are using the subscriptions to OPC items (in OPC Classic) or to monitored items (in OPC UA) 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.

In QuickOPC-UA, you can call a static method [EasyUAClient.RetryAll](#) to force a retrial on all objects that are in a failure state. This method terminates the waiting delay, and causes all such objects to be retried on next opportunity.

## 13.9 Timeout Handling

In QuickOPC-UA, timeout values for certain QuickOPC tasks are accessible via properties on the [EasyUAClient](#) object. The explanation of the individual timeout values is provided in the Reference documentation.

The remainder of this chapter applies mainly to QuickOPC "Classic".

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 [InstanceParameters.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.

## 13.10 Data Types

OPC specifications have their own rules about data types, and these have to be reflected in the components and applications. This chapter describes the details of the data type handling.

### 13.10.1 Data Types in OPC Classic

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:

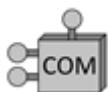
COM type (VARIANT)	.NET type (Object)
VT_EMPTY	<a href="#">System.Object</a> (null reference)
VT_NULL	<a href="#">System.DBNull</a> (singleton class)
VT_I2	<a href="#">System.Int16</a>
VT_I4	<a href="#">System.Int32</a>

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   <i>vtElement</i>	System.Array of the converted <i>vtElement</i> 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 [VarType](#) 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   <i>vtElement</i>	VT_ARRAY   VT_VARIANT( <i>vtElement</i> ) *(see note below)

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.

## 13.10.2 Data Types in OPC-UA

OPC Unified Architecture has its own type system, based on a set of built-in types, and a type graph that supports inheritance and creation of structured types.



The types in Microsoft .NET Framework are different, and all data is represented using an [Object](#) type and its derivatives. 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 components (assemblies) are fully CLS compliant, and that includes the way the data types are converted to and from OPC types.

QuickOPC-UA converts data from OPC-UA to .NET according to following table:

OPC-UA type	.NET type (Object)
Boolean	System.Boolean
Byte	System.Byte
ByteString	System.Byte[]
DataValue	OpcLabs.EasyOpc.UA.UAAttributeData
DateTime	System.DateTime (UTC)
Double	System.Double
ExpandedNodeId	OpcLabs.EasyOpc.UA.AddressSpace.UANodeId
ExtensionObject	decoded body of the extension object
Float	System.Single
Guid	System.Guid
Int16	System.Int16
Int32	System.Int32
Int64	System.Int64
LocalizedText	System.String (see note)
NodeId	OpcLabs.EasyOpc.UA.AddressSpace.UANodeId
QualifiedName	OpcLabs.EasyOpc.UA.AddressSpace.UAQualifiedName
SByte	System.Int16
StatusCode	OpcLabs.EasyOpc.UA.UAStatusCode
String	System.String
UInt16	System.Int32
UInt32	System.Int64
UInt64	System.Decimal
XmlElement	System.Xml.XmlElement
array of <i>elementType</i>	System.Array of the converted <i>elementType</i>

Note: LocalizedText is converted to its text part.

Types that are highlighted do not convert from OPC-UA 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.



In QuickOPC-UA for COM, the same types as in QuickOPC-UA for .NET are used. The type conversions between COM and .NET are performed by the Microsoft interoperability layer. Reference to Microsoft documentation for correspondences between the .NET and COM types.

## 13.11 Multithreading and Synchronization

The [EasyDAClient](#), [EasyAEClient](#) and [EasyUAClient](#) objects and all their related helper objects are thread-safe.



In QuickOPC.NET and QuickOPC-UA, if you are hooking to the event notifications provided by the [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) 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](#), [EasyAEClient](#) or [EasyUAClient](#) object has a [SynchronizationContext](#) property. This property can either be set to a null reference, or to an instance of [System.Threading.SynchronizationContext](#) class. When the [SynchronizationContext](#) is set to a null reference, [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) calls any event handlers or callback methods on its own internal thread, which is possibly 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](#), [EasyAEClient](#) or [EasyUAClient](#) then typically uses the [Post](#) method of this synchronization context to invoke event handlers in your application.

When the [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) object is initially 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](#), [EasyAEClient](#) or [EasyUAClient](#) 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 constructing 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 of the [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) object is called on the form's main thread, as is the case if you place the [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) 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 [System.Threading.SynchronizationContext](#).



In QuickOPC.NET and QuickOPC-UA, objects in the Forms namespaces 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-COM and QuickOPC-UA for COM, if you are hooking to the event notifications provided by the [EasyDAClient](#), [EasyAEClient](#) or [EasyUAClient](#) 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](#), [EasyAEClient](#) or [EasyUAClient](#) 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.

## 13.12 64-bit Platforms

You can create 32-bit or 64-bit, or platform-independent applications with QuickOPC. 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 the Itanium platform (IA-64).

When you use QuickOPC for COM development, the components are compiled using the "just-in-time" by the .NET Framework for either 32-bit or 64-bit execution, depending on the bitness of the process that loads the QuickOPC components.

## 13.12.1 32-bit and 64-bit Code

QuickOPC.NET (for OPC Classic; not QuickOPC-UA) 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 load the so-called mixed mode assemblies (assemblies that contain both managed and native code) for a proper processor architecture.

Any application built with QuickOPC.NET or QuickOPC-UA 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.

## 13.12.2 Classic 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.

## 13.13 Prerequisites Boxing

QuickOPC uses "boxing", a special technique based on file system and registry virtualization, which allows to transparently load and use dependent software that would otherwise have to be installed (and uninstalled separately). With boxing, QuickOPC libraries can simply be placed on disk, registered (for COM) and referenced from .NET or COM code, without installing other software.

The boxing encompasses following software:

1. Microsoft Visual C++ 2015 Redistributable (x86, x64). From Microsoft Corporation.  
Needed because parts of our software are written in managed C++ (for OPC Classic only).
2. OPC Core Components 3.00 Redistributable (x86 or x64). From OPC Foundation.  
Provides proxies/stubs that allow OPC "Classic" components to communicate, and a server enumeration facility (OPCEnum) for a discovery of OPC "Classic" servers.
3. OPC UA Certificate Generator utility.

In order to improve application startup speed, conserve memory, or in edge cases where the "boxing" approach fails, a user can still install any of the above prerequisites, and QuickOPC will use the software installed on the disk. The download links for the individual prerequisites for such case are installed with QuickOPC, under the Redist folder ("3rd-party Redistributable Packages" from the program group under Start menu).

## 13.14 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 above rule only applies to the effects of the installation on the target system – things like Start menu folders and shortcuts, files installed into the Program files directory, etc. As to the actual component assemblies, you can have many versions and even different builds and revisions on the same computer, and they will all work in mutual isolation.

The simulation OPC server (for OPC "Classic") 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.

## 14 Best Practices

This section describes best practices that aid in developing applications and libraries that use the QuickOPC library.

All information described in this section is merely a set of guidelines to aid development. These guidelines do not constitute an absolute truth. They are patterns that we found helpful; not rules that should be followed blindly. There are situations where certain guidelines do not apply. It is up to each individual developer to decide if a certain guideline makes sense in each specific situation.

The guidelines in this section are listed in no particular order.

### 14.1 Do not write any code for OPC failure recovery

As described in Failure Recovery, problems in communications with target OPC server are automatically recognized, and the component performs the recovery automatically. Attempts to duplicate this functionality in your code will interfere with this mechanism and won't work.

It is fine (and indeed, suggested) to catch the errors and handle them appropriately. But the error handling should not include attempts to remedy the situation.

In QuickOPC-UA, you can call a static method [EasyUAClient.CloseAll](#) to close all unused sessions that are open to OPC-UA servers.

### 14.2 With single-operand synchronous methods, only catch OpcException or UAException



Methods that operate on a single element, e.g. an OPC Data Access item, throw an [OpcException](#) exception for all OPC-related errors. In OPC-UA, [UAException](#) is used for the same purpose. The only other exceptions that can be thrown are program execution-related errors (such as [OutOfMemoryException](#), or [StackOverflowException](#)), and argument validation exceptions, such as [ArgumentException](#).

When you catch [OpcException](#) or [UAException](#), you are completely safe that all OPC-related situations that you cannot control will be covered. This includes errors related to arguments that can only be discovered in run-time and depend on the state of environment. For example, an invalid OPC-DA item ID does not throw [ArgumentException](#), but still an [OpcException](#). So, even if something changes in the target OPC server, you can handle it by catching [OpcException](#) (or [UAException](#)). For more information, see Error Handling.

By catching exceptions other than [OpcException](#) (or [UAException](#)) in this case, you could improperly hide programming or catastrophic errors that should not be handled in this way.

The [SubscribeXXXX](#) and [UnsubscribeXXXX](#) methods, such as [EasyDAClient.SubscribeItem](#), [EasyDAClient.UnsubscribeItem](#), or [EasyAECClient.SubscribeEvents](#), work asynchronously. This rule therefore does not apply to them; instead, see Do not catch any exceptions with asynchronous or multiple-operand methods.

### 14.3 Do not catch any exceptions with asynchronous or multiple-operand methods



Asynchronous methods (such as [SubscribeXXXX](#)), or methods that work on multiple elements at once (such as [EasyDAClient.ReadMultipleItems](#)) only throw program execution-related errors (such as [OutOfMemoryException](#), or [StackOverflowException](#)), and argument validation exceptions, such as [ArgumentException](#). The OPC-related exceptions are reported in a different way: By event notifications or callbacks (for asynchronous methods), or in the [OperationResult.Exception](#) properties of the methods return value array (for

multiple-operand methods).

By catching exceptions in this case, you could hide programming or catastrophic errors that should not be handled in this way. For more information, see Errors and Multiple-Element Operations.

## Always test the Exception property before accessing 14.4 the actual result (Value, Vtq, or AttributeData property)

Whenever QuickOPC return an [OperationResult](#) object, or an object derived from it, such as [ValueResult](#), [DAVtqResult](#) or [UAAttributeDataResult](#), the actual result properties are only valid when the result represents a success, i.e. its [Exception](#) property is a null reference. You should therefore always test the [Exception](#) property for null-ness first. Only the properties available in the base [OperationResult](#) class are always accessible.

The same applies to the [Exception](#) property in event notifications, for all event arguments that are derived from the [OperationEventArgs](#) class. For example, the [Exception](#) property must be checked in event handlers for the [EasyDAClient.ItemChanged](#) event (in [EasyDAltemChangedEventArgs](#)), and in event handlers for the [EasyUAClient.DataChangeNotification](#) event (in [EasyUADataChangeNotificationEventArgs](#)).

If you attempt to get other properties while [OperationResult.Exception](#) or [OperationEventArgs.Exception](#) is not null, the outcome is undefined; most likely an exception will be thrown, but that would be a serious exception indicating a programming error. For more information, see Error Handling.

## 14.5 Always test the HasValue property before accessing DAVtq.Value or UAAttributeData.Value

The controlling member in the [DAVtq](#) class (OPC Data Access Value/Timestamp/Quality) is the [Quality](#) property. When the quality is not sufficient, the [Value](#) property is not available (and QuickOPC assures that is a null reference in this case). Getting the [Value](#) property when it is not available makes little sense.

The same applies to the [UAAttributeData](#) class (in OPC Unified Architecture) with regard to its [StatusCode](#) and [Value](#) properties.

It is commonly said that the value is available whenever the quality (or status code) is not Bad, but precisely speaking, this is not true, as there are some border cases (per OPC specifications) in which this rule does not apply. Your best choice is to test the [HasValue](#) property, which has the precise, OPC-compliant functionality built in. Only if [HasValue](#) is true, the [Value](#) property contains valid data.

For more information, see Data Objects.

## 14.6 Use multiple-operand methods instead of looping

Due to way OPC internally works, it is significantly more efficient to perform more operations at once. Whenever your application logic allows it, use methods with the word **Multiple** in their name, i.e. methods that work on multiple elements (such as OPC items) at once, and try to group together a bigger number of operands. This approach gives much better performance.

## 14.7 Do not block inside OPC event handlers or callback methods

You should not block for excessive or significant time inside the OPC event handlers (such as an event handler for [EasyDAClient.ItemChanged](#)) or callback methods. Doing so delays further event processing, and in serious cases, can lead to excessive memory usage and eventually loss of events.

QuickOPC internally queues the events, and therefore any blocking in your code cannot have negative influence on the target OPC server. In the long-term, however, the average processing time for your event handlers or callback methods must be equal to or less than the average rate the OPC events are incoming.

## 14.8 Use generic or type-safe access wherever possible



OPC "Classic" represents data using OLE Automation VARIANT-s, which are transformed to .NET (CLR) objects. As such, a value of any type can appear in place of OPC data. Your code typically knows that certain piece of data (such as OPC item) is of specific type (such as 32-bit signed integer), and expects that the value indeed is of this type.

Instead of trying to put the type conversions and safety checks in your code, you should use the features already available in QuickOPC.NET.

If your language supports generics and you feel comfortable working with them, it is the best choice to use generic types and generic access (see e.g. Generic Types and Generic Access). The second best choice is type-safe access, which provides similar functionality, but without use of generics (see e.g. Type-safe Access).

## 14.9 Use the state instead of handles to identify subscribed entities

The handles returned from any [EasyXXClient.SubscribeXXXX](#) methods are only meant for use as input arguments to other methods that manipulate the existing subscriptions – such as [EasyXXClient.ChangeXXXX](#) or [EasyXXClient.UnsubscribeXXXX](#).

If you need to identify the subscribed entities inside event handlers, use the 'state' argument that can be passed to [EasyXXClient.SubscribeXXXX](#) methods (or a [State](#) property on [XXXXArguments](#) objects). You can pass any type of object in the state, as it is of type [System.Object](#) (and then cast the [System.Object](#) back to the original type in the event handler).

Why shouldn't the handles be used? There are multiple reasons for that:

1. The [SubscribeXXXX](#) methods are asynchronous. There is no guaranteed relation between the moment the [SubscribeXXXX](#) method returns, and the moment when events start flowing into your event handler. The events can start flowing before the methods returns. Or, they can start flowing after the method returns, but before you get a chance to process the returned handles, and store them somewhere where the event handler can access them.
2. Similarly, [UnsubscribeXXXX](#) methods are asynchronous as well, and therefore you cannot know precisely when any event handle became invalid. You may (temporarily) receive events for a previously returned handles, even after the [UnsubscribeXXXX](#) method finishes.
3. With handles, you will need to set up and properly maintain an additional data structure – something that keeps a correspondence between the handles returned (whose values you do not know in advance), and your application data that relate to these handles. Typically, this data structure is a dictionary. Maintaining and accessing this additional data structure means extra code, and inefficient access (application data must be looked up by handle from inside the event handler).
4. Using the handles returned from [SubscribeXXXX](#) in an event handler requires coordination of the access to a data structure where the handles are stored – i.e. an inter-thread synchronization. This leads to complicated and often buggy code, with risk of deadlocks or invalid data being accessed.

All these disadvantages can simply be overcome by passing the application data needed into the [SubscribeXXXX](#) method as a state object, and then casting the [State](#) to the appropriate type and using directly inside the event

handler.

Note that earlier versions passed the handle to your code in the event arguments, making it relatively easy to make a mistake of using them from inside the handler. This is no longer the case with the current version, and the likeliness of violating this best practice is therefore much lower. With extra effort, you might be able to somehow "stick" the handle into the event notification, possibly by back-filling the [State](#) property after the subscription was made. It is strongly recommended that you do not do that for the reasons explained above.

## 15 Additional Resources

If you are migrating from earlier version, please read the "What's New" document (available from the Start menu or the Launcher application).

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

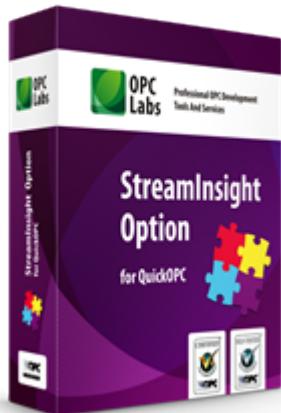
Explore the examples and bonus tools and materials installed with the product.

You may have a look at "OPC Foundation Whitepapers" folder under the Bonus Material group in Start menu. We have included a selection of OPC Foundation White Papers (download) that you may find useful while getting accustomed with OPC in general, or dealing with its specific aspects. If your task involves OPC "Classic", 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.

## 16 Options

### 16.1 StreamInsight Option



Microsoft, MSDN, SQL Server, StreamInsight, Visual C#, Visual Studio, Windows, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

#### 16.1.1 Introduction

QuickOPC is a toolkit for writing OPC client applications. Microsoft StreamInsight is a platform for writing and running complex event processing (CEP) applications.

The StreamInsight Option for QuickOPC allows you to bring in data from OPC sources into StreamInsight, analyze them and process them further, and even feed the results back to OPC servers. Data from OPC can be combined with data from multiple other sources. You can monitor the data for meaningful patterns, trends and exceptions. Data can be analyzed and correlated while they are in-flight.

The StreamInsight Option for QuickOPC supports connections to OPC servers with following specifications:

- OPC Data Access
- OPC Alarms and Events
- OPC Unified Architecture (data access)

The StreamInsight support in QuickOPC is not separate from its other parts. Instead, it builds on top of many features and components already available in QuickOPC. Specifically, the StreamInsight Option makes ideal usage of the connection-less approach in QuickOPC, and the reactive programming model (OPC Reactive Extensions, Rx/OPC). This means that you can also easily combine the StreamInsight development with all other options and features provided by QuickOPC.

This part of the documentation provides information that is specific to the StreamInsight Options for QuickOPC. You should be able to follow the main ideas presented here without advance knowledge of QuickOPC, but for serious development, you will need QuickOPC knowledge as well. For information about QuickOPC itself, please refer to corresponding QuickOPC documentation parts, such as the Users' Guide, or the Reference documentation.

## 16.1.2 Installation and Getting Started

The StreamInsight Option for QuickOPC is installed together with the QuickOPC product, as described further below (there is no separate installation package). Please refer to the Installation section of this document if you need details about the QuickOPC installation procedure.

### 16.1.2.1 Licensing

The StreamInsight Option for QuickOPC is a licensed product. You must obtain an appropriate 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.

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 your purposes, please contact the vendor or producer, describe your needs, and a special trial license may be provided to you.

### 16.1.2.2 Installing a StreamInsight Instance

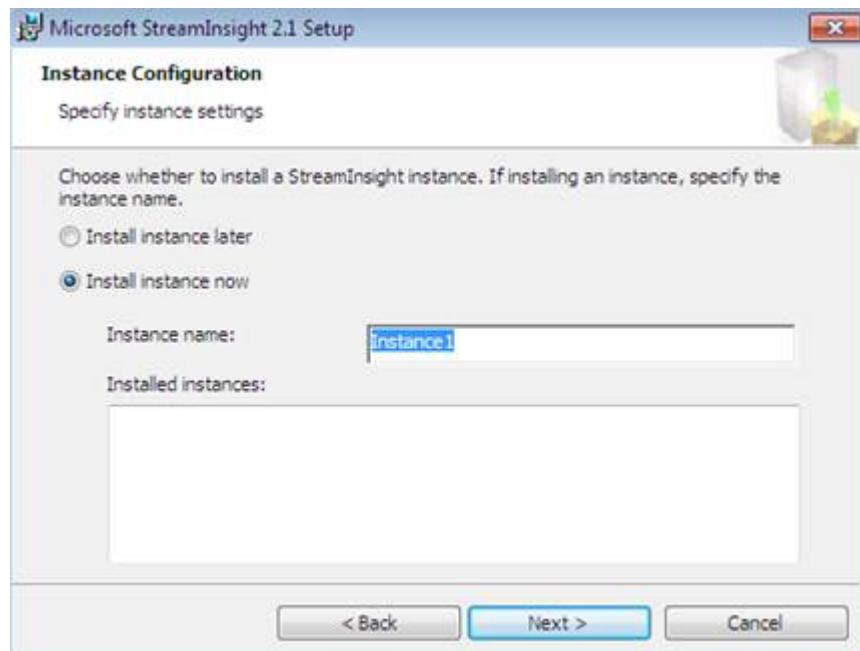
The Getting Started procedures assume an existence of a local StreamInsight instance named Instance1. You need to install Microsoft StreamInsight 2.1, and create this instance. General setup instructions to Microsoft StreamInsight can be found here: <http://technet.microsoft.com/en-us/library/ee378749.aspx>.

We have created our own summary of the steps needed, which you may find useful, and easier to follow:

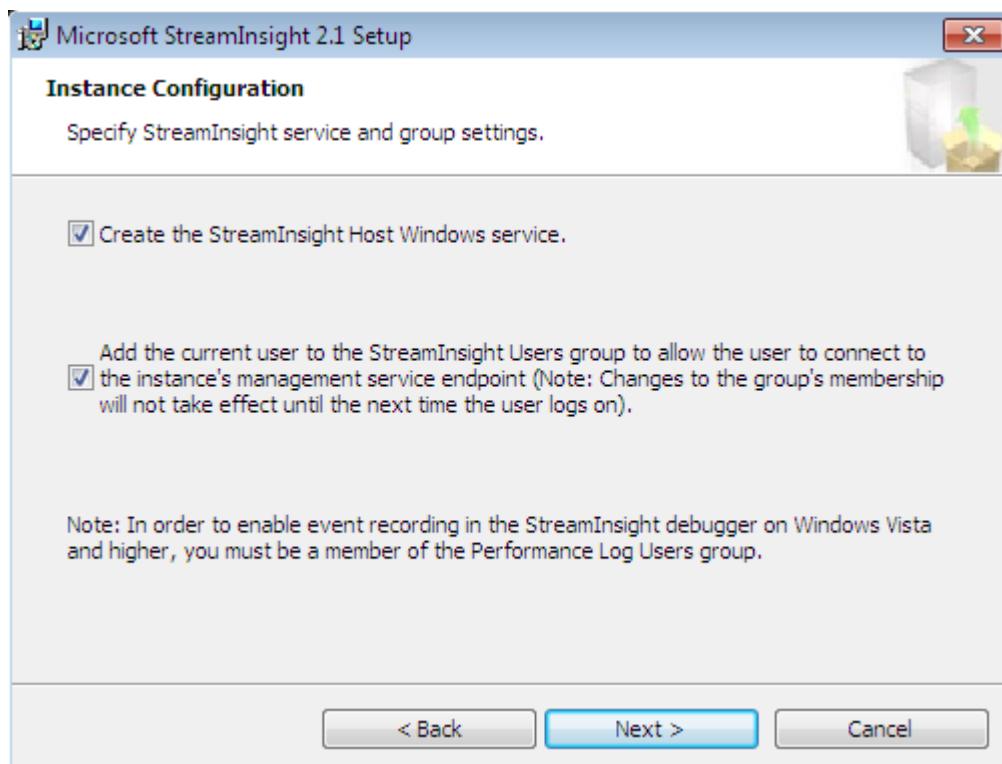
1. Download the x86 (for 32-bit operating systems) or x64 (for 64-bit operating systems) version of StreamInsight.msi (not StreamInsightClient.msi) from Microsoft (<http://www.microsoft.com/en-us/download/details.aspx?id=30149>, or search for "Microsoft SQL Server StreamInsight 2.1"), and run it.

Note: Do not use StreamInsight installation from Microsoft SQL Server 2012 or earlier – it contains Microsoft StreamInsight 2.0 or older, which is not supported by StreamInsight Option for QuickOPC.

2. Go through the installation wizard pages, accepting defaults or entering information as needed. On the "Instance Configuration" page, select "Install instance now" option, and enter "Instance1" (without the double-quotes) into the "Instance name" field. Creation of this instance is recommended now (although you can create it later), because it is the instance that we use in our examples.



3. Continue going through the installation wizard pages. We recommend that you check the "Add the current user to the StreamInsight Users group..." box. Finalize the installation wizard then.



4. If you have checked the "Add the current user to the StreamInsight Users group..." box as instructed above, log out of Windows, and then log in again.

## 16.1.2.3 Building and Running a First StreamInsight Application with OPC

In this Getting Started procedure, we will build and run a console application in C# or Visual Basic that shows how to

create an OPC Data Access event source, and query it for events carrying even data value.

We assume that you have already installed a StreamInsight instance named "Instance1", as described in the previous chapter. We also assume that you have Microsoft Visual Studio 2012 or 2013 installed.

1. Install QuickOPC. If you are reading this text, chances are that you have already done this. The "StreamInsight Option" installation component must be checked during the installation, but it is on by default, so unless you are doing a custom installation and choosing a non-default different installation type, you do not have to do anything special during the QuickOPC installation.
2. From the Start menu (locate the QuickOPC product submenu) or the Launcher application, and select "StreamInsight Option -> Examples Solution (Source Code)". This will open the solution in Visual Studio.
3. In the Solution Explorer of Visual Studio, double-click "Program.cs" under "SimpleDASStreamInsightApplication" project. This will give you a chance to review the one-page program we are going to build and run.
4. Right-click on "SimpleDASStreamInsightApplication" project in the Solution Explorer, and select "Debug -> Start new instance". This will build and run the program.
5. Observe the results in the console window, and after you receive enough events, press Enter to stop the server. The output should look similar to this:

```
MyStreamInsightServer is running, press Enter to stop the server

sink_Server...: 118 09/30/2013 11:35:44 AM; 192
sink_Server...: 120 09/30/2013 11:35:46 AM; 192
sink_Server...: 122 09/30/2013 11:35:48 AM; 192
sink_Server...: 124 09/30/2013 11:35:50 AM; 192
sink_Server...: 126 09/30/2013 11:35:52 AM; 192
sink_Server...: 128 09/30/2013 11:35:54 AM; 192
sink_Server...: 130 09/30/2013 11:35:56 AM; 192
sink_Server...: 132 09/30/2013 11:35:58 AM; 192
sink_Server...: 134 09/30/2013 11:36:00 AM; 192
sink_Server...: 136 09/30/2013 11:36:02 AM; 192
sink_Server...: 138 09/30/2013 11:36:04 AM; 192
sink_Server...: 140 09/30/2013 11:36:06 AM; 192
sink_Server...: 142 09/30/2013 11:36:08 AM; 192
sink_Server...: 144 09/30/2013 11:36:10 AM; 192
sink_Server...: 146 09/30/2013 11:36:12 AM; 192
sink_Server...: 148 09/30/2013 11:36:14 AM; 192
```

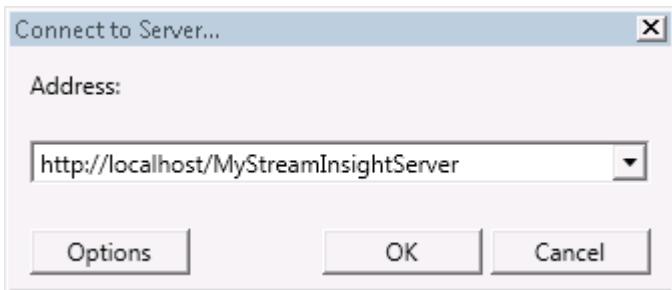
You can also try similar projects for other OPC Specifications:

- **SimpleAESStreamInsightApplication:** For OPC Alarms&Events (OPC-A&E), or
- **SimpleUASStreamInsightApplication:** For OPC Unified Architecture (OPC-UA).

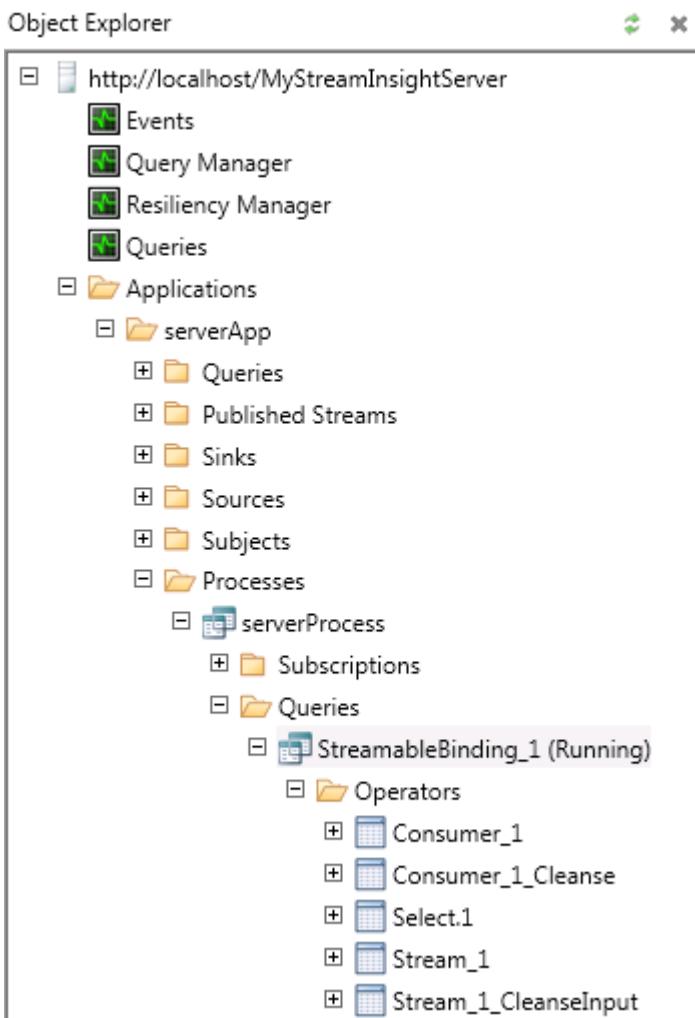
## 16.1.2.4 Debugging the StreamInsight Event Flow

In this Getting Started procedure, we will use the StreamInsight Event Flow Debugger to view how the events are internally processed in StreamInsight. We assume that you have performed the preceding Getting Started procedures already.

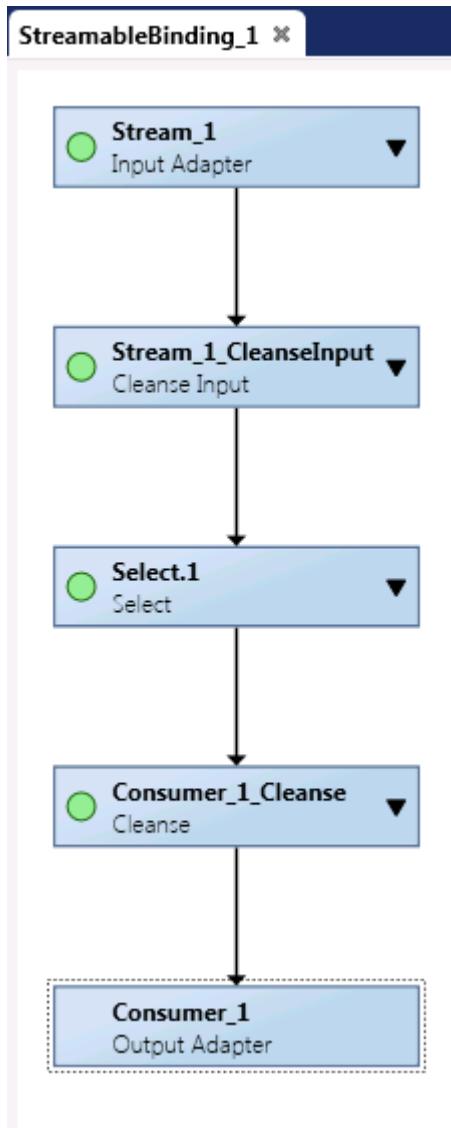
1. As before, right-click on "SimpleDASStreamInsightApplication" project in the Solution Explorer, and select "Debug -> Start new instance". This will build and run the program. Observe the results in the console window, but let the application run.
2. From the Start menu, select "Microsoft StreamInsight 2.1 -> StreamInsight Event Flow Debugger".
3. From the debugger menu, select "File -> Connect to Server... (Ctrl+K)" command.
4. In the "Connect to Server..." dialog, enter the address as <http://localhost/StreamInsight/MyStreamInsightServer>, and press OK.



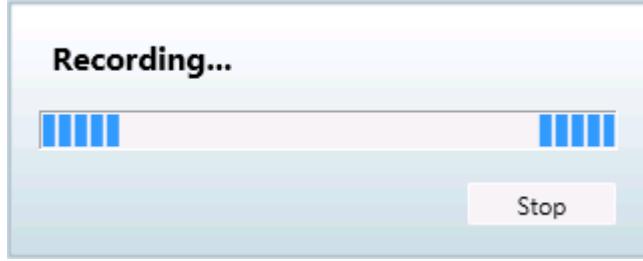
5. In the Object Explorer pane of the StreamInsight Event Flow Debugger, navigate and expand nodes as follows:



6. Under "Applications -> serverApp -> Processes -> ServerProcess -> Queries", right-click the StreamableBinding\_1 query node, and select the "Display Query" command. This will display the query flow chart:



7. Press the "Start Recording Events" button on the toolbar. If it happens to be disabled (grayed out), select the "Display Query" command again, as in the previous step.
8. Let the events be recorded for some time (e.g. about a minute), and then press the "Stop" button on the "Recording" window:



- Alternatively, you can press the "Stop Recording Events" button on the toolbar.
9. You can now view the events at various stages of the query processing, by clicking the downwards-pointing triangle in any of the query operators. For example, events on the "Select.1" operator may look like this:

EventKind	StartTime	EndTime	ErrorCode	Item	Server	VtqPayload.Qui	VtqPayload.Timestamp	VtqPayload.Value
Cti	2013-10-12 11:15:07.7466270							
Insert	2013-10-12 11:15:08.8366269	2013-10-	0	ItemId=Sim://OPC/L	192		2013-10-12 11:15:08.7816269	1500
Cti	2013-10-12 11:15:08.8366270							
Cti	2013-10-12 11:15:09.9266270							
Insert	2013-10-12 11:15:10.9066269	2013-10-	0	ItemId=Sim://OPC/L	192		2013-10-12 11:15:10.7516269	1502
Cti	2013-10-12 11:15:10.9066270							
Cti	2013-10-12 11:15:11.7866270							
Insert	2013-10-12 11:15:12.8776269	2013-10-	0	ItemId=Sim://OPC/L	192		2013-10-12 11:15:12.7216269	1504
Cti	2013-10-12 11:15:12.8776270							
Cti	2013-10-12 11:15:13.8566270							
Insert	2013-10-12 11:15:14.7366269	2013-10-	0	ItemId=Sim://OPC/L	192		2013-10-12 11:15:14.6816269	1506
Cti	2013-10-12 11:15:14.7366270							
Cti	2013-10-12 11:15:14.8266270							

You can sort

and/or filter the events according to various criteria, and the debugger also gives you analysis options, such as Root Cause Analysis or Event Propagation Analysis.

10. Close the StreamInsight Event Flow Debugger.
11. Switch to the console output window of the application, and press Enter to stop it and close the window.

## 16.1.3 Fundamentals

This chapter describes the fundamental concepts used within the StreamInsight Option for QuickOPC.

### 16.1.3.1 Product Parts

#### 16.1.3.1.1 Assemblies

The StreamInsight Option for QuickOPC uses the core QuickOPC assemblies described in the "QuickOPC User's Guide".

In addition, following assemblies are needed to develop programs with the StreamInsight Option for QuickOPC:

Assembly Name or File	Title	Description
<a href="#">OpcLabs.OpcComplexEventProcessing</a>	OPC Labs OPC Complex Event Processing	StreamInsight support for OPC-DA, OPC-A&E and OPC-UA (Unified Architecture) specifications

The assemblies are written to target Microsoft .NET Framework 4.5.

As with the core QuickOPC Assemblies, the StreamInsight Option for QuickOPC assemblies are also delivered with XML comment files, ReSharper annotations, and Code Contracts assemblies.

#### 16.1.3.1.2 Examples

Example code, projects and solutions are available from the Start menu or the Launcher application. You can also access them from the file system, in the ExamplesNet subfolder of the installation folder.

A separate Examples chapter in the StreamInsight Option documentation contain list of all examples and their purpose.

## 16.1.3.1.3 Documentation and Help

The StreamInsight Option documentation is part of the QuickOPC User's Guide and Reference. You can also access it from the Start menu or the Launcher application.

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

The help contents integrates Microsoft Visual Studio 2012/2013/2015 Help (Microsoft Help Viewer 2 format).

## 16.1.3.2 Usage

The StreamInsight Option for QuickOPC is suitable for use from within wide range of tools or languages based on Microsoft .NET Framework. Most commonly, you will use from inside C# or Visual Basic project in Visual Studio.

The project can be e.g. a class library, a console application, Windows Forms or WPF application, or a Web service.

### 16.1.3.2.1 How it Works

The main purpose of StreamInsight Option for QuickOPC is to make it easy to create StreamInsight event sources that generate data that come from underlying OPC servers. These event sources generate StreamInsight events that are then processed inside StreamInsight like any other events.

In order to bring OPC data to StreamInsight, you first reference the necessary QuickOPC assemblies in your project. Then, in your code, you create instances of OPC "observables" – the reactive objects that call a specified method whenever new data is available. Several types of "observables" are available, for OPC Data Access, OPC Alarms and Events, and OPC Unified Architecture, respectively. The observables are then used to create event sources inside StreamInsight, by converting them to temporal streams.

### 16.1.3.2.2 Referencing the Assemblies

In order to develop applications with StreamInsight Option for QuickOPC, your project first needs to reference the appropriate assemblies.

Besides the "standard" assemblies used in your project, and the assemblies you already have referenced for various specific purposes of yours, you will need to reference:

- The QuickOPC assemblies needed for various tasks. They are described in the Product Parts section of the "QuickOPC User's Guide" document.
- The StreamInsight Option for QuickOPC assembly or assemblies (as listed in "Product Parts") in this document
- Microsoft Reactive Extensions Assemblies (installed with Microsoft StreamInsight).
- Supporting Microsoft StreamInsight assemblies.

To make this a bit simpler, we have put together some illustrations below that summarize the assemblies that are typically referenced in your projects.

In a StreamInsight project that uses OPC "Classic", the references list should look similarly to this:

- ◀ ■■■ References
  - Microsoft.ComplexEventProcessing
  - Microsoft.ComplexEventProcessing.ManagementService
  - Microsoft.CSharp
  - OpcLabs.BaseLib
  - OpcLabs.BaseLibExtensions
  - OpcLabs.EasyOpcClassic
  - OpcLabs.EasyOpcClassicExtensions
  - OpcLabs.EasyOpcClassicInternal
  - OpcLabs.OpcComplexEventProcessing
  - System
  - System.Core
  - System.Data
  - System.Data.DataSetExtensions
  - System.Reactive
  - System.Reactive.Providers
  - System.ServiceModel
  - System.Xml
  - System.Xml.Linq

In a StreamInsight project that uses OPC Unified Architecture, the references list should look similarly to this:

- ◀ ■■■ References
  - Microsoft.ComplexEventProcessing
  - Microsoft.ComplexEventProcessing.ManagementService
  - Microsoft.CSharp
  - OpcLabs.BaseLib
  - OpcLabs.BaseLibExtensions
  - OpcLabs.EasyOpcUA
  - OpcLabs.EasyOpcUAExtensions
  - OpcLabs.EasyOpcUAInternal
  - OpcLabs.OpcComplexEventProcessing
  - System
  - System.Core
  - System.Data
  - System.Data.DataSetExtensions
  - System.Reactive
  - System.Reactive.Providers
  - System.ServiceModel
  - System.Xml
  - System.Xml.Linq

*Note: In QuickOPC version 5.40 and later, ignore the EasyOpcClassicExtensions, EasyOpcUAInternal and EasyOpcUAExtensions assemblies above – they are no longer needed, and no longer present.*

You can also combine the two lists together, if your application needs both OPC "Classic" and OPC Unified Architecture servers.

## 16.1.3.2.3 Example Walkthrough

In this chapter, we will explain the usage of StreamInsight Option for QuickOPC on one of the examples that comes with the product. It is the SimpleDASStreamInsightApplication project in C#. This example uses OPC Data Access; other

examples, for OPC Alarms and Events, or OPC Unified Architecture, are quite similar.

In order to make it easy to understand the specifics of StreamInsight Option for QuickOPC, and stay away from parts that common to all StreamInsight application, we have created this set of simple examples by deriving from the examples that Microsoft uses to teach StreamInsight. We can thus stay away from explaining the information that is already available by Microsoft.

Our example will expose an embedded server from inside a console application, and also run the query and display the results. It is based on Microsoft's "StreamInsight Example: Server - Exposing an Embedded Server", [http://msdn.microsoft.com/en-us/library/hh995352\(v=sql.111\).aspx](http://msdn.microsoft.com/en-us/library/hh995352(v=sql.111).aspx). It is recommended that you study the Microsoft's example first.

## Introductory Steps

The first steps are the same:

- Create the (StreamInsight) server instance
- Create the (StreamInsight) application

## Define and Deploy a Source

In the next step, an input source is defined and deployed to the (StreamInsight) server with a name so that it can be used by other StreamInsight clients. In this example, the data is a simple temporal stream of point events generated by the OPC Data Access server.

The `DAItemChangedObservable.Create<int>` method creates an observable that generates a sequence of `EasyDAItemChangedEventArgs<int>` objects. Each notification contains, among other information, the value of a given integer OPC item (if available), as it is delivered by the OPC server. Because the `EasyDAItemChangedEventArgs<int>` objects are not directly suitable as StreamInsight event payloads, we convert them to `DAItemChangedPayload<int>`, which is pre-made payload object provided by the StreamInsight Option for QuickOPC.

### Define and Deploy a Source

```
// DEFINE a simple SOURCE (returns a point event every second)
const string machineName = "";
const string serverClass = "OPCLabs.KitServer.2";
const string itemId = "Simulation.Incrementing (1 s)";
var observable = DAItemChangedObservable.Create<int>(machineName,
    serverClass, itemId, 100);
var mySource = myApp
    .DefineObservable(() => observable)
    .ToPointStreamable()
    .EventArgs =>
    PointEvent.CreateInsert(DateTimeOffset.Now,
        (DAItemChangedPayload<int>)eventArgs),
    AdvanceTimeSettings.StrictlyIncreasingStartTime);
```

## Compose a Query over the Source

Next, compose a query over the input source. The query uses LINQ as the query specification language. In this example, the query returns the events where the value of the OPC item is an even number.

### Compose a Query over the Source

```
// Compose a QUERY over the source
// (return every event carrying even data value)
var myQuery = from e in mySource
               where e.VtqPayload.Value % 2 == 0
               select e;
```

Technically, this definition translates to a filter operator that drops all events from the sequence that do not fulfill the filter predicate (where `e.VtqPayload.Value % 2 == 0`) and returns the event value. For more information about LINQ query operators, see [Using StreamInsight LINQ](#).

## Define and Deploy a Sink

Next, an output sink is created that can be bound to the query and process the resulting sequence. In this example, a simple function is created that simply writes the stream values to the console. The `DAItemChangedPayload<int>` class has a convenient `ToString()` method that returns the relevant payload information nicely formatted for output.

### Define and Deploy a Sink

```
// DEFINE a simple observer SINK (writes the value to the server console)
var mySink = myApp.DefineObserver(() =>
    Observer.Create<DAItemChangedPayload<int>>(
        payload => Console.WriteLine("sink_Server...: {0}", payload)));
```

The sink is then deployed to the server with a name, in the same way as in the original Microsoft example.

## Bind and Run the Query and Sink

The remainder of the example is not doing anything specific to OPC. The observable query is bound to the observer output sink, and the query is then run in a process in the server. This process continues to run until the user stops it by typing in the console.

### 16.1.3.3 Creating OPC Event Sources

The power of StreamInsight Option for QuickOPC comes from its OPC event sources. On the surface, they are just handful of relatively "small" objects, with uncomplicated interface. Below the surface, however, there is an enormous amount of functionality, providing things such as type conversions, error handling, connection pooling, connection persistence, optimizations, OPC compliance, wide interoperability, and security.

#### 16.1.3.3.1 OPC Observables

The StreamInsight Option for QuickOPC provides OPC event sources in form of OPC observables, i.e. .NET reactive object with `IObservable` interface. In order to simplify creation of these OPC event sources, static classes with various overloads of `Create` method are provided:

- In order to create OPC Data Access event source, call one of the `DAItemChangedObservable.Create< TValue >` method overloads, passing it arguments such as the OPC server ProgID, OPC item ID, and requested update rate.
- In order to create OPC Alarms and Events event source, call one of the `AENotificationObservable.Create` method overloads, passing it arguments such as the OPC server ProgID, notification rate, and subscription filter.
- In order to create OPC Unified Architecture (data) event source, call one of

the `UADataChangeNotificationObservable.Create<TValue>` method overloads, passing it arguments such as the OPC server endpoint, OPC node ID, and sampling interval.

For more information and details on the OPC Observables, see the appropriate chapters (separately for OPC-DA, OPC-A&E, and OPC-UA) in the Reactive Programming Model section of this document.

## 16.1.3.3.2 Errors in OPC Sequences

When there is an error in communication with the target OPC server, the OPC observable sends a message (one of the `XXXXEventArgs` objects) with following characteristics:

- The `Exception` property is non-null, and contains an exception object that describes the error.
- The `ErrorCode` property contains a negative number.
- The `ErrorMessage` property contains a string that describes the error.
- The `Succeeded` property is equal to '`false`'.
- Other properties that normally contain the actual notification data may not contain valid data, or may be null.

Your StreamInsight application code needs to be prepared for such messages. As a minimum, if you want to ignore error situations, you need to filter out the error messages, and in case of error, do not access the fields of the `XXXXEventArgs` that do not contain valid data.

When you use the pre-defined OPC Event Payload classes (described further below), the conversion code already takes the possible errors into account, and behaves accordingly, by filling in the payload fields with default values, setting negative error code in the payload, etc.

Note that the errors described here are NOT errors in the sense of calling the `OnError` method of the `IObserver` interface. In fact, the OPC observables never call `OnError`, because that would mean an irrecoverable error and a completion of the sequence. The OPC errors are, however, potentially transient and recoverable, and must therefore be a part of normal notifications.

## 16.1.3.3.3 Other Special Cases in OPC Sequences

Leaving error situations aside, you still need to be careful when interpreting properties of `XXXXEventArgs` objects provided by the OPC observables. For precise description, study the corresponding topics in the "QuickOPC User's Guide" and the reference documentations. Here are some basic points to be aware of:

- In OPC Data Access, the `EasyDAItemChangedEventArgs<T>.TypedVtq.TypedValue` property does not contain valid data when the quality of the VTQ denotes a "Bad" quality.
- In OPC Alarms and Events, the `EasyAENotificationEventArgs.EventData` property does not contain valid data for in a notification that signifies a re-gained connection to the OPC server, or in a notification that indicates that a subscription refresh is complete.
- In OPC Unified Architecture, the `EasyUADataChangeNotificationEventArgs<T>.TypedAttributeData.TypedValue` property does not contain valid data when the status code of the attribute data denotes a "Bad" status.

When you use the pre-defined OPC Event Payload classes (described further below), the conversion code already takes these special cases into account, e.g. by filling in the payload fields with special values.

## 16.1.3.4 OPC Event Payloads

When you develop application for Microsoft StreamInsight, you are responsible for defining *event payloads*, i.e. .NET data structures that hold the data associated with StreamInsight events. The fields in the payload are user-defined and their types are based on the .NET type system.

There are various rules and limitations regarding the design of event payloads for Microsoft StreamInsight. For details,

refer to StreamInsight documentation, e.g. the "Event Structure" article ([http://msdn.microsoft.com/en-us/library/ee378905\(v=sql.111\).aspx](http://msdn.microsoft.com/en-us/library/ee378905(v=sql.111).aspx)).

The objects provided as notifications by OPC observables are not directly suitable as StreamInsight event payloads. You need to define the event payload structure for StreamInsight according to the needs of your application. There is no general rule as to what to use as StreamInsight event payload – it all depends on the specifics of the application. You can, for example

- Pick up a single field from the notification provided by the OPC observable – for example, the actual data value. This may be very useful in applications that only need the data values, but beware of special cases such as communication errors etc.
- Create a custom .NET data structure, and define its fields and organize the data in it precisely as your application requires. Then, convert the notifications provided by the OPC observable, to instances of this >NET event payload structure.
- Use ready-made OPC Event Payload classes provided by the StreamInsight Option for QuickOPC. These classes are already designed to conform to StreamInsight rules, and payload instances can be easily created by converting them from the notifications coming from OPC observables.

The following paragraphs describe the OPC Event Payload classes for various OPC specifications.

## 16.1.3.4.1 Common Payload Characteristics

All OPC Event Payload classes contain an integer `ErrorCode` field. This field is equal to zero for successes, it contains a positive number for warnings, and a negative number for errors.

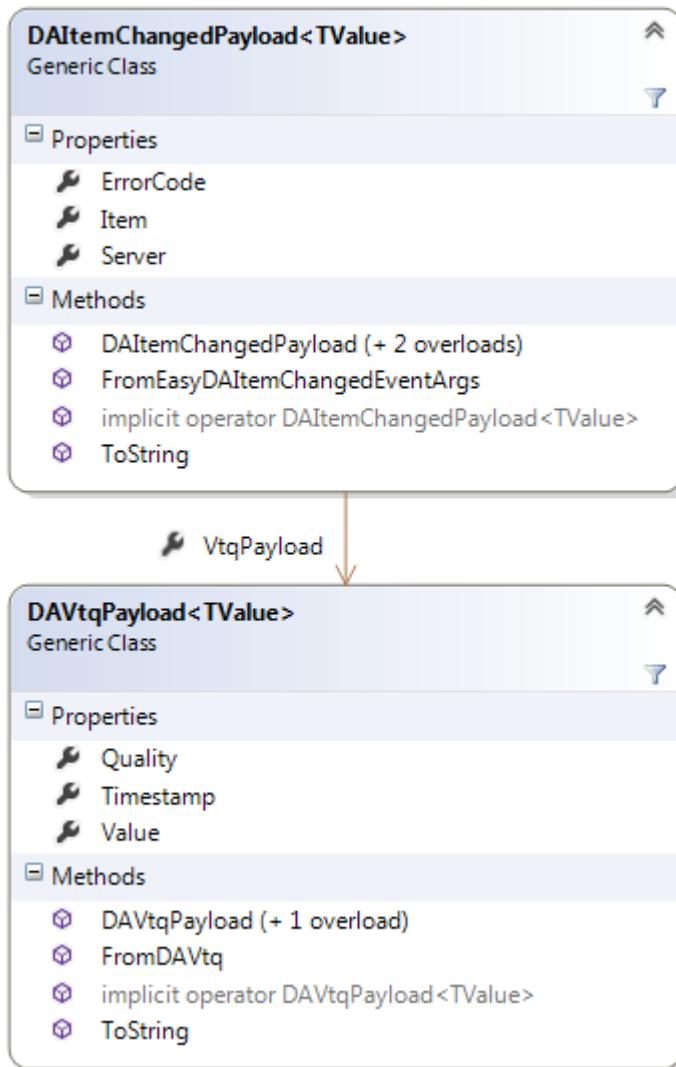
The OPC Event Payload classes contain no more information about the error (such as the error message), due to possible negative effect on the payload size, memory consumption, and performance. If your application needs extra information such as the error message strings, derive your payload from one of the OPC Event Payload classes and extend it, or use your own even payload structure.

Each OPC Event Payload class contains a method (called `FromXXXX`) that creates it from a non-payload object that comes natively from the OPC observable, and a corresponding implicit conversion operator.

For easy viewing, logging and similar purposes, all OPC Event Payload classes contain a `ToString()` method that formats the contents of relevant fields of the payload into a readable string.

## 16.1.3.4.2 Payloads for OPC Data Access

The event payload class for OPC Data Access is the `DAItemChangedPayload< TValue >`, where `TValue` is the type of the OPC values. This class contains an embedded `DAVtqPayload< TValue >` object, which contains the actual OPC value, timestamp, and quality. This object is always present; in case of errors, it contains default values.



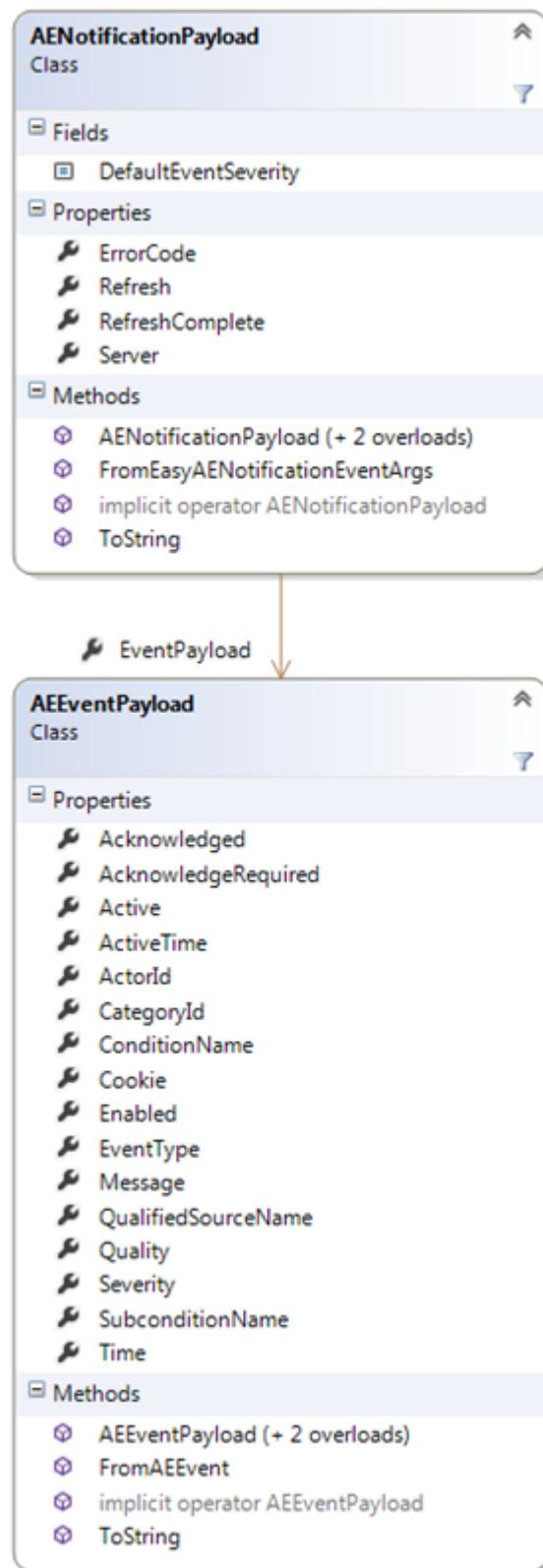
In addition, there are `Server` and `Item` string properties, identifying the source of the event. For a single OPC observable, they always contain the same value, but they become useful once you start combining events from different OPC sources into one stream.

### 16.1.3.4.3 Payloads for OPC Alarms and Events

The event payload class for OPC Data Access is the [AENotificationPayload](#) class. This class contains an embedded [AEEEventDataPayload](#) object, which contains the actual OPC event. This object is always present; in case of errors and special cases, it contains default or pre-defined values.

A special event priority value is used when no OPC event is present (in case of errors, but also re-gained connections, and a completed refresh). Normal OPC event priorities are in the range from 0(1) to 1000. When creating a payload for a notification where no actual OPC event is present, an event priority value of 9999 is used. The value 9999 is chosen to be higher than the severity of any OPC-A&E event, yet still in a 4-digit range.

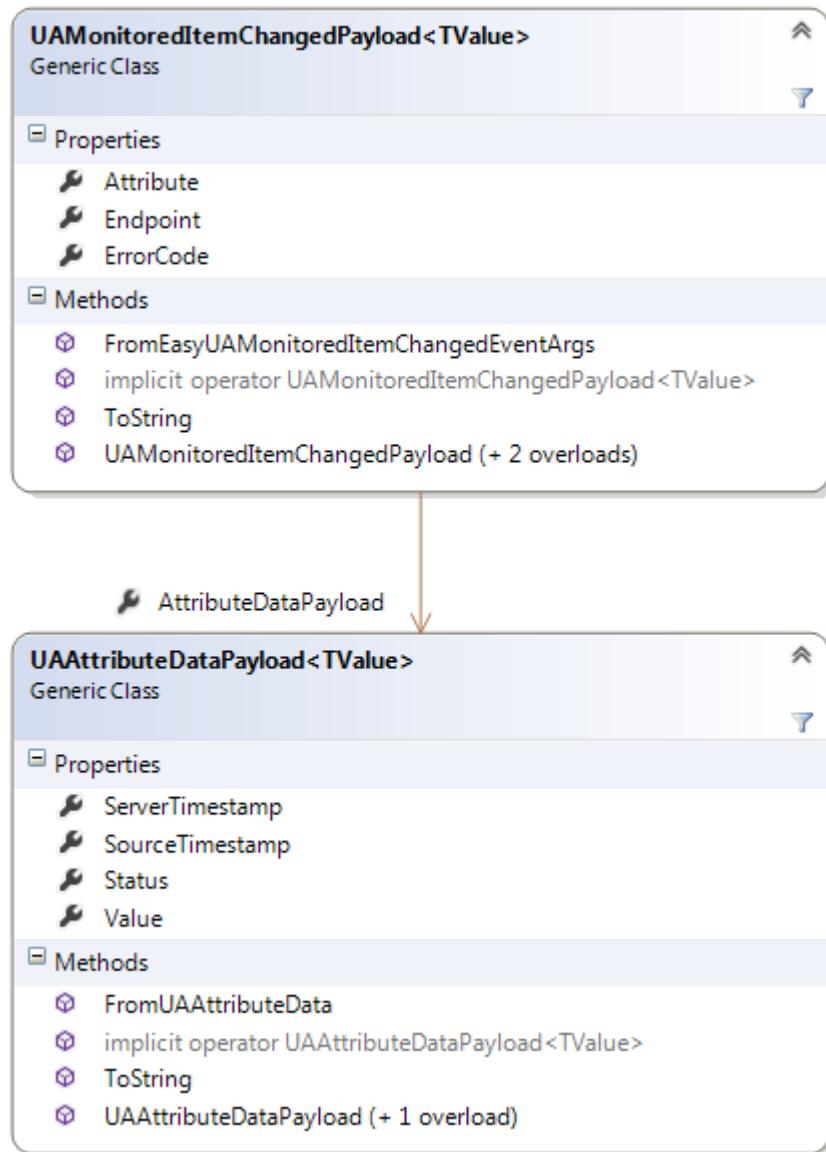
The payload class does not contain any fields for OPC server-specific event attributes. If you need to process the OPC event attributes in StreamInsight, derive your own class from [AEEEventDataPayload](#), add the fields you need, and develop a code that extracts the attributes from the [AEEEventData](#) object (in [EasyAENotificationEventArgs.EventData](#)).



In addition, there is a **Server** string property, identifying the source of the event. For a single OPC observable, it always contains the same value, but it becomes useful once you start combining events from different OPC sources into one stream.

## 16.1.3.4.4 Payloads for OPC Unified Architecture

The event payload class for OPC Unified Architecture is the `UADataChangeNotificationPayload<TValue>`, where `TValue` is the type of the OPC values. This class contains an embedded `UAAttributeDataPayload<TValue>` object, which contains the actual OPC value, timestamps, and status. This object is always present; in case of errors, it contains default values.



In addition, there are `Endpoint` and `Attribute` string properties, identifying the source of the event. For a single OPC observable, they always contain the same value, but they become useful once you start combining events from different OPC sources into one stream.

## 16.1.3.5 Time in OPC and StreamInsight

In StreamInsight, the time that comes with events has to be monotonic, i.e. it cannot go back. More precisely, at the moment a CTI (Current Time Increment) event with certain time is stored into a stream, your application has committed to this point in time, and cannot store any events into the stream that have time that precedes the last CTI time. And, you must generate the CTI events, because before committing to certain time (by storing a CTI), StreamInsight cannot perform processing on the incoming stream events, because it does not know for sure that they represent the entirety of the information to be processed.

The time presents a challenge in many StreamInsight applications, and the combination of StreamInsight and OPC

brings some additional points to consider. We try to present a summary of them below.

## 16.1.3.5.1 Time Synchronization in OPC

In OPC "Classic" (COM-based) specifications, such as OPC Data Access or OPC Alarms and Events, there is little or nothing specified about the times the OPC servers should use, and the timestamps that come with the OPC data. Specifically, there is no requirement that the OPC servers must have proper and synchronized time.

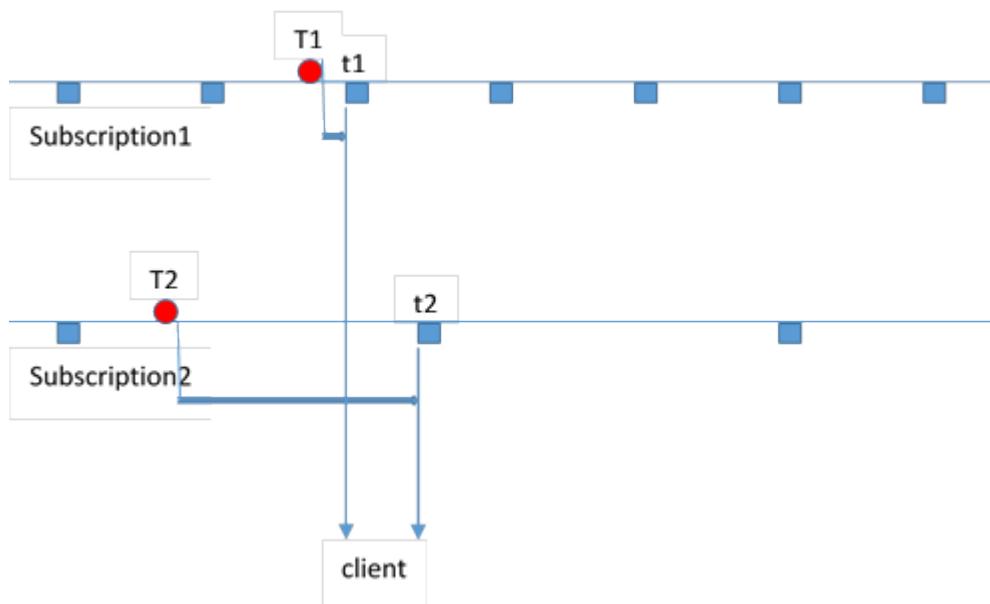
This becomes an issue when your application starts combining data from multiple OPC servers, and even more if they run on different computers. Unless you can guarantee their time synchronization by some application-specific means, you must assume that their times might be out of sync. Events that come in sequence from different servers may have their timestamps in decreasing order instead.

The OPC Unified Architecture specification has given more thought to time, and (with some simplification), we may say that it requires the times to be synchronized on both sides of the communication (between the OPC server and the OPC client), although it does not provide a mechanism to do, and again relies on external means to achieve.

The OPC-UA requirements consequently mean that if the OPC client (such as your application with StreamInsight Option for QuickOPC) communicates with multiple OPC servers, all their times should be synchronized. In reality, however, not all systems will be set according to OPC specifications, and adding to that, the synchronization required by OPC-UA is not precise, and there may be several seconds' difference between the times on various computers. In StreamInsight, however, the time monotony requirement is a strict one – you cannot go back in time even by a fraction of a second.

## 16.1.3.5.2 Time in OPC Subscriptions

Even if the times used by all OPC servers were perfectly synchronized, one needs to consider how the OPC subscriptions are realized and what effect it has on the time monotony. In OPC, data change or event delivery is not usually immediate, for performance (throughput) reasons. Notifications are transferred in larger chunks, and only after certain time elapses, so that they can accumulate.



Let's have a look at following situation where we have two subscription, using different notification rates:

The red circles denote times when new source data appear. The blue squares denote times when the server delivers notifications to the client.

In this case, following occurrences happen in real time sequence:

1. At time T2, the new source data appear on Subscription2 (which has slower notification rate).
2. At time T1, the new source data appear on Subscription1 (which has faster notification rate).
3. At time t1, the data from Subscription1, carrying a T1 timestamp, are delivered to the client.
4. At time t2, the data from Subscription2, carrying a T2 timestamp, are delivered to the client.

As a result, the event that has appeared later (on T1) is delivered to the client earlier than the event that has appeared sooner (on T2). If the StreamInsight application wanted to use the timestamps of the source data, the time monotony requirement would be broken.

In addition, even with a single subscription, the OPC "Classic" server may send data with timestamps in reversed time order, albeit that would be quite rare and probably an indication of a buggy server.

### 16.1.3.5.3 Conclusions

The above findings can be summarized briefly as follows:

Microsoft StreamInsight has strict requirements on time monotony, which OPC cannot fulfill directly.

While the above statement may sound somewhat pessimistic, there are several ways how to adapt OPC times to StreamInsight, and we will mention some of them here. There is also material available from Microsoft and on the Web from various sources that deals with such issues, because other data sources that are used to feed events into StreamInsight deal with the same or similar issues.

Here are some options that can be used in various situations, either alone, or in combination:

- Use current system time on the client side for StreamInsight time, i.e. do not use OPC timestamps at all. This is the approach used in the simple examples shipped with the product, and in this document. The OPC timestamps become fields of the event payload, and can therefore have any valid value.
- You may use only single notification rate, or avoid combination of events from different sources before you advance the StreamInsight time.
- Use various [AdvanceTimeSettings](#) available in StreamInsight. For example, you may place an upper bound for how far back from the previous CTI the start time of the future event can be. Using [AdvanceTimePolicy](#), you can also choose whether incoming events with start time before the current application time are dropped, or simply adjusted to the current application time.

The way that you resolve the issue of adapting OPC times to StreamInsight depends completely on the needs of your application, and there is no single approach that fits all applications.

### 16.1.4 Examples

The QuickOPC installation contains usage examples for the StreamInsight Option in various programming languages. The examples are being updated and enhanced more frequently than the base product, so if you like them, please check for newer builds of QuickOPC from time to time.

#### 16.1.4.1 List of Examples

The examples assume an existence of a local StreamInsight instance named Instance1.

The examples are currently provided mainly for Microsoft Visual Studio 2012. They can all be automatically converted to Microsoft Visual Studio 2013.

The examples are targeting Microsoft .NET Framework 4.5. In Visual Studio, it is possible to re-target the projects to newer framework versions (e.g. 4.5.1) by changing the appropriate setting in the properties of the project.

Visual Studio solution with examples in C# is available from the Start menu or the Launcher application.

	Visual C#	Visual Basic	Visual F#	Visual C++
<b>SimpleAESTreamInsightApplication:</b> This example shows how to create an OPC Alarms&Events event source, and query it for events with severity 20 or higher.	✓			
<b>SimpleDASTreamInsightApplication:</b> This example shows how to create an OPC Data Access event source, and query it for events carrying even data value.	✓			
<b>SimpleUASTreamInsightApplication:</b> This example shows how to create an OPC Unified Architecture data event source, and query it for events carrying even data value.	✓			

## 16.1.5 Application Deployment

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

For application deployment, you need to consider a combination of rules that exist for QuickOPC, and for StreamInsight.

To properly deploy the QuickOPC part, please refer to the “Application Deployment” chapter in this document. All information given there (for .NET) also applies to applications created with the StreamInsight Option for QuickOPC. In Addition, you need to deploy the assemblies that are specific to the StreamInsight Option for QuickOPC, but you do it in the same way as with other assemblies. You need to deploy one additional assembly:

- **OpcLabs.OpcComplexEventProcessing.dll**

For deploying the StreamInsight server, and the StreamInsight entities into a StreamInsight server, please see the appropriate StreamInsight documentation.

## 16.1.6 Additional Resources

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

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

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

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

## 17 Examples

This documentation section consists of two parts:

- .NET Examples, and
- COM Examples.

Depending on whether you develop OPC applications for Microsoft .NET, or are using Microsoft COM, please refer to the corresponding part.

Examples are provided for OPC "Classic" (COM-based) specifications, and OPC Unified Architecture (OPC UA).

The QuickOPC installation contains usage examples in various programming languages. The examples are being updated and enhanced more frequently than the base product, so if you like them, please check for newer builds of QuickOPC from time to time.

Note that the examples are different for QuickOPC.NET and QuickOPC-COM. The text is marked with corresponding .NET or COM icon further below.

### 17.1 .NET Examples

The examples are currently provided for Microsoft Visual Studio 2012. They can all be automatically converted to Microsoft Visual Studio 2013 or 2015.

The examples are targeting Microsoft .NET Framework 4.5.2. In Visual Studio, it is possible to re-target the projects to newer framework versions (e.g. 4.6) by changing the appropriate setting in the properties of the project.

Visual Studio solutions with examples in Visual Basic, C#, F# and C++ are available from the Start menu or the Launcher application.

Following categories of .NET examples are available:

- **Examples for OPC Classic (OPC DA, OPC XML-DA and OPC A&E) (Section 17.1.1)**
- **Example for OPC Unified Architecture (OPC UA) (Section 17.1.2)**
- **Integration Examples (Section 17.1.3)**
- **Reactive Programming Examples (Section 17.1.4)**
- **LINQPad Examples (Section 17.1.5)**

#### 17.1.1 Examples for OPC "Classic" (OPC-DA, OPC XML-DA and OPC-A&E)

	Visual C#	Visual Basic	Visual F#	Visual C++
ArrayValues: Shows how to write into an OPC item that is of array type, and read the array value back.	✓	✓		
AutoRefreshWeb: Web application with a screen that refreshes itself periodically.	✓	✓		
BrowseAndReadValues: Console application that recursively browses and displays the nodes in the OPC address space, and attempts to read and display values of all OPC items it finds.	✓	✓		
BrowseBranchesWeb: Browses the branches in the OPC server (ASP.NET Web application).	✓	✓		

	Visual C#	Visual Basic	Visual F#	Visual C++
BrowseServersWeb: Browses the available OPC servers (ASP.NET Web application).	✓	✓		
ConsoleApplication1: The simplest console application. Reads and displays an OPC item value.	✓	✓	✓	✓
ConsoleDataTypes: Shows how different data types can be processed, including rare types and arrays of values.	✓	✓		
ConsoleDemo: Shows how to read multiple items with one method call.	✓	✓		
ConsoleEvents: Hooking up events and receiving OPC item changes in a console application.	✓	✓		
ConsoleLiveMapping: Creates an object structure for a boiler, describes its mapping into OPC Data Access server using attributes, and then performs the live mapping. Boiler data is then read, written and/or subscribed to using plain .NET object access.	✓	✓		
DataGridWebApplication: Demonstrates how easily can WebControls.GridView be populated with data read from OPC Data Access server.	✓	✓		
DocExamples: A collection of OPC "Classic" console-based examples that illustrate the use of individual objects in the product, and their members. These are the same examples that appear in reference documentation, with an extra control routine that allows the user to choose an example to be performed.	✓	✓		
EasyOpcNetDemo: This is a source of the Demo application for OPC "Classic" that ships with the QuickOPC.NET product. The application shows most product functions, including the browsing forms, OPC property access, and event-based subscriptions.	✓	✓		
EasyOpcNetDemoXml: This is a source of the Demo application for OPC "Classic" (OPC XML-enabled) that ships with the QuickOPC.NET product. The application shows most product functions, including the browsing forms, OPC property access, and event-based subscriptions. The defaults are pre-filled for OPC XML-DA demo server, but the application is written in such a way that it can handle COM servers as well.	✓			
HmiScreen: Windows Forms application that shows how to use implement an HMI screen by storing OPC Item IDs in the Tag property of screen controls, and animate the controls by subscribing to all items at once. Also shows a possibility how to write to an OPC item form the screen.	✓	✓		
ListView1: Shows how (Windows Forms) ListView items can be populated with OPC data, either using explicit Read, or with a subscription.		✓		
<b>LiveBindingDemo:</b> Shows live binding of OPC Data Access information (from simulation OPC server) to standard Windows Forms controls (Microsoft). All binding to OPC data is achieved with no manual coding, only by configuring the components in Visual Studio.	✓	✓		

	Visual C#	Visual Basic	Visual F#	Visual C++
				
<b>LiveBindingDemo2:</b> Shows advanced live binding features. Among others, it demonstrates:	✓			
<ul style="list-style-type: none"> <li>Binding kinds: binding to local vs. UTC timestamp.</li> <li>Conversions: Use of LinearConverter, even bi-directionally.</li> <li>Animations: Moving a control around the form, depending on an OPC tag value.</li> <li>Cumulative: Adding incoming values to ListBox or ListView.</li> </ul>				
<b>LogAsStringToSql:</b> Logs OPC Data Access item changes into an SQL database, using a subscription. Values of all data types are stored in a single NVARCHAR column.	✓	✓		
<b>LogAsUnionToSql:</b> Logs OPC Data Access item changes into an SQL database, using a subscription. Values of all data types are stored in separate columns.	✓	✓		
<b>LogToSqlEnhanced:</b> Logs OPC Data Access item changes into an SQL database, using a subscription. Item values and qualities are stored in their respective columns. Notifications with the same timestamp are merged into a single row.	✓	✓		
MultipleItems: Show how to write into multiple OPC items using a single method call, and read multiple item values back.	✓	✓		
OpcDAQualityDecoder: A simple WPF application that decodes the cryptic OPC quality numbers into the separate fields and their symbolic representation.	✓			
OvenControl: Monitors sensors in an industrial oven, indicates level alarms by changing colors, allows the user to change a setpoint, and logs the values into a CSV file.	✓	✓		
QualityStrings: Shows how numerical OPC quality codes are converted to displayable strings (Windows Forms application).	✓	✓		
SimpleLogToSql: Logs OPC Data Access item changes into an SQL database, using a subscription. Values of all data types are stored in a single SQL_VARIANT column.	✓	✓		
SubscribeFromXml: Loads list of OPC items from an XML file and subscribes to them.	✓	✓		
SubscribeToMany: Demonstrates and measures performance with large number of subscribed OPC items.	✓	✓		
ValueToMessageBox: Very simple Windows Forms application that reads and displays an OPC item value in a message box after the user clicks on a button.	✓	✓		
WcfClient1: Using a Web service provided by the WcfService1 project (under Web	✓	✓		

	Visual C#	Visual Basic	Visual F#	Visual C++
folder), gets and displays a value of an OPC item.				
WcfService1: WcfService1: A simple Web service using WCF technology. Provides a GetData method to read a value of an OPC item. Use WcfClient1 project (under Console folder) to test this Web service.	✓	✓		
WebApplication1: The simplest ASP.NET Web application for OPC "Classic". Reads and displays an OPC item value.	✓	✓		
WebService1: A simple Web service using ASMX technology. Provides "Hello World" method to read a value of an OPC item.	✓	✓		
WindowsFormsApplication1: The simplest Windows Forms application. Reads and displays an OPC item value on a form. This is what you get if you follow the steps described in Quick Start for QuickOPC.NET.	✓	✓		
WpfApplication1: The simplest WPF application for OPC "Classic". Reads and displays an OPC item value.	✓			
XmlEventLogger: Logs OPC Alarms and Events notifications into an XML file.	✓	✓		
XmlLogger: Logs OPC Data Access item changes into an XML file.	✓	✓		
WindowsService1: A Windows Service that subscribes to items from the simulation server, and logs their changes into a file.	✓	✓		

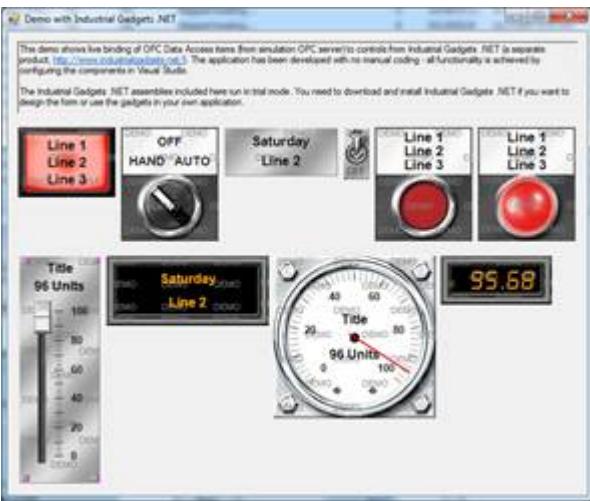
## 17.1.2 Examples for OPC Unified Architecture (OPC-UA)

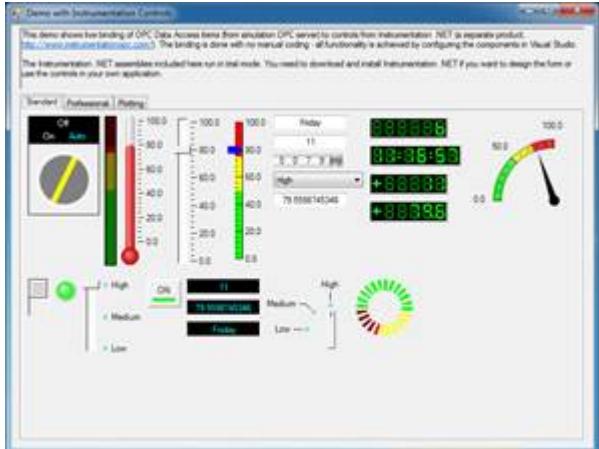
	Visual C#	Visual Basic	Visual F#	Visual C++
EasyOpcUADemo: This is a source of the Demo application for OPC-UA that ships with the QuickOPC-UA product. The application shows the most important product functions, including event-based subscriptions.	✓	✓		
UAConsoleApplication1: The simplest console application. Reads and displays an OPC-UA node value.			✓	
UADocExamples: A collection of OPC-UA console-based examples that illustrate the use of individual objects in the product, and their members. These are the same examples that appear in reference documentation, with an extra control routine that allows the user to choose an example to be performed.	✓	✓	✓	
Note: The amount of available example material may differ between the languages.				
UAConsoleLiveMapping: Creates an object structure for a boiler, describes its mapping into OPC Unified Architecture server using attributes, and then performs the live mapping. Boiler data is then read, written and/or subscribed to using plain .NET object access.	✓	✓		
<b>UALiveBindingDemo:</b> Shows live binding of OPC Unified Architecture information (from sample OPC-UA server) to standard Windows Forms controls (Microsoft). All binding to OPC-UA data is achieved with no manual coding, only by configuring the components in Visual Studio.	✓	✓		

	Visual C#	Visual Basic	Visual F#	Visual C++
				
UASimpleLogToSql: Logs OPC Unified Architecture data changes into an SQL database, using a subscription. Values of all data types are stored in a single SQL_VARIANT column.	✓			
UAWebApplication1: The simplest ASP.NET Web application for OPC-UA. Reads and displays a value of a node in an OPC-UA server.	✓	✓		
UAWpfScreen: Shows how to update WPF controls with dynamic OPC-UA data.	✓			

## 17.1.3 Integration Examples

These examples show how QuickOPC can be integrated with products from other parties.

	Visual C#	Visual Basic	Visual F#	Visual C++
IndustrialGadgetsDemo: Shows live binding of OPC Data Access items (from simulation OPC server) to controls from Industrial Gadgets .NET (a separate product, <a href="http://www.industrialgadgets.net/">http://www.industrialgadgets.net/</a> ). The application has been developed with no manual coding - all functionality is achieved by configuring the components in Visual Studio.	✓	✓		
				
InstrumentationControlsDemo: Shows live binding of OPC Data Access items	✓	✓		

	Visual C#	Visual Basic	Visual F#	Visual C++
(from simulation OPC server) to controls from Instrumentation .NET (a separate product, <a href="http://www.instrumentationopc.com/">http://www.instrumentationopc.com/</a> ). The binding is done with no manual coding - all functionality is achieved by configuring the components in Visual Studio.				
 <p><b>SymbolFactoryDemo:</b> Shows live binding of OPC Data Access items (from simulation OPC server) to controls from Symbol Factory .NET (a separate product, <a href="http://www.symbolfactory.net/">http://www.symbolfactory.net/</a> ). All controls are bound to a single source item. The application has been developed with no manual coding - all functionality is achieved by configuring the components in Visual Studio.</p> 	✓	✓		

## 17.1.4 Reactive Programming Examples

Examples for reactive programming model need an installation of Microsoft Reactive Extensions for proper building. These extensions are now provided by Microsoft in the form of NuGet packages.

The examples only contain references to the additional packages needed, and not the packages themselves. With the help of automatic NuGet Package Restore feature (<http://docs.nuget.org/docs/reference/package-restore>), when enabled, the missing packages will automatically be downloaded and installed when you first build the project that references them.

In Visual Studio 2013 or 2015, this works "out of the box" (with default settings). Visual Studio 2012 ships with an earlier version of NuGet Package Manager which does not have the automatic package restore feature. If building the project in Visual Studio 2012 fails due to missing NuGet packages, follow the steps below:

1. In Visual Studio 2012, select command Tools -> Extensions and Updates.

2. On the left side of the "Extension and Updates" dialog, select Updates -> Visual Studio Gallery. A "NuGet Package Manager" entry will appear in the list of updates, with an "Update" button (on the right side of the dialog, the "New Version" will be indicated as 2.8 or later).
3. Press the Update button and go through the process. You will probably have to restart the Visual Studio in the end.
4. Rebuild the solution then, packages will be restored.

	Visual C#	Visual Basic	Visual F#	Visual C++
ReactiveDocExamples: A collection of console-based examples for reactive programming model that illustrate the use of individual objects in the product, and their members. These are the same examples that appear in reference documentation, with an extra control routine that allows the use to choose an example to be performed.	✓			

## 17.1.5 LINQPad Examples

We have created and tested LINQPad examples with LINQPad 4. They should work well under LINQPad 5 as well.

You will find LINQPad examples under the CSharp, FSharp and VBNET directories in the examples, always under the LINQPad4 subdirectory. They are available in form of LINQPad sample library (.zip file). This allows you conveniently see the examples organized in a hierarchical structure, and select them quickly. In order to use the sample library from LINQPad, switch to the "Samples" tab in the lower left part of LINQPad, click "Download/import more samples..." in the tree view, and then click the "Browse..." link in the bottom left corner, and navigate to the .zip file from QuickOPC.

In addition, the same LINQPad examples are included with the NuGet packages of QuickOPC. This means that whenever you reference one of the QuickOPC NuGet packages in LINQPad, LINQPad will automatically add the QuickOPC examples under its "Samples" tab.

## 17.2 COM Examples

The examples are provided to cover wide range of tools and languages used in development of industrial applications.

For OPC Data Access, the examples show:

- reading and writing OPC items,
- working with multiple items at once,
- error handling,
- subscribing for item changes, and managing subscriptions,
- getting property values,
- browsing for OPC servers, branches and leaves, and properties,
- obtaining information about OPC servers,
- processing events,
- user interface for browsing,
- and more.

For OPC Alarms and Events, the examples show:

- subscribing to events,
- event filtering,
- error handling,

- subscribing for item changes, and managing subscriptions,
- browsing for OPC servers, areas and sources,
- querying for event categories, conditions and attributes,
- obtaining information about OPC servers,
- processing events,
- acknowledging conditions,
- refreshing event subscriptions,
- and more.

For OPC Unified Architecture, the examples show:

- reading and writing attributes of OPC nodes,
- working with multiple nodes/attributes at once,
- error handling,
- subscribing for monitored item changes, and managing subscriptions,
- discovering OPC servers,
- browsing for OPC nodes,
- processing events,
- user interface for browsing,
- and more.

Following categories of COM examples are available , and corresponding shortcuts exist in the Start menu or the Launcher application:

- **Free Pascal Examples (Lazarus) (Section 17.2.1)**
- **JScript Examples (IE, WSH) (Section 17.2.2)**
- **Object Pascal Examples (Delphi) (Section 17.2.3)**
- **Perl Examples (Section 17.2.4)**
- **PHP Examples (Section 17.2.5)**
- **Portable C++ Examples**
- **Python Examples (Section 17.2.7)**
- **REALbasic (Xojo) Examples (Section 17.2.8)**
- **VBA Examples in Excel (Section 17.2.9)**
- **VBScript Examples (ASP, IE, WSH) (Section 17.2.10)**
- **Visual Basic Examples (VB 6.0) (Section 17.2.11)**
- **Visual C++ Examples**
- **Visual FoxPro Examples (Section 17.2.13)**
- **Xbase++ Examples**

## 17.2.1 Free Pascal Examples (Lazarus)

The examples are currently provided for Lazarus 1.6 (FPC 3.0).

- UADocExamples: Contains all Free Pascal examples for OPC-UA that are given in the Reference documentation.

## 17.2.2 JScript Examples (IE, WSH)

### JScript Examples in IE

The examples here run inside Internet Explorer, i.e. directly on client. No Web server is required. The EasyOPC-DA component must be installed on the client side, and the OPC server(s) must be accessible locally or via DCOM from the computer where EasyOPC-DA component resides.

- **ReadAndDisplayValue\_JScript.htm**: Reads and displays an OPC item value.

## JScript Examples in WSH

The examples here run in Windows Script Host (WSH), e.g. from Windows Command Prompt.

- **DocExamples** folder: Contains all JScript examples for OPC Classic that are given in the Reference documentation.
- **ReadAndDisplayValue.js**: Reads and displays an OPC item value.
- **UADocExamples** folder: Contains all JScript examples for OPC-UA that are given in the Reference documentation.

## 17.2.3 Object Pascal Examples (Delphi)

The examples in Object Pascal are currently provided for Delphi XE7. Although you may not be able to directly load the projects into earlier versions of Delphi, the actual source code is likely to work just well with no or minor modifications.

We have also briefly tested the examples with Delphi XE8 and Delphi 10 Seattle.

- **DocExamples**: Contains all Delphi examples for OPC Classic that are given in the Reference documentation.
- **Imports**: Contains Delphi components created by importing QuickOPC type libraries.
- **ReadAndDisplayValue**: The simplest form. Reads and displays an OPC item value.
- **UADocExamples**: Contains all Delphi examples for OPC-UA that are given in the Reference documentation.

## 17.2.4 Perl Examples

The Perl examples were tested in ActivePerl v5.14.2.

- **ReadAndDisplayValue.pl**: The simplest console application for OPC "Classic". Reads and displays an OPC item value.
- **SubscribeToValue.pl**: Subscribes to changes of an OPC Classic item, and displays the values.
- **UAReadAndDisplayValue.pl**: The simplest console application for OPC UA. Reads and displays an OPC item value.
- **UASubscribeToValue.pl**: Subscribes to changes of an OPC UA monitored item, and displays the values.

## 17.2.5 PHP Examples

The examples were tested in PHP v5.6, under command-line interpreter and Internet Information Server (IIS). The examples use the COM extension for PHP. This extension can be enabled e.g. in following ways:

- a) Add following to the php.ini file:

```
extension=ext/php_com_dotnet.dll
```

- b) Use following in the php.ini file:

```
extension_dir = "ext"  
extension=php_com_dotnet.dll
```

- c) From IIS, under Internet Information Services (IIS) Manager, by PHP Manager: Use

```
[PHP_COM_DOTNET]
```

extension=php\_com\_dotnet.dll

## PHP Console Examples

In order to run the command-line examples, use

PHP.EXE -f "filename"

For easier troubleshooting of the command-line scripts, it is recommended that you enable the display\_error option, e.g. in one of the following ways:

- a) Use the following on the command line:

-d display\_errors

- b) Add following to the php.ini file:

display\_errors = On

- c) Copy the file php.ini-development over your php.ini file.

The provided examples are:

- **Console\DocExamples** folder: Contains all command-line PHP examples for OPC Classic that are given in the Reference documentation.
- **Console\UADocExamples** folder: Contains all command-line PHP examples for OPC-UA that are given in the Reference documentation.

## PHP Web Examples

In order to run the Web-based examples, you need to configure an IIS virtual directory that points to the physical directory where the examples are installed.

Examples in this folder are Web server scripts: the page, including OPC data, is generated on the server. The EasyOPC component must be installed on the server side, and the OPC server(s) must be accessible locally or remotely from the computer where EasyOPC component resides.

- **Web\ReadAndDisplayValue.php**: The simplest Web application. Reads and displays an OPC item value.
- **Web\ReadAndDisplayMultipleItems.php**. Reads four values at once and displays their contents. Aimed at showing how PHP can pass input and output arguments that are VARIANT arrays.
- **Web\UADocExamples** folder: Contains all Web server PHP examples for OPC-UA that are given in the Reference documentation.

## 17.2.6 Portable C++ Examples

*Note: These examples are not automatically installed with the product in full. Instead, a shortcut is provided which allows you to download a ZIP file containing the examples.*

These examples are designed to only use C++ language features and libraries that are portable across compilers. Note that does not mean you can run QuickOPC on anything else than Windows - it just means you can use various compilers.

- **VoleReadItemValue**: Shows how to read value of a single OPC item, and display it, using STLSoft and VOLE libraries.
- **VoleReadMultipleItems**: Shows how to read 4 items at once, and display their values, timestamps and qualities, using STLSoft and VOLE libraries.

Note that due to the number of files in the STLSoft and VOLE libraries, and their size, the installation program only places a download link to the corresponding directory. The link allows you to download a ZIP file with the examples from the Web.

## 17.2.7 Python Examples

The examples were tested in Python 3.4.2, with pywin32-219 extension (Build 219).

- **DocExamples** folder: Contains all Python examples for OPC Classic that are given in the Reference documentation.
- **UADocExamples** folder: Contains all Python examples for OPC-UA that are given in the Reference documentation.

## 17.2.8 REALbasic (Xojo) Examples

*Note: We do not regularly update these examples with new versions. You may have to modify the code to use newer versions.*

The REALbasic (Xojo) examples were originally created in REAL Studio 2010r5, and converted to REAL Studio 2011r1.1, and later versions.

- **Xojo2014\ReadAndDisplayValue.xojo\_binary\_project**: The simplest form, for OPC "Classic". Reads and displays an OPC item value.
- **Xojo2014\UAReadAndDisplayValue...**: The simplest form, for OPC UA. Reads and displays an OPC item value.

## 17.2.9 VBA Examples in Excel

VBA stands for Visual Basic for Applications. The examples were developed and tested in Excel 2007 or 2010, but are saved in Excel 2003 format for compatibility.

- **ReadAndDisplayMultipleValues.xls**: Reads multiple OPC Classic item values, and stores them in cells of a worksheet.
- **ReadAndDisplayValue.xls**: The simplest Excel VBA example for OPC "Classic". Reads an OPC item value and stores it in a cell of a worksheet.
- **ReadAndWriteValue.xls**: Shows how to read or write an OPC Classic item value upon press of a button.
- **SubscribeToMultipleItems.xls**: Shows how to subscribe to and unsubscribe from multiple OPC Classic items, and how to continuously update the worksheet cells with their values.
- **UAClientCertificateParameters.xls**: Sets the application name for the client certificate.
- **UADataDialog.xls**: Shows how a user can interactively select an OPC-UA endpoint and a data node.
- **UAReadAndDisplayValue.xls**: The simplest Excel VBA example for OPC UA. Reads an OPC UA value and stores it in a cell of a worksheet.
- **UAReadWriteValue.xls**: Shows how to read or write an OPC UA value upon press of a button.
- **UASubscribeToMultiple.xls**: Shows how to subscribe to and unsubscribe from multiple OPC UA monitored items, and how to continuously update the worksheet cells with their values.
- **UASubscribeToMultiple2.xls** and **UASubscribeToMultiple3.xls**: As UASubscribeToMultiple.xls, but allows sheet editing while being subscribed, using different approaches. The UASubscribeToMultiple3.xls example also shows how to specify an absolute deadband.

## 17.2.10 VBScript Examples (ASP, IE, WSH)

### VBScript Examples in ASP

These VBScript examples run in Internet Information Server (IIS). In order to run the examples, you need to configure an IIS virtual directory that points to the physical directory where the examples are installed.

Examples in this folder are Web server scripts: the page, including OPC data, is generated on the server. The EasyOPC component must be installed on the server side, and the OPC server(s) must be accessible locally or remotely from the computer where EasyOPC component resides.

The server must be in the Web Server (IIS) role, and in it, make sure you have the "Web Server -> Application Development -> ASP" role service.

- **ReadAndDisplayValue\_VBScript.asp:** The simplest Web application. Reads and displays an OPC item value.

## VBScript Examples in IE

The examples here run inside Internet Explorer (IE), i.e. directly on client. No Web server is required. The EasyOPC component must be installed on the client side, and the OPC server(s) must be accessible locally or remotely from the computer where EasyOPC component resides.

- **ReadAndDisplayValue\_VBScript.htm:** Reads and displays an OPC item value.

## VBScript Examples in WSH

The examples here run in Windows Script Host (WSH), e.g. from Windows Command Prompt.

It is possible to use CScript (console output) or WScript (window output) to run the examples. Some examples give longer output, and CScript is better for them as you do not have to confirm each "line" of output by pressing a button as in WScript.

- **APISimplifier.wsc, ReadUsingSimplifier.vbs:** Shows how to create and use a component that provides domain-specific API to QuickOPC.
- **ReadAndDisplayValue.vbs:** Reads and displays an OPC item value.
- **DocExamples** folder: Contains all VBScript examples for OPC Classic that are given in the Reference documentation.
- **UADocExamples** folder: Contains all VBScript examples for OPC-UA that are given in the Reference documentation.

## 17.2.11 Visual Basic Examples (VB 6.0)

The examples are for Visual Basic 6.0.

- **ReadAndDisplayValue:** The simplest form. Reads and displays an OPC item value. This is what you get if you follow the steps described in Quick Start for QuickOPC-COM.
- **SubscribeMultipleItems:** Subscribes to multiple OPC-DA items, and displays the incoming changes in the list box.
- **UADocExamples:** Contains all VB 6.0 examples for OPC-UA that are given in the Reference documentation.
- **UAReadAndDisplayValue:** The simplest form for OPC-UA. Reads and displays an OPC-UA node value.
- **WriteMultipleItemValues:** Writes different values to multiple OPC Data Access items at once.

## 17.2.12 Visual C++ Examples

Examples in Visual C++ are currently provided for Microsoft Visual Studio 2012, with a possible conversion to Microsoft Visual Studio 2013 or 2015.

- **MFC\EasyOPCDA\Demo:** This is a clone of the Demo application that ships with the QuickOPC-COM product.
- **Win32\DumpAddressSpace:** Dumps OPC server's address space recursively to the console.
- **Win32\RandomReads:** Reads random number of random items in a random fashion. This example can serve

as kind of a stress test.

- **Win32\ReadMultipleItems:** Reads multiple items, and displays the results.
- **Win32\ReadMultipleItemsXml:** Reads multiple items from an OPC XML-DA server, and displays the results.
- **Win32\SubscribeMultipleItems:** Subscribes to changes of multiple items and display the value of the item with each change.
- **Win32\UADocExamples:** Contains all C++ examples for OPC-UA that are given in the Reference documentation.
- **Win32\WriteMultipleItemValues:** Writes values into multiple items at once.

Note that you can also use the "Portable C++ Examples" (see elsewhere in this document) in Visual C++.

## 17.2.13 Visual FoxPro Examples

*Note: We do not regularly update these examples with new versions. You may have to modify the code to use newer version numbers.*

The examples were tested in Visual FoxPro 9.0 with Service Pack 2.

- **ReadItemValue:** Reads a value of an OPC Classic item and displays it in a message box.
- **UAReadItem:** Reads a value of an OPC UA node and displays it in a message box.

## 17.2.14 Xbase++ Examples

*Note: We do not regularly update these examples with new versions. You may have to modify the code to use newer version numbers.*

The examples were tested with Xbase++ 1.90 SL1.

- **UAReadValue.prg:** Reads a values of an OPC UA node displays it.
- **WriteAndReadValue.prg:** Writes a value into an OPC Classic item, reads it back and displays it.

## 18 Reference

The reference is provided online. The printable documentation (PDF booklet) only contains introductory chapters for each kind of reference material, and links to the online content.

### 18.1 .NET Assemblies

#### List of .NET assemblies

- [OpcLabs.BaseLib Assembly](#)
- [OpcLabs.BaseLibForms Assembly](#)
- [OpcLabs.EasyOpcClassic Assembly](#)
- [OpcLabs.EasyOpcForms Assembly](#)
- [OpcLabs.EasyOpcUA Assembly](#)
- [OpcLabs.OpcComplexEventProcessing Assembly](#)

### 18.2 COM Type Libraries

#### List of COM type libraries

- [OpcLabs\\_BaseLib ActiveX DLL](#)
- [OpcLabs\\_BaseLibForms ActiveX DLL](#)
- [OpcLabs\\_EasyOpcClassic ActiveX DLL](#)
- [OpcLabs\\_EasyOpcForms ActiveX DLL](#)
- [OpcLabs\\_EasyOpcUA ActiveX DLL](#)

The links above may not work in Microsoft Help Viewer (Visual Studio) due to a bug in the software we use to create the documentation and help. Following links to the Web-based (online) help can be used instead:

- [OpcLabs.BaseLib ActiveX DLL](#)
- [OpcLabs.BaseLibForms ActiveX DLL](#)
- [OpcLabs.EasyOpcClassic ActiveX DLL](#)
- [OpcLabs.EasyOpcForms ActiveX DLL](#)
- [OpcLabs.EasyOpcUA ActiveX DLL](#)

### Usage notes

The COM objects of QuickOPC are based upon the underlying .NET objects. In order to avoid duplicities and mismatches, the documentation is primarily maintained for the .NET objects. The .NET objects are exposed using the "interop" mechanism to the COM world using an automated translation provided by Microsoft. Therefore, a documentation that applies to a .NET type or member that is exposed to COM can be assumed to apply to the corresponding .NET type or member as well.

The bulk of the reference documentation for COM type libraries is generated from the type libraries themselves. This means that only a limited descriptive text (typically, one line) is available with each type or member. You need to look to the reference documentation for the .NET assemblies in order to find the more detailed documentation.

In addition, some languages or COM-based tools do not make direct use of the type libraries, and therefore require some additional effort to use the COM objects - such as knowing their ProgIDs, dealing with interface IDs (GUIDs), etc.

Below are some rules and conventions that help with using the reference documentation for COM.

## Determining the assembly name

This is not usually needed, but may be useful if you are writing a setup program for your application, and want to deploy only the .NET assemblies that are needed to run the COM objects that you are actually using.

The .NET assembly has the same name as its corresponding type library, except that the .DLL extension is used instead of .TLB.

For example, in order to use types from the **OpcLabs.EasyOpcUA.tlb** type library, the **OpcLabs.EasyOpcUA.dll** assembly is needed. Note that due to assembly dependencies, some more assemblies may be needed as well.

## Determining the name of the corresponding .NET object

For a COM object, you need to locate the corresponding .NET object in order to find more detailed documentation related to the object.

The .NET objects that corresponds to the COM object has the same simple name, but is qualified with some namespace. Use the Search functionality of the help system to find the reference documentation for the .NET object, typing in the name of the COM object you are interested in.

For example, the [EasyUAClient](#) COM object corresponds to [OpcLabs.EasyOpc.UA.EasyUAClient](#) object in .NET.

## Determining a CLSID of the object

The CLSID may be needed to create a COM object in some languages (e.g. C++).

In order to determine the CLSID of the COM object, first find the reference documentation for its corresponding .NET object. The CLSID is then listed as the [GuidAttribute](#) of this .NET object.

## Determining a ProgID of the object

The CLSID may be needed to create a COM object in some languages (e.g. VBScript).

In order to determine the CLSID of the COM object, first find the reference documentation for its corresponding .NET object. The ProgID is then same as the fully qualified name of that type (i.e. including the namespace). For example, for the [EasyUAClient](#) COM object, the ProgID is "[OpcLabs.EasyOpc.UA.EasyUAClient](#)".

## Determining the default COM interface

The default COM interface is the interface that implements the members of the COM object. You need to locate the reference documentation for the default interface to find more detailed information related to the object.

The default COM interface has the same name as the COM object, but is preceded with an underscore ('\_'). In order to view the reference documentation for the default COM interface, first find the reference documentation for the .NET object that corresponds to the COM object you are interested in. On the Overview page of that .NET object, you will see the list of interfaces that the object implements. Click on the interface that starts with an underscore and has the same name as the object.

For example, the default COM interface for [EasyDAClient](#) object is named [\\_EasyDAClient](#).

## Determining the IID of the default COM interface

The IID may be needed to access the COM object in some languages (e.g. C++).

In order to determine the IID of the default COM interface, first find the reference documentation for this interface.

The IID is then listed as the GuidAttribute of this interface.

## 18.3 Format Strings

Formatting converts objects to strings, usually for display purposes. QuickOPC follows the design of formatting in the .NET Framework ([https://msdn.microsoft.com/en-us/library/26etazsy\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/26etazsy(v=vs.110).aspx) ).

Certain QuickOPC objects implement the [IFormattable](#) interface, and can therefore participate in formatting schemes provided by the .NET Framework – e.g. their formatting can be controlled by corresponding format specifiers in the [String.Format](#) method overloads. This appendix describes the formatting of QuickOPC types that have this kind of support.

In this part of the Reference:

- [OperationResult Format Strings](#)
- [ValueResult Format Strings](#)
- [DAQuality Format Strings](#)
- [DAVtq Format Strings](#)
- [DAVtqResult Format Strings](#)
- [UAAttributeData Format Strings](#)
- [UAAttributeDataResult Format Strings](#)

## 19 Index

.bin, 423-424  
.config.xml, 427-429  
.Contracts.dll, 35  
.dll, 33-35 , 36-37  
.exe, 36-37  
.NET, 16 , 15-16  
.NET Assemblies, 481  
.NET Demo Applications, 37  
.NET Demo Applications for OPC Classic, 37-38  
.NET Demo Applications for OPC UA, 38-40  
.NET Examples, 468 , 41  
.NET Framework, 16  
.REG, 423  
.tlb, 36-37  
\_XXXX, 63  
32-bit and 64-bit Code, 442  
64-bit Platforms, 441-442  
AbstractMapping, 326-327  
AcceptAnyCertificate, 97-98

**AEGAttributeDialog**, 388-389  
**AEGAttributeElement**, 85  
**AEBrowseParameters**, 260-263  
**AECCategoryConditionDialog**, 387-388  
**AECCategoryDialog**, 386-387  
**AECCategoryElement**, 85 , 417  
**AECCategoryElementCollection**, 434-435 , 263-266  
**AEConditionElement**, 85 , 417  
**AEConditionState**, 252-254 , 434-435  
**AEEEventData**, 285-290 , 434-435 , 462-463  
**AEEEventDataPayload**, 462-463  
**AENodeDescriptor**, 85-86  
**AENodeElement**, 260-263 , 85  
**AENodeElementCollection**, 260-263  
**AENotificationObservable**, 459-460 , 371  
**AENotificationPayload**, 462-463  
**AESubconditionElement**, 85  
**AESubscriptionFilter**, 272-276  
**AESubscriptionFilterType**, 268-272  
**AESubscriptionParameters**, 272-276  
**AEUtilities**, 91  
**Aggregates**, 94  
**Alarms – Shelving and Unshelving**, 294  
**Alarms&Events Extensions**, 417  
**AlarmsAndEvents**, 63  
**AllFields**, 299-302  
**AllowCertificatePrompt**, 98-101  
**AllowedMessageSecurityModes**, 97  
**AllowUserAcceptCertificate**, 97-98  
**Alternate Access Methods**, 415-417  
**Always test the Exception property before accessing the actual result (Value, Vtq, or AttributeData property)**, 445  
**Always test the HasValue property before accessing DAVtq.Value or UAAttributeData.Value**, 445  
**AnchorElementType**, 381-384 , 400-406  
**AnonymousTokenInfo**, 251  
**Any CPU**, 441-442  
**app.config**, 427-429  
**App\_Web\_OpcLabs.EasyOpcClassicRaw.amd64**, 33-35  
**App\_Web\_OpcLabs.EasyOpcClassicRaw.amd64.dll**, 420 , 53

**App\_Web\_OpcLabs.EasyOpcClassicRaw.x86**, 33-35  
**App\_Web\_OpcLabs.EasyOpcClassicRaw.x86.dll**, 420 , 53  
**Application Configuration**, 427-429 , 97-98  
**Application Deployment**, 420 , 467  
**ApplicationCertificateStore**, 98-101  
**ApplicationCertificateSubject**, 427-429 , 98-101  
**ApplicationElement**, 389-393  
**ApplicationName**, 427-429 , 98-101  
**ApplicationUriString**, 427-429 , 98-101  
**AreaElements**, 385-386  
**Argument Objects**, 84  
**ArgumentException**, 444-445 , 69-80 , 444  
**arguments**, 350-351 , 285-290 , 306-310 , 227-228 , 228-229 , 180-186 , 186-200  
**Arguments and Results**, 350-351  
**Arguments Path**, 351-352  
**ArgumentsPath**, 363-364 , 364 , 366 , 353-354 , 364 , 364-365  
**Array**, 437-439 , 439-440  
**ASP**, 58 , 16 , 57 , 478-479  
**ASP.NET**, 16  
**assemblies**, 423-424 , 455 , 33-35 , 31-32  
**Assemblies (.NET)**, 420  
**Assemblies and Deployment Elements**, 418-419  
**Assemblies\net452**, 33-35 , 35-36  
**Assembly Referencing with NuGet**, 54-55  
**Assembly Referencing with Visual Studio Toolbox**, 54  
**Assembly Registration tool**, 420  
**Attribute**, 302-306  
**Attribute IDs**, 95  
**AttributeData**, 228-229 , 364  
**AttributeElement**, 388-389  
**AttributeElements**, 388-389  
**Attributeld**, 95 , 323-324 , 388-389  
**Attributelds**, 388-389  
**AttributeReadParametersTemplate**, 352  
**Attributes for Live Mapping**, 320-321  
**AttributeSubscribeParametersTemplate**, 352  
**AttributeWriteParametersTemplate**, 352  
**AutoConditionRefresh**, 306  
**Automated deployment with Production installer**, 423-424

**Automated deployment, roll your own, 421-422**  
**Automatic Read (On Form Load), 362**  
**Automatic Subscription (With the Form), 362**  
**Automatically Get an OPC Property (On Form Load), 366**  
**AutomaticPublishingFactor, 186-200**  
**AutoRead, 362 , 366 , 359**  
**AutoSubscribe, 362 , 366 , 359 , 365**  
**AutoWrite, 359**  
**BaseDataType, 94**  
**BaseObjectType, 94**  
**Best Practices, 444**  
**Bin, 421**  
**BinaryFormatter, 434-435**  
**binder, 346 , 346**  
**BinderBase, 360-361**  
**Binders and Binding Providers, 346**  
**binding, 343-344**  
**binding bag, 343-344**  
**Binding Bags, 346**  
**Binding Collection Editor, 355-357**  
**Binding Extender, 346 , 343-344**  
**binding group, 347 , 345-346**  
**Binding Group Collection Editor, 357-358**  
**Binding Groups, 347**  
**binding operation, 345-346 , 343-344**  
**Binding Operations, 345**  
**binding provider, 346 , 343-344**  
**binding source, 345-346 , 343-344**  
**binding target, 345-346**  
**binding template, 343-344**  
**BindingBag, 355**  
**BindingErrors, 360-361**  
**BindingExtender, 362 , 346 , 355-357 , 346 , 347-348 , 355 , 360-361 , 346-347 , 355 , 51-52 , 358 , 354 , 361**  
**BindingGroup, 362 , 366 , 366 , 365**  
**BindingGroups, 362 , 366 , 357-358 , 366 , 365**  
**BindingOperations, 345 , 366 , 362-363 , 365-366 , 359 , 365 , 365**  
**Bindings, 345-346**  
**Boolean, 437-439 , 439-440 , 94 , 353-354**

**boxing, 442**  
**Browse Path and Node Id Resolution, 324-326**  
**Browse Paths, 87**  
**Browse Paths for OPC Classic, 87**  
**Browse Paths in OPC-UA, 87-88**  
**BrowseAccessPaths, 164-165**  
**BrowseAreas, 260-263**  
**BrowseBranches, 161-164**  
**BrowseControlKinds, 407-408 , 408-410**  
**BrowseDataNodes, 173-180**  
**BrowseDataVariables, 173-180**  
**BrowseDirections, 173-180**  
**BrowseEventSources, 295**  
**BrowseFailure, 407-408 , 408-410**  
**BrowseLeaves, 161-164**  
**BrowseMethods, 173-180**  
**BrowseName, 324-326**  
**BrowseNodes, 295 , 161-164 , 260-263 , 173-180**  
**BrowseNotifiers, 295**  
**BrowseObjects, 173-180**  
**BrowsePath, 324-326 , 87 , 87-88 , 87**  
**BrowseProperties, 165-166 , 173-180**  
**BrowseServers, 160-161 , 258-260**  
**BrowseSources, 260-263**  
**BrowseVariables, 173-180**  
**Browsing for Event Sources, 295**  
**Browsing for Information, 159-160 , 258 , 294**  
**Browsing for Notifiers, 295**  
**Browsing for OPC Classic Access Paths, 164-165**  
**Browsing for OPC Classic Nodes (Branches and Leaves), 161-164**  
**Browsing for OPC Classic Properties, 165-166**  
**Browsing for OPC Classic Servers, 160-161**  
**Browsing for OPC Nodes (Areas and Sources), 260-263**  
**Browsing for OPC Servers, 258-260**  
**Browsing for OPC UA Nodes, 173-180**  
**Building and Running a First StreamInsight Application with OPC, 451-452**  
**Button, 366 , 365**  
**Byte, 437-439 , 439-440**  
**ByteString, 439-440**

C#, 16, 41, 17-18, 18-20, 21-22  
C++, 58, 477-478, 479-480  
Callback, 229-231  
Calling Methods in OPC-UA, 231-248  
CallMethod, 231-248, 292  
CallMultipleMethods, 231-248  
CanBeNull, 35  
catch, 81-82, 82  
CategoryElement, 386-387  
CategoryElements, 386-387  
CategoryId, 388-389, 387-388, 386-387, 263-266  
CategoryIds, 386-387  
CEP, 449  
certificate, 28-30  
certification, 425  
Change Color According to Status, 363-364  
ChangeDataChangeSubscription, 200-209  
ChangeEventSubscription, 276-279  
ChangeItemSubscription, 200-209  
ChangeMonitoredItemSubscription, 200-209, 306  
ChangeMultipleDataChangeSubscriptions, 200-209  
ChangeMultipleItemSubscriptions, 200-209  
ChangeMultipleMonitoredItemSubscriptions, 200-209, 306  
Changing Existing Subscription, 200-209, 276-279, 306  
Clamped, 143-153  
Classic OPC on 64-bit Systems, 442  
ClientParameters, 248-251, 291  
ClientSelector, 369-370, 370, 371, 371  
CloseAll, 444, 435  
CLS, 437-439, 439-440  
CLSID, 160-161, 258-260, 442, 85-86, 481-483  
ClidString, 85-86  
coclass, 36-37  
Code Contracts, 35  
Code Contracts assemblies, 455, 31-32  
code signing certificate, 28-30  
CodeBits, 84  
Color, 353-354

**Coloring Conventions, 63-64**  
**COM, 16 , 15-16**  
**COM Automation, 437-439 , 16**  
**COM Components, 35-36**  
**COM Components and Development Libraries, 420**  
**COM Components for OPC Classic, 58**  
**COM Components for OPC UA, 58**  
**COM Demo Applications, 40-41**  
**COM Examples, 474-475 , 41**  
**COM interop, 36-37**  
**COM Type Libraries, 481-483**  
**COM/DCOM, 442**  
**comcomponents, 423-424**  
**COMException, 80-81**  
**Common Concerns, 27**  
**Common Functionality in Browsing Dialogs and Controls, 410-411**  
**Common Naming Conventions, 63**  
**Common Payload Characteristics, 461**  
**CommonDialog, 372-373**  
**COMODO, 28-30**  
**CompletesAsynchronously, 143-153**  
**Complex Event Processing, 455 , 449**  
**COMPONENTS, 423-424**  
**Components and Objects, 64-66**  
**ComposedBindingBag, 355-357**  
**CompositeConnectivity, 347-348 , 51-52 , 354**  
**Computational Objects, 66-67**  
**Computer Browser Dialog, 373-376 , 384-385**  
**ComputerBrowserDialog, 373-376 , 384-385**  
**Conclusions, 466**  
**Condition, 84**  
**ConditionElement, 387-388**  
**ConditionName, 387-388**  
**Conditions - Disabling/Enabling, and Applying Comments, 292**  
**ConfigurationSources, 427-429 , 97-98**  
**Confirm, 292-294**  
**Connected, 312**  
**Connection-less Approach, 69**  
**ConnectionState, 312**

**Connectivities**, 347-348  
**connectivity**, 347-348  
**console**, 16  
**constructor**, 440-441  
**contextual help**, 44  
**Conventions**, 62  
**Converter**, 363-364 , 343-344 , 352-353  
**Create**, 66-67 , 369-370 , 370 , 368-369 , 371 , 371  
**CreateAnonymousIdentity**, 251  
**CreateKerberosIdentity**, 251  
**CreateUserNameIdentity**, 251  
**CreateX509Certificate**, 251  
**Creating OPC Event Sources**, 459  
**CScript**, 478-479  
**CurrentNodeChanged**, 407-408 , 408-410  
**CurrentNodeDescriptor**, 381-384 , 407-408 , 408-410 , 400-406  
**CurrentNodeElement**, 381-384 , 407-408 , 408-410 , 400-406  
**custom**, 423-424  
**CustomNetworkCredential**, 251  
**D**, 63  
**DA**, 63  
**DABrowseParameters**, 161-164  
**DAClientItemMapping**, 328 , 319-320  
**DAClientItemSource**, 319 , 319-320  
**DAClientMapper**, 327 , 313-319  
**DAClientPropertyMapping**, 328 , 319-320  
**DAClientPropertySource**, 319 , 319-320  
**DAConnectivity**, 347-348 , 51-52 , 17-18 , 352 , 348 , 354  
**DAGroupParameters**, 200-209 , 326-327 , 434-435 , 90-91 , 180-186  
**DAItem**, 320-321 , 328-330 , 328 , 323-324  
**DAItemArguments**, 102-107 , 107-110  
**DAItemChangedObservable**, 457-459 , 369-370 , 459-460 , 368-369  
**DAItemChangedPayload**, 457-459 , 461-462  
**DAItemDescriptor**, 87 , 85-86 , 102-107 , 180-186 , 137-141 , 107-110  
**DAItemDialog**, 377-380  
**DAItemGroupArguments**, 434-435 , 180-186  
**DAItemIdTemplate**, 320-321 , 324-326  
**DAItemMappingOperations**, 328

**DAItemPoint**, 348  
**DAItemValueArguments**, 412-413 , 137-141  
**DAItemVtqArguments**, 412-413  
**DALimitChoice**, 82-83  
**DAMappedNode**, 327-328 , 322  
**DAMappingContext**, 313-319  
**DAMember**, 320-321 , 323-324  
**DANode**, 320-321 , 324-326 , 327 , 323-324  
**DANodeDescriptor**, 87 , 85-86 , 326-327 , 92  
**DANodeElement**, 87 , 161-164 , 85-86 , 85 , 92  
**DANodeElementCollection**, 161-164 , 434-435  
**DAProperty**, 320-321 , 323-324  
**DAPropertyDescriptor**, 85-86 , 92  
**DAPropertyDialog**, 380-381  
**DAPropertyElement**, 165-166 , 85 , 92  
**DAPropertyElementCollection**, 165-166  
**DAPropertyId**, 165-166  
**DAPropertyIds**, 165-166  
**DAPropertyIDSet**, 415-417  
**DAPropertyPoint**, 348  
**DAQualities**, 82-83  
**DAQuality**, 82-83 , 353-354  
**DAQualityChoice**, 82-83  
**DARead**, 320-321  
**DAReadParameters**, 326-327 , 102-107  
**DAStatusChoice**, 82-83  
**DASubscription**, 320-321 , 322  
**Data Access Extensions**, 412  
**data converter**, 352-353  
**Data Objects**, 82  
**Data Type in OPC UA Write**, 153  
**Data Types**, 437  
**Data Types in OPC Classic**, 437-439  
**Data Types in OPC-UA**, 439-440  
**DataAccess**, 63  
**DataChange**, 63  
**DataChangeCallback**, 229-231  
**DataChangeFilter**, 295-298  
**DataChangeNotification**, 445 , 69-80 , 228-229 , 186-200

**DataEventArgs**, 323  
**DataSource**, 102-107  
**DataTypesFolder**, 94  
**WithValue**, 439-440  
**DATE**, 82  
**Date and Time Conventions**, 62  
**DateTime**, 437-439 , 439-440 , 62 , 82  
**DAType**, 320-321 , 327-328 , 322  
**DAUtilities**, 91  
**DAVtq**, 445 , 62 , 412-413 , 319-320 , 434-435 , 227-228 , 368-369 , 102-107 , 83  
**DAVtqPayload**, 461-462  
**DAVtqResult**, 445 , 412-413 , 102-107 , 84-85 , 353-354  
**DAWriteItemValueObserver**, 370  
**DBNull**, 437-439  
**DCOM**, 442  
**DeadbandType**, 186-200  
**Debugging**, 434  
**Debugging the StreamInsight Event Flow**, 452-455  
**Decimal**, 437-439  
**Default**, 90-91  
**default binding group**, 347  
**DefaultBindingGroup**, 357-358  
**DefaultBindingProperty**, 344-345  
**DefaultProperty**, 344-345  
**Deferred**, 327  
**Deferred Mapping**, 327  
**DeferredMapNodeFunction**, 327  
**Delphi**, 16 , 58-62 , 476  
**Demo Applications**, 37  
**Deployment Automation**, 423  
**Deployment Elements**, 420  
**Deployment Methods**, 421  
**Descriptor Objects**, 85-86  
**Design**, 33-35  
**Design Offline**, 17-18 , 20-21  
**Design Online**, 17-18 , 20-21  
**Designer Commands**, 355  
**Designer Verbs**, 355

**Details, 418**  
**Development Libraries (COM), 36-37**  
**Development Models, 50**  
**Development Tools, 16**  
**devlibs, 423-424**  
**Diagnostics, 251-252**  
**DiagnosticsMasks, 251-252**  
**DiagnosticsSummary, 84-85**  
**Dialogs - Responding, 292**  
**Direct License Deployment, 422**  
**Disable, 292**  
**DiscoverApplications, 166-172 , 173**  
**Discovering OPC UA Servers, 166 , 294-295**  
**DiscoverServers, 166-172**  
**discovery URL, 166-172**  
**DiscoveryParameters, 248-251**  
**DiscoveryUriTemplateStrings, 166-172**  
**disk space, 27**  
**DispatcherSynchronizationContext, 440-441**  
**dispinterface, 36-37**  
**Display Errors with ErrorProvider, 364**  
**Display Write Errors, 366**  
**DisplayValue, 84 , 83**  
**Do not block inside OPC event handlers or callback methods, 445-446**  
**Do not catch any exceptions with asynchronous or multiple-operand methods, 444-445**  
**Do not write any code for OPC failure recovery, 444**  
**Documentation and Help , 455-456 , 44**  
**Documentation for Obsoleted Features, 367**  
**Double, 437-439 , 439-440 , 186-200**  
**DXXXXEvents, 63**  
**Easy, 63**  
**EasyAE, 63**  
**EasyAECClient, 66-67 , 69 , 69-80 , 67-68 , 68 , 440-441 , 285-290 , 434-435 , 371 , 371 , 417 , 252 , 248-251 , 291 , 68 , 279-283 , 229-231**  
**EasyAECClientConfiguration, 248-251**  
**EasyAENotificationEventArgs, 285-290 , 434-435 , 371 , 460 , 462-463 , 268-272 , 290-291**  
**EasyDA, 63**  
**EasyDAClient, 445 , 66-67 , 69 , 445-446 , 69-80 , 413 , 414 , 67-68 , 68 , 23-24 , 18-20 , 440-441 , 434-435 , 227-228 , 369-370 , 370 , 102 , 248-251 , 68 , 436-437 , 413 , 413 , 229-231 , 481-483**

**EasyDAClientConfiguration**, 248-251  
**EasyDAItemChangedEventArgs**, 445 , 457-459 , 412-413 , 434-435 , 227-228 , 369-370 , 460 , 180-186 , 353-354  
**EasyDAItemSubscriptionArguments**, 412-413  
**EasyDataChangeNotificationEventArgs**, 251-252  
**EasyOPC "Classic" Library**, 33-35  
**EasyOPC "Classic" Raw Library**, 33-35  
**EasyOPC "Classic" Raw Library (x64)**, 53  
**EasyOPC "Classic" Raw Library (x86)**, 53  
**EasyOPC Extensions for .NET**, 412  
**EasyOPC Forms**, 33-35  
**EasyOPC-UA Library**, 33-35 , 36-37 , 53  
**EasyUA**, 63  
**EasyUAClient**, 445 , 427-429 , 421-422 , 66-67 , 69 , 444 , 429-434 , 69-80 , 435-436 , 67-68 , 435 , 68 , 24-26 , 21-22 , 292 , 440-441 , 306-310 , 434-435 , 292 , 369-370 , 370 , 228-229 , 389-393 , 312 , 102 , 98-101 , 92-93 , 248-251 , 311-312 , 68 , 186-200 , 97-98 , 229-231 , 172-173  
**EasyUAClientConfiguration**, 429-434 , 248-251  
**EasyUADataChangeNotificationEventArgs**, 445 , 434-435 , 369-370 , 228-229 , 460 , 186-200 , 353-354  
**EasyUAEventArgs**, 306  
**EasyUAEventArgs**, 306-310  
**EasyUAEventHandler**, 310-311  
**EasyUAMonitoredItemEventArgs**, 95 , 95-96 , 221-227 , 186-200 , 310-311  
**EasyUAServerConditionChangedEventArgs**, 312  
**EasyUASubscriptionChangeArguments**, 200-209  
**EasyXXClient**, 90-91  
**Element Objects**, 85  
**embedded**, 423-424  
**Embedded assemblies**, 33-35  
**Enable**, 292  
**EndpointDescriptor**, 228-229 , 396-400 , 312 , 362-363  
**EngineParameters**, 248-251 , 291  
**Error**, 364 , 366  
**Error Handling**, 80-81 , 444  
**Error Handling in Live Binding**, 360-361  
**Error Handling in the Mapper Object**, 338-339  
**ErrorCode**, 461 , 80-81 , 360-361 , 460  
**ErrorColor**, 353-354  
**ErrorMessage**, 364 , 460  
**ErrorProvider**, 364 , 366 , 344-345  
**Errors and Multiple-Element Operations**, 81-82

**Errors in OPC Sequences, 460**  
**Errors in Subscriptions, 82**  
**Event Flow Debugger, 452-455**  
**event link, 345-346**  
**Event Logging, 429**  
**Event Logging for OPC "Classic", 429**  
**Event Logging for OPC UA, 429-434**  
**Event Pull Mechanism, 69-80**  
**event source, 343-344**  
**Event Sources, 346-347**  
**EventCallback, 310-311**  
**EventData, 285-290 , 306-310 , 460 , 462-463**  
**EventFilter, 295-298**  
**EventLinkingFailure, 360-361**  
**EventNotification, 69-80 , 306-310 , 295-298**  
**EventSources, 295**  
**EventTypes, 386-387**  
**EventTypesFolder, 94**  
**Example Walkthrough, 457-459**  
**Examples, 466 , 468 , 455 , 41 , 41**  
**Examples for OPC "Classic" (OPC-DA, OPC XML-DA and OPC-A&E), 468-471**  
**Examples for OPC Unified Architecture (OPC-UA), 471-472**  
**Excel, 16 , 478**  
**Exception, 445 , 444-445 , 80-81 , 360-361 , 81-82 , 460 , 82 , 285-290 , 306-310 , 227-228 , 369-370 , 228-229 , 371 , 312 , 84-85 , 353-354 , 143-153**  
**ExecuteRead, 360**  
**ExecuteSubscribe, 360**  
**ExecuteWrite, 360**  
**Exit Codes, 423-424**  
**ExpandedNodeId, 439-440**  
**ExpandedText, 94**  
**Extender Properties, 355**  
**ExtenderProvider, 364 , 366 , 364 , 344-345**  
**ExtensionObject, 439-440**  
**Extensions for OPC Items, 413**  
**Extensions for OPC Properties, 413**  
**Extensions on Helper Types, 418**  
**F#, 41**  
**FAILED, 80-81**

failure recovery, 444 , 435-436  
**FastestAutomaticPublishingInterval**, 186-200  
**FieldResults**, 306-310  
**FilterOperands**, 302-306  
**FilterOperator**, 302-306  
**FindApplications**, 166-172  
**FindEventCategory**, 417  
**FindEventCondition**, 417  
**FlagBits**, 84  
**Float**, 439-440  
**FolderType**, 94  
**format**, 87 , 88-90 , 483  
**Format Strings**, 483  
**FoxPro**, 16 , 480  
**FPC**, 475  
**Free Pascal**, 475  
**Free Pascal Examples (Lazarus)**, 475  
**FromInt32**, 90-91  
**FromType**, 91  
**full installer**, 423-424  
**Fundamentals**, 46 , 455  
**Fusion**, 420  
**GAC**, 423-424 , 420 , 33-35 , 28-30  
**GDS**, 172-173  
**GdsEndpointDescriptor**, 172-173  
**Generalized OPC UA Discovery**, 166 , 173  
**Generic Access**, 413 , 414  
**Generic OPC Browsing Dialog**, 381-384 , 389  
**Generic OPC Classic Browsing Control**, 407-408  
**Generic OPC UA Browsing Control**, 408-410  
**Generic OPC-UA Browsing Dialog**, 400-406  
**Generic Types**, 412  
**Generic Types for OPC-DA**, 412-413  
**German**, 28-30  
**Get an OPC Property on Custom Event**, 366  
**GetBaseException**, 80-81  
**GetConditionState**, 260-263 , 252-254  
**GetItemPropertyRecord**, 415-417

**GetMonitoredItemArguments**, 221-227  
**GetMonitoredItemArgumentsDictionary**, 221-227  
**GetMultiplePropertyValues**, 415-417 , 111-117  
**GetName**, 165-166  
**GetPropertyType**, 165-166  
**GetPropertyValue**, 165-166 , 414 , 111-117  
**GetPropertyValueArrayOfXXXX**, 413  
**GetPropertyValueDictionary**, 415-417  
**GetPropertyValueXXXX**, 413  
**Getting Condition State**, 252-254  
**Getting OPC Classic Property Values**, 111-117  
**Getting Started**, 17 , 28-30  
**Getting Started with OPC Classic under .NET**, 17  
**Getting Started with OPC Classic under COM**, 23  
**Getting Started with OPC UA under .NET**, 20  
**Getting Started with OPC UA under COM**, 24  
**GetXXXXPropertyValue**, 414-415  
**Global Assembly Cache**, 423-424 , 420 , 33-35 , 28-30  
**Global Discovery Server**, 172-173  
**Green color**, 63-64  
**Guid**, 439-440 , 85-86  
**Hardware**, 27  
**HasChild**, 94  
**HasComponent**, 94  
**HasDataValueInfo**, 84  
**HasProperty**, 94  
**HasValue**, 445  
**help**, 44  
**Helper Types**, 82  
**hexadecimal**, 36-37  
**HierarchicalReferences**, 94  
**HoldPeriods**, 90-91 , 248-251 , 291  
**HostParameters**, 248-251  
**How it Works**, 456 , 339-340  
**How the Binding Extender Automatically Goes Online and Offline**, 359  
**HRESULT**, 80-81  
**http**, 42 , 91-92 , 42  
**https**, 91-92  
**IA-64**, 441-442

**IDataConverter**, 363-364 , 352-353  
**Identifying information in OPC XML**, 91  
**Identifying Information in OPC-UA**, 92  
**IDispatch**, 36-37  
**IE**, 475-476 , 478-479  
**IEasyAEClient**, 67-68  
**IEasyDAClient**, 67-68  
**IEasyUAClient**, 67-68 , 292  
**IFormattable**, 483  
**IID**, 481-483  
**IIS**, 476-477 , 478-479  
**Importing Type Libraries to Delphi**, 58-62  
**Imports**, 55-57  
**Included Software**, 27  
**IncludeSubtypes**, 173-180  
**Index Range Lists**, 95-96  
**IndexRangeList**, 95-96 , 323-324  
**Industrial Gadgets .NET**, 37-38 , 472-473  
**Infinite**, 82  
**InfoType**, 84  
**InnerException**, 80-81  
**Inno Setup**, 423-424  
**Input**, 330-332  
**InputOutputs**, 381-384  
**Inputs**, 372-373 , 407-408 , 408-410  
**Inputs/Outputs**, 372-373  
**InputsOutputs**, 407-408 , 408-410 , 400-406  
**InputTypeCodes**, 231-248  
**InputTypeFullName**, 231-248  
**InputTypes**, 231-248  
**Install**, 421-422 , 98-101  
**Installation**, 27 , 31-32 , 28-30  
**Installation and Getting Started**, 449-450  
**Installing a StreamInsight Instance**, 450-451  
**installtoggac**, 423-424  
**InstanceParameters**, 90-91 , 172-173  
**Instrumentation .NET**, 472-473  
**Instrumentation Controls**, 37-38

**Int16, 437-439 , 439-440**  
**Int32, 437-439 , 439-440**  
**Int64, 437-439 , 439-440**  
**Integer, 94**  
**Integration Examples, 472-473**  
**IntelliSense, 455-456 , 31-32 , 35**  
**Interface Names in COM, 63**  
**Interfaces and Extension Methods, 67-68**  
**Internal Optimizations, 435**  
**InternalValue, 84**  
**Internet, 48-49**  
**Internet Explorer, 475-476 , 478-479**  
**Internet Information Server, 478-479**  
**Internet Information Services, 476-477**  
**Introduction, 15 , 449 , 418**  
**InvalidOperationException, 69-80**  
**Invoking the Operations, 333-334**  
**IObserver, 460 , 369-370 , 370 , 371 , 371**  
**IsBad, 82-83**  
**IsDataVariable, 418**  
**ISerializable, 434-435**  
**IsGood, 82-83**  
**Isolated, 68**  
**Isolated Clients, 68**  
**IsolatedParameters, 90-91**  
**IsProperty, 418**  
**IsServer, 418**  
**IsUncertain, 82-83**  
**ItemChanged, 445 , 445-446 , 69-80 , 227-228 , 180-186**  
**ItemDescriptor, 362-363 , 365-366**  
**ItemId, 324-326 , 164-165 , 85-86 , 92 , 92**  
**ItemIdTemplateString, 324-326**  
**ItemName, 92**  
**ItemPath, 92 , 92**  
**ItemReadParametersTemplate, 352**  
**ItemSubscribeParametersTemplate, 352**  
**ItemType, 323-324**  
**ItemWriteParametersTemplate, 352**  
**IXmlSerializable, 434-435**

**JScript, 16 , 475-476 , 57**  
**JScript Examples (IE, WSH), 475-476**  
**keep-alive interval, 434**  
**KeepAliveIntervalDebug, 434**  
**KerberosTokenInfo, 251**  
**Kind, 407-408 , 408-410 , 323-324**  
**LAN, 47 , 46 , 48-49**  
**late binding, 36-37**  
**Launcher, 33**  
**Lazarus, 475**  
**Lib, 24-26 , 23-24 , 57**  
**license, 423 , 422-423**  
**License Manager, 423-424 , 421-422 , 422 , 423 , 43 , 28 , 55 , 421 , 421**  
**License Manager Console, 421-422**  
**License Store, 422-423**  
**LicenseException, 80-81**  
**LicenseFile, 423-424**  
**licensemanager, 423-424**  
**LicenseManager.exe, 421-422 , 421**  
**licenses.licx, 55**  
**Licensing, 28 , 55 , 450 , 421**  
**Licensing in Visual Studio, 55**  
**LimitBitField, 82-83**  
**LimitInfo, 84**  
**LinearConverter, 353**  
**LinkParameters, 291**  
**LINQ, 457-459**  
**LINQPad, 474 , 41 , 31-32**  
**LINQPad Examples, 474**  
**LINQPad support, 41**  
**List of Examples, 466-467**  
**List of Packages, 32**  
**ListBox, 344-345**  
**Literal, 302-306**  
**Live Binding, 17 , 20 , 17-18**  
**Live Binding Demo, 37-38 , 38-40**  
**Live Binding Details, 358-359**  
**Live Binding in the Designer, 354**

Live Binding Model, 50 , 51-52 , 343  
Live Binding Model for OPC Data (Classic and UA), 343  
Live Binding Overview, 343-344  
Live Mapping Example, 313-319  
Live Mapping Model, 50 , 52-53 , 313  
Live Mapping Model for OPC Data (Classic and UA), 313  
Live Mapping Overview, 319  
LMConsole Utility (License Manager Console), 43-44  
LMConsole.exe, 423-424 , 421-422 , 421  
Local, 62  
LocalizedText, 439-440  
Location, 376-377 , 91-92  
log file, 31  
LogEntry, 429-434  
LogEntryEventArgs, 429-434  
MachineParameters, 248-251 , 291  
Making a first COM application, 23-24  
Making a first OPC Classic application using Live Binding, 17-18  
Making a first OPC Classic application using traditional coding, 18-20  
Making a first OPC UA application using Live Binding, 20-21  
Making a first OPC UA application using traditional coding, 21-22  
Making a first UA application, 24-26  
managed C++, 16  
Manual Assembly Referencing, 53-54  
Manual Deployment, 421  
Mapped Node Classes, 327-328  
Mapping Arguments and Phases, 330-332  
Mapping Context, 333  
Mapping Event Members, 323  
Mapping Kinds, 328-330  
Mapping Method Members, 322-323  
Mapping Operations, 328  
Mapping Property and Field Members, 322  
Mapping Sources, 319  
Mapping Tags, 327  
Mapping Targets, 319  
Mapping Your Objects, 332-333  
Mappings, 319-320  
MappingTag, 320-321 , 327

**Map-through, OPC Nodes and OPC Data, 323-324**  
**MaximumAge, 117-128**  
**Member Mapping, 322**  
**MERGETASKS, 423-424**  
**Message, 80-81**  
**MessageSecurityPreference, 97**  
**Meta- Members, 326-327**  
**MetaMember, 320-321 , 326-327**  
**Microsoft .NET Framework, 33-35 , 421 , 27-28 , 421**  
**Microsoft Help Viewer, 44**  
**Microsoft Reactive Extensions, 368**  
**Microsoft Windows, 27**  
**Microsoft Windows Server, 27**  
**Mode, 372-373 , 381-384 , 407-408 , 408-410 , 400-406 , 248-251 , 291**  
**Modifying Information, 137 , 254 , 292**  
**MonitoredItem, 63**  
**MonitoredItemParameters, 248-251**  
**MonitoringParameters, 228-229**  
**Multiple, 69**  
**MultiSelect, 381-384 , 385-386 , 388-389 , 386-387 , 377-380**  
**Multithreading and Synchronization, 440-441**  
**Name, 326-327 , 94**  
**namespace, 94 , 93-94**  
**namespace index, 94**  
**Namespace indices in Node Ids, 94**  
**Namespaces, 55-57**  
**NamespaceUriString, 94**  
**Naming Conventions, 62-63**  
**NetworkSecurity, 91-92**  
**Node IDs, 93-94**  
**Node Types, 340-342**  
**NodeClass, 418**  
**NodeDescriptor, 327-328 , 228-229 , 385-386 , 380-381 , 396-400 , 362-363**  
**NodeDescriptors, 377-380**  
**NodeDoubleClick, 407-408 , 408-410**  
**NodeElement, 377-380 , 396-400**  
**NodeElements, 385-386 , 377-380**  
**NodeId, 324-326 , 87-88 , 439-440**

**NodeIdTemplateString**, 324-326  
**NodePath**, 92  
**Nodes and Items**, 92  
**NormalColor**, 353-354  
**Notification**, 69-80 , 285-290 , 268-272  
**Notification Event**, 285-290 , 306-310  
**Notification Input**, 330-332  
**Notification Output**, 330-332  
**Notifiers**, 295  
**NotNull**, 35  
**NuGet**, 54-55 , 27 , 32 , 31-32 , 473-474 , 368 , 32 , 32  
**NuGet package**, 32 , 32  
**NuGet packages**, 35 , 32 , 31-32  
**Number**, 94  
**NumericUpDown**, 365-366  
**Object**, 437-439 , 111-117 , 180-186 , 446-447 , 107-110  
**Object Browser**, 455-456 , 35  
**Object Pascal**, 16 , 476  
**Object Pascal Examples (Delphi)**, 476  
**Object Serialization**, 434-435  
**ObjectId**, 85-86  
**ObjectMemberLinkingTarget**, 319  
**objects**, 64-66  
**ObjectsFolder**, 94  
**ObjectTypesFolder**, 94  
**observable**, 456 , 369-370 , 371  
**observer**, 370 , 371  
**Obsolete Live Binding Features**, 367  
**Obtaining Information**, 102 , 252 , 292  
**Obtaining Subscription Information**, 221-227  
**offline**, 359  
**OfflineEventSource**, 359  
**Old Project Conversion**, 367  
**OleView**, 36-37  
**On Error Goto 0**, 80-81  
**On Error Resume Next**, 80-81  
**OnCompleted**, 369-370 , 370 , 371 , 371  
**OneDimension**, 95-96  
**OnError**, 460 , 369-370 , 370 , 371 , 371

**OneShotShelve**, 294  
**Online**, 360 , 361 , 359  
**Online Design**, 358  
**OnlineEventSource**, 346-347 , 359  
**OnNext**, 369-370 , 370 , 371 , 371  
**OPC "Classic" via UA Wrapper**, 427  
**OPC Alarms and Events**, 449 , 425 , 15-16  
**OPC Classic Browse Path Format**, 87  
**OPC Classic Controls**, 406-407  
**OPC Classic Item Changed Event or Callback**, 227-228  
**OPC Classic or OPC UA thick-client COM applications on LAN**, 47  
**OPC Classic thick-client .NET applications on LAN**, 46  
**OPC Classic Web applications (server side)**, 47-48  
**OPC Common**, 425  
**OPC Common Dialogs**, 372-373 , 372  
**OPC Computer and Server Dialog**, 377 , 385  
**OPC Controls**, 406 , 372  
**OPC Core Components**, 442  
**OPC Data Access**, 437-439 , 449 , 425 , 15-16  
**OPC Data Observables**, 369-370  
**OPC Data Observers**, 370  
**OPC Event Payloads**, 460-461  
**OPC Foundation**, 448 , 425  
**OPC Interoperability**, 425-426  
**OPC Labs Base Library Core**, 33-35 , 58 , 58 , 36-37 , 53  
**OPC Labs Base Library Forms**, 33-35 , 58 , 58 , 36-37 , 53  
**OPC Labs EasyOPC "Classic" Library**, 58  
**OPC Labs EasyOPC "Classic" Library**, 58 , 36-37 , 53  
**OPC Labs EasyOPC Forms**, 58 , 58 , 36-37 , 53  
**OPC Labs EasyOPC-UA Library**, 58  
**OPC Observables**, 459-460  
**OPC Reactive Extensions**, 449  
**OPC Reactive Extensions (Rx/OPC)**, 368-369 , 371  
**OPC Server Dialog**, 376-377 , 385  
**OPC Specifications**, 425  
**OPC UA Attribute Data**, 84  
**OPC UA Certificate Generator**, 442  
**OPC UA Discovery**, 166

**OPC UA Global Discovery, 166 , 172-173**  
**OPC UA Local Discovery, 166 , 166-172**  
**OPC UA Monitored Item Changed Event or Callback, 228-229**  
**OPC UA Sample Server, 42**  
**OPC UA Status Code, 84**  
**OPC UA Thick-client applications on LAN, WAN or Internet, 48-49**  
**OPC UA Web applications (server side), 49-50**  
**OPC Unified Architecture, 449 , 425 , 15-16**  
**OPC XML, 91-92**  
**OPC XML Sample Server, 42**  
**OPC XML-DA, 92 , 425 , 92 , 15-16**  
**opc.tcp, 42 , 42**  
**Opc.Ua.CertificateGenerator.exe, 98-101**  
**opc.xmlda, 91-92**  
**OPC-A&E Area or Source Dialog, 385-386**  
**OPC-A&E Attribute Dialog, 388-389**  
**OPC-A&E Category Condition Dialog, 387-388**  
**OPC-A&E Category Dialog, 386-387**  
**OPC-A&E Common Dialogs, 384**  
**OPC-A&E Observables, 371**  
**OPC-A&E Observers, 371**  
**OPC-A&E Queries, 417**  
**OpcBrowseControl, 407-408**  
**OpcBrowseDialog, 381-384 , 389 , 407-408**  
**OpcComplexEventProcessing, 455**  
**OPC-DA Common Dialogs, 373**  
**OPC-DA Item Dialog, 377-380**  
**OPC-DA Property Dialog, 380-381**  
**OpcElementType, 381-384**  
**OPCEnum, 442**  
**OpcException, 80-81 , 81-82 , 82 , 444**  
**OpcLabs.BaseLib, 33-35**  
**OpcLabs.BaseLib.dll, 420 , 420 , 53**  
**OpcLabs.BaseLib.tlb, 36-37 , 57**  
**OpcLabs.BaseLibForms, 33-35**  
**OpcLabs.BaseLibForms.dll, 420 , 420 , 53**  
**OpcLabs.BaseLibForms.tlb, 36-37**  
**OpcLabs.EasyOpcClassic, 33-35**  
**OpcLabs.EasyOpcClassic.dll, 420 , 420 , 53**

**OpcLabs.EasyOpcClassic.tlb**, 36-37  
**OpcLabs.EasyOpcForms**, 33-35  
**OpcLabs.EasyOpcForms.dll**, 420 , 420 , 53  
**OpcLabs.EasyOpcForms.tlb**, 36-37  
**OpcLabs.EasyOpcUA**, 33-35  
**OpcLabs.EasyOpcUA.dll**, 420 , 420 , 53  
**OpcLabs.EasyOpcUA.tlb**, 36-37  
**OPCLabs.KitServer.2**, 41-42  
**OpcLabs.QuickOpc**, 54-55 , 32 , 32  
**OpcLabs.QuickOpc.Forms**, 32  
**OpcServerDialog**, 376-377 , 385  
**OPC-UA Browse Path Format**, 88-90  
**OPC-UA Common Dialogs**, 389  
**OPC-UA Controls**, 408  
**OPC-UA Data Dialog**, 396-400  
**OPC-UA Endpoint Dialog**, 389-393  
**OPC-UA Global Discovery Server**, 172-173  
**OPC-UA Host and Endpoint Dialog**, 393-396  
**OPC-UA Modelling (Preliminary)**, 339  
**OPC-UA Security**, 97  
**OPC-UA via UA Proxy**, 426-427  
**opcua.demo-this.com**, 42 , 42  
**opcxml.demo-this.com/**, 42  
**Operand**, 299-302  
**Operands**, 299-302  
**Operating Systems**, 27  
**Operation Monitoring and Control**, 312  
**OperationArguments**, 84 , 69  
**OperationEventArgs**, 445  
**OperationResult**, 445 , 444-445 , 81-82 , 84-85 , 251-252 , 69 , 143-153  
**Operations**, 328 , 323-324  
**Options**, 449  
**Orange color**, 63-64  
**Organizes**, 94  
**Other Special Cases in OPC Sequences**, 460  
**OutOfMemoryException**, 444-445 , 69-80 , 444  
**Output**, 330-332  
**outputs**, 350-351 , 372-373 , 381-384 , 407-408 , 408-410 , 400-406

**Overflow, 84**  
**Overview of the Assemblies Available, 53**  
**package manager, 31-32**  
**Package Manager Console, 32**  
**Package Manager GUI, 32**  
**ParallelDiscovery, 166-172**  
**Parameter Objects, 90-91**  
**Parameter Templates, 352**  
**Parameters, 90-91 , 350**  
**ParentItemID, 324-326**  
**ParentNodeID, 324-326**  
**ParentNodePath, 324-326**  
**Password, 251**  
**Payloads for OPC Alarms and Events, 462-463**  
**Payloads for OPC Data Access, 461-462**  
**Payloads for OPC Unified Architecture, 463-464**  
**percent deadband, 180-186**  
**Perl, 476**  
**Perl Examples, 476**  
**PHP, 16 , 476-477**  
**PHP Examples, 476-477**  
**php\_com\_dotnet, 476-477**  
**Point Binder, 347**  
**Point Editor, 348-350**  
**PointBinder, 362 , 366 , 346 , 357-358 , 347-348 , 366 , 360-361 , 51-52 , 347 , 352 , 354 , 365**  
**PointBinding, 350-351 , 366 , 347 , 360 , 362-363 , 359**  
**PointBindingGroup, 360 , 359**  
**PointBindingOperations, 345**  
**Points, 348**  
**portable, 477-478**  
**Portable C++ Examples, 477-478**  
**Prerequisites, 27-28 , 421**  
**Prerequisites Boxing, 442**  
**Procedural Coding, 17 , 20 , 18-20 , 21-22**  
**Procedural Coding Model, 50 , 50-51 , 102**  
**Procedural Coding Model for OPC Classic A&E, 252**  
**Procedural Coding Model for OPC Data (Classic and UA), 102**  
**Procedural Coding Model for OPC UA Alarms & Conditions, 291-292**  
**ProcedureCallException, 69-80**

**Product Parts**, 455 , 33  
**production installer**, 423-424  
**productioncom**, 423-424  
**productionnet**, 423-424  
**ProductUriString**, 427-429 , 98-101  
**ProgID**, 160-161 , 258-260 , 442 , 85-86 , 63 , 481-483  
**ProgIDs (COM)**, 63  
**Programmatic Access to Live Binding**, 360  
**Propagated Parameters**, 321-322  
**Properties**, 92  
**PropertyDescriptor**, 366  
**PropertyElement**, 380-381  
**PropertyId**, 165-166 , 92  
**Providing Client Instance Certificate**, 98-101  
**PullDataChangeNotification**, 69-80  
**PullDataChangeNotificationQueueCapacity**, 69-80  
**PullEventNotification**, 69-80  
**PullItemChanged**, 69-80  
**PullMultipleDataChangeNotifications**, 69-80  
**PullMultipleEventNotifications**, 69-80  
**PullMultipleItemChanges**, 69-80  
**PullMultipleNotifications**, 69-80  
**PullMultipleServerConditionChanges**, 69-80  
**PullMultipleXXXX**, 69-80  
**PullNotification**, 69-80  
**PullServerConditionChanged**, 69-80  
**PullXXXX**, 69-80  
**PullXXXXQueueCapacity**, 69-80  
**Python**, 478  
**Python Examples**, 478  
**pywin32**, 478  
**Qualified Names**, 94  
**QualifiedName**, 165-166 , 439-440 , 92  
**QualifiedNames**, 299-302  
**Quality**, 445  
**Quality in OPC Classic**, 82-83  
**QualityChoiceBitField**, 82-83  
**QueryCategoryAttributes**, 268

**QueryCategoryConditions**, 266  
**QueryEventCategories**, 263-266  
**Querying for OPC Event Attributes**, 268  
**Querying for OPC Event Categories**, 263-266  
**Querying for OPC Event Conditions on a Category**, 266  
**Querying for OPC Event Conditions on a Source**, 266-268  
**QueryServers**, 172-173  
**QuerySourceConditions**, 266-268  
**Queued Execution**, 352  
**QueuedExecution**, 352  
**QuickOPC User's Guide** , 0  
**QuickStart Alarm Condition Server (OPC UA)**, 42  
**reactive programming**, 368  
**Reactive Programming Examples**, 473-474  
**Reactive Programming Model**, 50 , 53 , 368  
**Reactive Programming Model for OPC Alarms and Events**, 370-371  
**Reactive Programming Model for OPC Data (Classic and UA)**, 368  
**Read**, 117-128  
**Read On Custom Event**, 362-363  
**ReadEventArgs**, 366 , 362-363 , 359  
**Reading Attributes of OPC UA Nodes**, 117-128  
**Reading from OPC Classic Items**, 102-107  
**Reading in OPC XML-DA**, 110-111  
**Reading just the value**, 107-110 , 128-137  
**ReadItem**, 161-164 , 413 , 102-107 , 107-110  
**ReadItemValue**, 413 , 69 , 107-110  
**ReadItemValueArrayOfXXXX**, 413  
**ReadItemValueXXXX**, 413  
**readme.txt**, 32  
**ReadMultiple**, 117-128 , 84-85  
**ReadMultipleItems**, 444-445 , 102-107 , 84-85  
**ReadMultipleItemValues**, 69 , 107-110  
**ReadMultipleValues**, 69 , 128-137  
**ReadParameters**, 117-128  
**ReadValue**, 69 , 128-137  
**REAL Studio**, 478  
**REALbasic**, 478  
**REALbasic (Xojo) Examples**, 478  
**Redist**, 442

reference, 32 , 481  
**ReferencesInverse**, 87-88  
**ReferenceTypeIds**, 173-180  
**ReferenceTypesFolder**, 94  
Referencing the Assemblies, 456-457  
Referencing the Assemblies (.NET), 53  
Referencing the Components (COM), 57  
Refresh, 285-290 , 306-310 , 306 , 268-272  
refresh when active, 268-272  
RefreshComplete, 285-290 , 306-310 , 306  
RefreshEventSubscription, 283-285  
Refreshing Condition States, 283-285 , 306  
RefreshInitiated, 306-310 , 306  
Regasm.exe, 420  
REGEDIT, 423  
registry, 423 , 422-423  
Related Products, 28  
RequestedDataType, 323-324 , 365-366  
Reset Toolbox, 64-66  
ReSharper, 35  
ReSharper annotations, 455 , 35  
Respond, 292  
Result Objects, 84-85  
results, 350-351  
RetainAppearance, 372-373  
RetrialDelay, 312  
RetryAll, 435-436  
RevertAppearance, 372-373  
RootFolder, 94  
Running the Setup, 28-30  
Runtime assemblies, 31-32  
Rx, 368  
Rx/OPC, 449 , 368-369 , 371  
**SAFEARRAY**, 111-117  
**SByte**, 439-440  
security, 97  
Security in Endpoint Selection, 97  
SecurityPolicyUriString, 97

Select clauses, 298-299 , 299-302  
SelectClauses, 298-299  
SelectedName, 373-376  
SelectElementType, 381-384 , 400-406  
SelectionChanged, 407-408 , 408-410  
SelectionDescriptors, 381-384 , 407-408 , 408-410  
SelectionElements, 381-384 , 407-408 , 408-410  
SemanticsChanged, 84  
Serializable, 434-435  
Server, 320-321 , 94  
Server Diagnostics in OPC-UA, 251-252  
Server Endpoints, 92-93  
ServerClass, 160-161 , 85-86 , 91-92  
ServerConditionChanged, 69-80 , 312  
ServerDescriptor, 160-161 , 258-260 , 85-86 , 366 , 111-117 , 326-327 , 385-386 , 388-389 , 387-388 , 386-387 , 377-380 , 380-381 , 362-363 , 102-107 , 91-92 , 180-186 , 137-141 , 107-110  
ServerElement, 160-161 , 258-260 , 85-86 , 85  
ServerElementCollection, 160-161 , 258-260  
ServerFamilies, 376-377 , 385  
Servers, 91-92  
SessionParameters, 251-252 , 92-93 , 248-251  
SetQualityAndSubStatus, 82-83  
Setting Parameters, 248-251 , 291 , 311-312  
Setup, 423-424 , 27 , 28-30  
Setup Log, 31  
Setup Program, 28  
Severity, 84  
Shared Instance, 68  
SharedInstance, 68  
SharedParameters, 90-91  
ShowDialog, 373-376 , 376-377 , 385-386 , 387-388 , 386-387 , 377-380 , 380-381 , 396-400 , 389-393  
ShowListBranches, 381-384 , 407-408 , 408-410 , 400-406  
SILENT, 423-424  
SimpleAttribute, 302-306  
Simulated, 407-408 , 408-410 , 400-406  
Simulation Server for OPC Classic, 41-42  
Simultaneous Operations, 69  
Single, 437-439 , 439-440  
SizeFactor, 372-373

**Software Toolbox Extender, 418**  
**Software Toolbox Extender Replacement, 418**  
**SourceComponent, 366 , 346-347 , 366 , 362-363 , 365-366 , 365 , 365**  
**SourceElements, 385-386**  
**SourceMember, 346-347 , 366 , 362-363 , 365-366 , 365**  
**SourcePath, 346-347**  
**Specifying Event Filters, 298-299 , 272-276**  
**SQL Server, 450-451 , 449**  
**StackOverflowException, 444-445 , 444**  
**Standard Node IDs, 94**  
**StandardName, 90-91**  
**Start menu, 28-30 , 30-31 , 33**  
**StartingNodeId, 87-88**  
**State, 285-290 , 306-310 , 228-229 , 84-85 , 180-186 , 268-272 , 446-447 , 299-302**  
**StatusBitField, 82-83**  
**StatusCode, 445 , 439-440**  
**StatusInfo, 363-364 , 353-354**  
**StatusToColorConverter, 363-364 , 353-354**  
**STLSoft, 477-478**  
**streaminsight, 423-424 , 449 , 449**  
**StreamInsight Event Flow Debugger, 452-455**  
**StreamInsight Option, 449**  
**String, 437-439 , 439-440 , 353-354**  
**string format, 352-353**  
**String Formatting, 363**  
**StructureChanged, 84**  
**Subscribe, 313-319 , 368-369**  
**Subscribe & Write, 365-366**  
**SubscribeDataChange, 295-298 , 186-200**  
**SubscribeEvent, 295-298**  
**SubscribeEvents, 260-263 , 268-272 , 444**  
**SubscribeItem, 161-164 , 180-186 , 444**  
**SubscribeMonitoredItem, 295-298 , 186-200 , 310-311**  
**SubscribeMultipleItems, 180-186**  
**SubscribeMultipleMonitoredItems, 295-298 , 186-200**  
**SubscribeXXXX, 446-447**  
**Subscribing to Information, 180 , 268 , 295**  
**Subscribing to OPC Classic Items, 180-186**  
**Subscribing to OPC Events, 268-272 , 295-298**

**Subscribing to OPC UA Monitored Items, 186-200**  
**SubscriptionParameters, 228-229 , 248-251**  
**SUCCEEDED, 80-81 , 460**  
**SwtbExtenderReplacement, 418**  
**Symbol Factory .NET, 37-38 , 472-473**  
**synchronization context, 440-441**  
**SynchronizationContext, 440-441 , 248-251 , 68**  
**TargetAccessException, 360-361**  
**TargetComponent, 360-361 , 344-345**  
**targeting pack for .NET Framework, 16**  
**TargetMember, 363-364 , 364 , 366 , 364 , 344-345**  
**TargetName, 87-88**  
**TASKS, 423-424**  
**Terminology, 15-16**  
**Test Servers, 41**  
**Test Tools, 44-45**  
**The Launcher application, 33**  
**The LinearConverter Component, 353**  
**The Mapper Object, 332**  
**The Select clauses, 299-302**  
**The StatusToColorConverter Component, 353-354**  
**The Where clause, 302-306**  
**thick-client, 47 , 46 , 48-49**  
**thread, 440-441**  
**thread-safe, 440-441**  
**Time in OPC and StreamInsight, 464-465**  
**Time in OPC Subscriptions, 465-466**  
**Time Periods, 82**  
**Time Synchronization in OPC, 465**  
**TimedShelve, 294**  
**Timeout, 82**  
**Timeout Handling, 436-437**  
**Timeouts, 90-91 , 248-251 , 436-437**  
**Timestamp, 62**  
**TimestampLocal, 62**  
**Toolbox, 54 , 64-66 , 17-18 , 18-20 , 20-21 , 21-22 , 354 , 68**  
**Toolbox Items, 354**  
**ToolTip, 364 , 344-345**

**ToolTip and Other Extenders, 364**  
**TopicParameters, 248-251**  
**TrackBar, 365-366**  
**trial license, 28**  
**Trigger, 228-229 , 186-200**  
**troubleshooting, 31**  
**Troubleshooting the Setup, 31**  
**TrustedEndpointUrlStrings, 97-98**  
**TrustedPeersCertificateStore, 97-98**  
**Trusting Server Instance Certificate, 97-98**  
**TryParse, 95-96**  
**Turquoise color, 63-64**  
**TYPE, 423-424**  
**type library, 36-37**  
**Type Mapping, 322**  
**Type Names, 63**  
**TypeCode, 153**  
**TypedAttributeData, 460**  
**TypedValue, 460**  
**TypedVtq, 412-413 , 460**  
**TypeId, 299-302**  
**typeinfo, 36-37**  
**Type-less Mapping, 334-338**  
**Type-safe Access, 413 , 413**  
**TypesFolder, 94**  
**Typical Binding Scenarios, 361-362**  
**Typical Usage, 46**  
**UA, 63**  
**UA Generic Client, 425**  
**UAApplicationElement, 85-86 , 85 , 166-172 , 172-173**  
**UAApplicationElementCollection, 166-172 , 172-173**  
**UAApplicationTypes, 418 , 166-172**  
**UAAttributeArguments, 95 , 95-96**  
**UAAttributeData, 445 , 439-440 , 319-320 , 434-435 , 84 , 228-229 , 117-128**  
**UAAttributeDataPayload, 463-464**  
**UAAttributeDataResult, 445 , 117-128 , 84-85 , 353-354**  
**UAAttributeField, 299-302**  
**UAAttributeFieldCollection, 298-299**  
**UAAttributId, 95**

**UAAttributePoint**, 348  
**UABaseEventObject**, 299-302  
**UABrowseControl**, 408-410  
**UABrowseDialog**, 400-406  
**UABrowseParameters**, 295 , 295 , 173-180  
**UABrowsePath**, 87-88 , 88-90  
**UABrowsePathElement**, 87-88  
**UABrowsePathElementCollection**, 299-302  
**UACallArguments**, 231-248  
**UACertificateAcceptancePolicy**, 97-98  
**UAClientDataMapping**, 328 , 319-320  
**UAClientDataMappingSource**, 319 , 319-320  
**UAClientEngine**, 427-429  
**UAClientEngineException**, 80-81  
**UAClientMapper**, 327  
**UACodeBits**, 84  
**UAConfigurationSources**, 427-429  
**UAConnectivity**, 347-348 , 51-52 , 20-21 , 352 , 348 , 354  
**UAContentFilterElement**, 298-299 , 302-306  
**UAContentFilterElementBuilder**, 302-306  
**UAData**, 320-321 , 328-330 , 328 , 323-324  
**UADataChangeFilter**, 320-321 , 228-229 , 186-200  
**UADataChangeNotificationObservable**, 369-370 , 459-460  
**UADataChangeNotificationPayload**, 463-464  
**UADataChangeTrigger**, 186-200  
**UADialog**, 396-400  
**UADataMappingOperations**, 328  
**UADatatypes**, 94  
**UADiscoveryParameters**, 166-172  
**UADiscoveryQuery**, 173  
**UAElementType**, 400-406  
**UAEndpoint**, 320-321  
**UAEndpointDescriptor**, 85-86 , 326-327 , 92-93 , 251 , 166-172 , 172-173  
**UAEndpointDialog**, 389-393  
**UAEndpointDiscoveryQuery**, 173  
**UAEndpointSelectionPolicy**, 97 , 92-93  
**UAEventData**, 306-310  
**UAEventFilter**, 298-299 , 295-298

**UAEventFilterBuilder**, 298-299  
**UAException**, 80-81 , 81-82 , 82 , 444  
**UAFilterElements**, 302-306  
**UAFilterOperand**, 302-306  
**UAFilterOperandCollection**, 302-306  
**UAFilterOperator**, 302-306  
**UAGlobalDiscoveryQuery**, 173  
**UAHostAndEndpointDialog**, 393-396  
**UAHostParameters**, 166-172  
**UAIndexRange**, 95-96  
**UAIndexRangeList**, 95-96  
**UALocalDiscoveryQuery**, 173  
**UAMappedNode**, 327-328 , 322  
**UAMember**, 320-321 , 323-324  
**UAMethodIds**, 94  
**UAMonitoredItemArguments**, 306  
**UAMonitoring**, 320-321 , 322  
**UAMonitoringParameters**, 326-327 , 295-298 , 186-200  
**UANamespace**, 320-321  
**UANode**, 320-321 , 324-326 , 327 , 323-324  
**UANodeArguments**, 117-128  
**UANodeDescriptor**, 87-88 , 173-180 , 85-86 , 326-327 , 93-94  
**UANodeElement**, 173-180 , 85-86 , 85 , 418  
**UANodeElementCollection**, 173-180 , 434-435  
**UANodeId**, 439-440 , 93-94  
**UANodeIdTemplate**, 320-321 , 324-326  
**UAObjectIds**, 94  
**UAObjectTypeIds**, 94  
**UAQualifiedName**, 439-440 , 94  
**UAQualifiedNameCollection**, 299-302  
**UAQueryServerFilter**, 172-173  
**UARead**, 320-321  
**UAReadArguments**, 95 , 95-96 , 117-128 , 128-137  
**UAReadParameters**, 326-327  
**UAReferenceTypeIds**, 94  
**UASecurityPolicyUriStrings**, 97  
**UAServiceException**, 80-81  
**UASimpleAttributeOperand**, 299-302  
**UAStatusCode**, 439-440 , 84 , 353-354

**UAStatusCodeException**, 80-81 , 128-137  
**UASubscription**, 320-321  
**UASubscriptionParameters**, 326-327 , 186-200  
**UAType**, 320-321 , 327-328 , 322  
**UAVariableIds**, 94  
**UAVariableTypeIds**, 94  
**UAWriteArguments**, 95 , 95-96 , 153  
**UAWriteResult**, 143-153  
**UAWriteValueArgument**, 143-153  
**UAWriteValueArguments**, 95 , 95-96 , 143-153 , 153  
**UAWriteValueObserver**, 370  
**UInt16**, 439-440  
**UInt32**, 439-440  
**UInt64**, 439-440  
**Unicode**, 437-439 , 439-440  
**Unified Architecture Extensions**, 418  
**unins000.exe**, 423-424  
**Uninstall Instructions**, 30-31  
**Uninstallation**, 30-31  
**Unit**, 58-62  
**UnknownColor**, 353-354  
**Unshelve**, 294  
**UnsubscribeAllEvents**, 279-283  
**UnsubscribeAllItems**, 209-221  
**UnsubscribeAllMonitoredItems**, 209-221 , 306  
**UnsubscribeEvents**, 279-283  
**UnsubscribeItem**, 209-221 , 444  
**UnsubscribeMonitoredItem**, 209-221 , 306  
**UnsubscribeMultipleItems**, 209-221  
**UnsubscribeMultipleMonitoredItems**, 209-221 , 306  
**UnsubscribeXXXX**, 446-447  
**Unsubscribing**, 209-221  
**Unsubscribing from OPC Events**, 279-283 , 306  
**UpdateFailure**, 360-361  
**UpdateOutputOnFailure**, 350-351  
**UpdateRates**, 248-251  
**Uri**, 92  
**UriString**, 92

**URL, 110-111**  
**UrlString, 92 , 91-92**  
**Usage, 456 , 412**  
**Usage Guidelines, 361**  
**Use generic or type-safe access wherever possible, 446**  
**Use multiple-operand methods instead of looping, 445**  
**Use the state instead of handles to identify subscribed entities, 446-447**  
**User Identity in QuickOPC-UA, 251**  
**User Interface, 372**  
**User interface components, 31-32**  
**User Interface Objects, 68**  
**UserIdentity, 251**  
**UserName, 251**  
**UserNameTokenInfo, 251**  
**UserPickEndpoint, 396-400**  
**using, 55-57**  
**Using Callback Methods Instead of Event Handler, 310-311**  
**Using Callback Methods Instead of Event Handlers, 229-231 , 290-291**  
**using namespace, 55-57**  
**Using Package Manager Console, 32**  
**Using Package Manager GUI, 32**  
**UTC, 62**  
**Utility Classes, 91**  
**Value, 445 , 95 , 84**  
**value conversion, 345-346 , 352-353**  
**Value Conversions, String Formats and Converters, 352-353**  
**value target, 343-344**  
**Value Targets, 344-345**  
**Value, Timestamp and Quality (VTQ) in OPC Classic, 83**  
**ValueAge, 102-107**  
**ValueResult, 445 , 412 , 306-310 , 84-85 , 107-110 , 128-137**  
**ValueTarget, 363-364 , 364 , 366 , 364**  
**ValueType, 323-324 , 153**  
**ValueTypeCode, 365-366 , 153**  
**VarEnum, 83-84**  
**VariableTypesFolder, 94**  
**VARIANT, 437-439 , 111-117 , 91 , 83-84 , 107-110**  
**Variant Type (VarType) in OPC Classic, 83-84**  
**Various Kinds of Binding, 364-365**

**VarType**, 91 , 83-84  
**VarType** , 437-439  
**VarTypes**, 83-84  
**VarTypeUtilities**, 91  
**VB**, 16  
**VB.NET**, 41  
**VB6**, 36-37  
**VBA**, 16 , 478  
**VBA Examples in Excel**, 478  
**VBScript**, 437-439 , 16 , 57 , 478-479  
**VBScript Examples (ASP, IE, WSH)**, 478-479  
**vendor name**, 258-260  
**version**, 442-443  
**Version Isolation**, 442-443  
**VERYSILENT**, 423-424  
**VFP**, 16  
**View**, 407-408 , 408-410  
**ViewsFolder**, 94  
**Visual Basic**, 36-37 , 16 , 468-471 , 471-472 , 17-18 , 479  
**Visual Basic 6.0**, 24-26 , 23-24 , 57 , 479  
**Visual Basic Examples (VB 6.0)**, 479  
**Visual Basic for Applications**, 16 , 57 , 478  
**Visual Basic.NET**, 16  
**Visual C#**, 468-471 , 471-472 , 449  
**Visual C++**, 468-471 , 471-472 , 479-480  
**Visual C++ Examples**, 479-480  
**Visual F#**, 468-471 , 471-472  
**Visual FoxPro**, 16 , 480  
**Visual FoxPro Examples**, 480  
**Visual Studio**, 16 , 455-456 , 44 , 27 , 354 , 17-18 , 18-20 , 20-21 , 21-22 , 449 , 479-480 , 35  
**Visual Studio Toolbox**, 54  
**VOLE**, 477-478  
**VT\_ARRAY**, 437-439  
**VT\_BOOL**, 437-439  
**VT\_BSTR**, 437-439  
**VT\_CY**, 437-439  
**VT\_DATE**, 437-439  
**VT\_DECIMAL**, 437-439

**VT\_DISPATCH, 437-439**  
**VT\_EMPTY, 437-439**  
**VT\_ERROR, 437-439**  
**VT\_I1, 437-439**  
**VT\_I2, 437-439**  
**VT\_I4, 437-439**  
**VT\_I8, 437-439**  
**VT\_INT, 437-439**  
**VT\_NULL, 437-439**  
**VT\_R4, 437-439**  
**VT\_R8, 437-439**  
**VT\_UI1, 437-439**  
**VT\_UI2, 437-439**  
**VT\_UI4, 437-439**  
**VT\_UI8, 437-439**  
**VT\_UINT, 437-439**  
**VT\_VARIANT, 437-439**  
**Vtq, 412-413 , 227-228 , 369-370 , 368-369 , 364**  
**VtqPayload, 457-459**  
**WAN, 48-49**  
**WarningColor, 353-354**  
**Web application, 47-48 , 49-50**  
**Well-known Properties, 414-415**  
**What Happens When a Binding Operation Method Is Called, 359-360**  
**What Happens When the Binding Extender Is Set Online or Offline, 359**  
**Where clause, 298-299 , 302-306**  
**Where do I go from here?, 20 , 22-23 , 26 , 24**  
**WhereClause, 298-299**  
**Windows, 449**  
**Windows 8, 33**  
**Windows Control Panel, 30-31**  
**Windows Forms, 16 , 17-18 , 18-20 , 20-21 , 21-22**  
**Windows Presentation Foundation, 440-441**  
**Windows Script Host, 16 , 475-476 , 478-479**  
**Windows Server, 449**  
**WindowsFormsSynchronizationContext, 440-441**  
**With single-operand synchronous methods, only catch OpcException or UAException, 444**  
**wizard, 28-30**  
**WPF, 16 , 440-441**

**Write, 153-159**  
**Write Group of Values on Custom Event, 365**  
**Write Single Value on Custom Event, 365**  
**WriteEventSource, 366 , 346-347 , 365-366 , 361 , 359 , 365 , 365**  
**WriteItem, 413 , 141-143**  
**WriteItemValue, 413 , 137-141**  
**WriteItemValueArrayOfXXXX, 413**  
**WriteItemValueXXXX, 413**  
**WriteMultiple, 153-159**  
**WriteMultipleItems, 141-143**  
**WriteMultipleItemValues, 137-141**  
**WriteMultipleValues, 143-153**  
**WriteParameters, 365-366**  
**WriteValue, 143-153**  
**Writing Attributes of OPC UA Nodes, 143-153**  
**Writing to OPC Classic Items, 137-141**  
**Writing value, timestamp and quality, 141-143**  
**Writing value, timestamps and status code, 153-159**  
**WScript, 478-479**  
**WSH, 58 , 475-476 , 57 , 478-479**  
**www.nuget.org, 27 , 31-32**  
**X1, 353**  
**X2, 353**  
**X509CertificateTokenInfo, 251**  
**x64, 442 , 441-442**  
**x86, 442 , 441-442**  
**Xbase++, 480**  
**Xbase++ Examples, 480**  
**XML comment files, 455**  
**XML Comments, 35**  
**XmlElement, 439-440**  
**XmlSerializer, 434-435**  
**Xojo, 478**  
**Y1, 353**  
**Y2, 353**